9-1-1981

Major Trends in Operating Systems Development

Margaret M. Reek

Rochester Institute of Technology
School of Computer Science and Technology

Major Trends
in
Operating Systems Development

A Thesis submitted in partial fulfillment of
Master of Science in Computer Science Degree Program

By: Margaret M. Reek

Approved By:

Michael J. Lutz: Advisor

Peter H. Lutz

Wiley R. McKinzie

Date: September 1, 1981

Table of Contents

1. Introduction and Overview

1.1. Introduction 1-1

1.2. Overview 1-1

2. Early History

3. Multiprogramming and Timesharing Systems

3.1. Introduction 3-1

3.2. Definition of Terms 3-1

3.3. Early Influences and Motivations 3-2

3.4. Historical Development 3-3

3.4.1. Early Concepts 3-3

3.4.2. Early Supervisors 3-3

3.4.3. The Growth Years - 1961 to 1964 3-7

3.4.4. The Close of the Era - 1965 to 1968 3-12

3.5. Contributions of the Era - 1957 to 1968 3-14

3.5.1. Hardware Refinements 3-14

3.5.2. Software Refinements 3-15

3.5.2.1. Scheduling 3-15

3.5.2.2. Resource Allocation 3-17

4. Virtual Memory Systems

4.1. Introduction 4-1

4.2. Definition of Terms 4-1

4.3. Early Influences and Motivations 4-2

Table of Contents

4.4. Historical Development 4-3

4.4.1. Early Concepts 4-3

4.4.2. Early Systems 4-4

4.4.3. The Growth Years - 1965 to 1970 4-8

4.5. Contributions of the Era - 1961 to 1970 4-18

4.5.1. Hardware Refinements 4-19

4.5.2. Software Refinements 4-21

5. Security and Protection Systems

5.1. Introduction 5-1

5.2. Definition of Terms 5-1

5.3. Early Influences and Motivations 5-2

5.4. Historical Development 5-3

5.4.1. Early Concepts 5-3

5.4.2. Early Systems 5-3

5.4.3. The Growth Years - 1965 to Present 5-5

5.4.4. Current Activity 5-20

5.5. Contributions of the Era 5-20

5.5.1. Hardware Refinements 5-20

5.5.2. Software Refinements 5-21

6. Distributed Processing Systems

6.1. Introduction 6-1

6.2. Definition of Terms 6-1

Table of Contents

6.3. Early Influences and Motivations 6-2

6.4. Early Concepts 6-3

6.5. Historical Development 6-4

6.5.1. Early Systems 6-4

6.5.2. The Growth Years - 1970 to Present 6-7

6.5.3. Current Activity 6-14

6.6. Contributions of the Era 6-19

6.6.1. Hardware Refinements 6-20

6.6.2. Software Refinements 6-21

7. Summary

Annotated Bibliography

Early History i

Multiprogramming and Timesharing Systems iv

Virtual Memory Systems xiii

Security and Protection Systems xviii

Distributed Processing Systems xxiv

Figures

Figure 4-1	Multics Address Translation	5-11
Figure 5-1	Multics Ring Structure and Operation	4-9
Figure 6-1	Cm* Configuratiion	6-12
Figure 6-2	Cambridge Model Distributed Ring	6-15
Figure 7-1	Summary of Major Developments	7-2

1. Introduction and Overview

1.1. Introduction

Operating systems have changed in nature in response to demands of users, and in response to advances in hardware and software technology. The purpose of this paper is to trace the development of major themes in operating system design from their beginnings through the present. This is not an exhaustive history of operating systems, but instead is intended to give the reader the flavor of the different periods in operating systems' development. To this end, the paper will be organized by topic in approximate order of development. Each chapter will start with an introduction to the factors behind the rise of the period. This will be followed by a survey of the state-of-the-art systems, and the conditions influencing them. The chapters close with a summation of the significant hardware and software contributions from the period.

1.2. Overview

Ever since the earliest computers were developed, people have tried to maximize their utilization. This desire led to the development of operating systems. These systems were primitive batch monitors in their early form, which matured into batch multiprogramming systems with the advent of interrupts. Multiprogramming systems became more complex as larger and faster storage systems became available, and timesharing grew out of these multiprogramming systems. Virtual memory was developed as a way to make these new storage technologies easily available to users. As more people gained easy, simultaneous access to computers, an awareness of the need for security grew. Concern over security has grown from the protection of the system supervisor from careless users to the protection of a distributed network from potentially hostile intruders. The concept of distributed

systems first developed as an offshoot of multiprogramming as a way to increase processing speed. It has now grown to include distant, independent computers linked together to allow sharing of programs and data. These developments have brought us to operating systems as we know them today.

2. Early History

Ever since electronic computers were developed in the late 1940s, people have looked for ways to make them more cost effective and easier to use. As knowledge about computers grew and the industry advanced, the means for achieving those goals became available. This chapter briefly traces the rise of automatic supervisors: the predecessors of modern operating systems.

The first programmers were presented with the monumental task taking a naked piece of computer hardware and making it do something useful. There were no tools to help them; all they could do was program it by using numeric machine code, enter the numbers by paper tape, and use the lights and switches on the console to debug and run their programs. As can be imagined, this was a time consuming and difficult task. Only a small population had the capability or inclination to get involved in the process.

This phase in software's history didn't last long. It occurred to people that the computer itself could be used to make programming easier. Granted, the computer required that instructions given it correspond to its internal representation, but that didn't necessarily mean that people had to program that way. Instead of programming in numbers, symbolic names could be given to the operations, and a program would translate the symbols into their numeric equivalents. Thus the first assemblers were born by the early 1950s.

Assemblers helped make programming easier, but there were still many hardware specifics to be dealt with. Every programmer who wanted to use the tape drive, line printer or any other input/output device had to include code to drive those devices in his programs. Not only did everyone have to recreate the I/O code, but the programs to control devices were complicated. Mistakes in I/O

programming were the most common sources of error, and the most difficult to debug. One has to consider that the time to set up the job, and assemble and load the program was many times longer than the time to run the job, especially if the program failed on an early input operation. To reduce the burden on the programmer and the time wasted debugging I/O drivers for commonly used devices, libraries of standard subroutines to perform these functions were developed. The library routines would be assembled with the rest of the program, and would be called to do the I/O.

The advent of symbolic addresses facilitated the use of libraries. In the earliest days, all programs were written using absolute addresses. If a programmer wanted to include someone else's subroutine in his program, the addresses in the subroutine would have to be changed. Relative addressing eased the problem by assuming that each routine started at address zero; and the system loader would add an offset to each instruction address to form the assigned memory address. Symbolic addressing made programming simpler since names, independent of physical addresses, could be used. The problem with absolute addresses (that the addresses changed whenever the program changed) was automatically resolved by the assembler when using symbolic addresses. These facilities were in common use by the middle 1950s.

By 1956 simple supervisor programs were being developed. If all the programmers were going to use the library routines, it made sense to have them reside on the machine so they would not have to be assembled with each job. The first supervisors were intended to force the user to use the library routines to control I/O.

Somewhat later supervisors added services to make job set up easier. Early manual scheduling of the computer was inefficient. Programmers would schedule time to run and debug their programs. While one programmer had the machine, it

was not available to anyone else. Much of the time was spent setting up tapes and card decks, loading the assembler and translating the program. If there were no errors, the loader was brought in from tape and the user's program loaded into memory. When the program did not work properly, the programmer spent time thinking about a problem while the machine sat idle. Changing the program caused the process to be repeated from the assembler stage; these iterations consumed valuable time. Finally, when a programmer was done, the area would have to be readied for the next user.

Even with production jobs, there was a period of many minutes when the machine sat idle, while the operator changed tapes, and set up switches to prepare for the next job. The early supervisors aimed at reducing the time wasted in this way.

To achieve these goals, jobs were collected into batches. Cards would be read onto a tape (an operation that could be done off line), and the tape would provide input to the computer system. Tapes were much faster than card readers and therefore better able to keep the processor busy. The system supervisor read a job off the tape, processed it, and wrote the output to another tape for off line printing. This process was repeated for each job on the tape. Operator intervention was only occasionally required to mount tapes as instructed by the supervisor. The step up between jobs was kept low, as most of the information was on the system input tape.

These supervisors performed other functions besides job sequencing. Resource usage was monitored, providing accounting statistics. A command language provided for several sequential job steps, such as assembly, loading and execution. Operator communications routines provided the operator with a more uniform interface with user programs.

The increased programmer and machine productivity afforded by supervisors, while representing a great step forward, was still far from optimum. First, the early supervisors had no protection from user programs. A user program could overwrite the supervisor, which caused time to be wasted in restoring the supervisor and the other programs in the batch stream. By 1957 hardware was being designed to prevent such catastrophes.

Second, the tape drives were still slow relative to the speed of the processor, forcing the processor to wait for the tape operations to complete before continuing with its computations. This meant that a significant portion of the processor's capabilities were being wasted. The interrupting I/O channel, which appeared around 1957, provided a means to reduce the wasted time. Interrupts allowed the processor to send a command to a peripheral device, then return immediately to its computation. The I/O would proceed without any need for intervention by the processor until it was completed; at that time the peripheral would interrupt the processor. If programs were carefully written, the time formerly spent waiting for the I/O could be used for computations. Only some of the time was retrievable however, since most programs needed the input data before it became be available. This time was a costly resource, providing an incentive to look for other ways to overlap the use of the cpu.

This brings the history of computer software to the beginning of the rapid rise of operating systems as we know them today.

3. Multiprogramming and Timesharing Systems

3.1. Introduction

This chapter traces the development of multiprogramming concepts from their infancy in the late 1950's, through their age of rapid growth in the early and middle 1960's, until research interest in them dwindled in the late 1960's.

During this decade of research, multiprogramming systems developed from simple user-controlled systems to automatically controlled multi-tasking systems. Major development topics were methods of automated job scheduling and resource allocation. Many systems were developed during this time period, representing different stages of growth and implementation methods. Several systems, representing major research efforts of long-range influence, were developed, among these was CTSS from MIT which will be discussed in further detail.

3.2. Definition of Terms

The meaning of some terms have changed as computer science has evolved. To minimize confusion the meaning, as used in this chapter, will be given.

Multiprogramming refers to the sharing of the cpu among several, possibly independent, programs. The term, by itself, does not indicate the method in which the sharing is implemented.

Timesharing refers to multiprogramming that is implemented so that each user receives a regular interval of time in which to do some work. This is somewhat different from the more contemporary interpretation that implies simultaneous user interaction.

An interactive system is one in which multiple users can access the system simultaneously, via consoles, and it appears to each user that there are no other users.

Multiprocessor refers to a system that contains more than one cpu, and those cpus are so tightly coupled as to be transparent to the user.

3.3. Early Influences and Motivations

This phase in operating systems' development was made possible by hardware advances in the middle and late 1950s. Memory, while still a scarce resource, was available in amounts that made a resident supervisor to control system components feasible. Magnetic drums had been available for some time and magnetic disks were becoming available, to provide relatively low cost, high speed storage to augment the limited central memory. Peripherals had acquired some rudimentary intelligence, so that they were capable of signaling the cpu when they required attention, through the use of interrupts. Removing the burden of I/O handling from the cpu opened the door to multiprogramming and further developments.

The motivation for multiprogramming was primarily economic. The computers available were very expensive and the cpu was spending much of its time doing nothing. This lack of utilization was due in large part to the disparity in the speeds of the cpu and the peripherals. Even when programmers attempted to overlap I/O and computation in their programs only small improvements were achieved. The natural conclusion was that the cpu time not used by one program could be used by another. It was with this background that research into multiprogramming systems started.

3.4. Historical Development

3.4.1. Early Concepts

The earliest multiprogramming systems were essentially manual. The programs to be run together were assembled onto a magnetic tape with explicit sequencing information embedded in the programs. They were treated by the supervisor as one large job [Crit63]. Unfortunately these programs were then bonded so tightly together that all had to be reassembled if a change was made in any one of them. It quickly became clear that the efficiences gained were canceled by the increased complexity, and time spent recompiling. The complexity problem was made more critical by the increased use of high level languages. High level languages removed the machine characteristics from the realm of the programmer. However, to synchronize programs to efficiently use the cpu, the programmer had to be very familiar with the particular machine's features. These two trends were diametrically opposed to each other. The value of high level languages was already fairly well accepted by this time, so some other method of multiprogramming had to be devised to remove the responsibility from individual programmers.

3.4.2. Early Supervisors

There were many parallel developments that influenced the ways in which multiprogramming was implemented. In 1956 and 1957 the UNIVAC LARC and IBM Stretch computers were under development. Both of these systems were highly parallel machines, using interrupts to coordinate the operation of multiple cpus. These systems prompted people designing multiprogramming systems to consider using hardware techniques and interrupts to control the use of a single cpu by multiple users.

One of the first attempts at machine controlled multiprogramming was the TX-2 computer at Lincoln Labs [Clar57]. This system attempted to implement multiprogramming almost entirely in hardware. The machine was equipped with 32 instruction counters: one for each possible program. There was a hardware scheduler that determined which of the 32 programs, including systems programs, would be executed next. There was no method of protecting the programs in memory from each other. This system did not have a long lasting effect on multiprogrammed systems development, but is interesting in its approach and its position as a forerunner in the field.

The next publicized attempt at multiprogramming was the GAMMA-60 system from France [Drey57]. This system is significant in that it was the first commercially available system with vendor supplied hardware and software for multiprocessing and multiprogramming. The software supervisor was primitive, but it represented vendor recognition of the need for software. The GAMMA-60 did not achieve widespread acceptance or recognition.

In 1959, Strachey presented his paper "Time-Sharing in Large Fast Computers" at the IFIP Congress [Stra59]. This was a milestone in that it was the first major paper devoted solely to the development of techniques to implement timesharing. The ideas proposed influenced future systems, so a closer look at his ideas is in order.

Strachey recognized that there were limitations on how much intelligence it was feasible to put into hardware. The logical complexity of a system had direct bearing on its cost, and the speed advantage to be gained was not that critical. Inasmuch as the peripherals couldn't be completely autonomous, the cpu must be involved in their operation. Since the time required was so small, it would be practical to borrow the cpu from the program using it. The peripheral would interrupt

when it needed attention and the cpu would temporarily stop processing the user program to execute a special sequence of instructions to handle the peripheral's request.

The many types of peripherals, with widely differing speeds, necessitated different classes of interrupts to indicate the nature of the request. If a high priority interrupt was received, while already processing an interrupt, the lower priority action would be suspended. All interrupt processing was based on priorities and first-come first-served (FCFS) within priority classes. The interrupting program would be responsible for saving and restoring any common registers that it used. Some of these ideas had been considered in the design of the Stretch and LARC computers, but had not been discussed in one place before this paper.

Strachey proceeded to develop some new concepts. His concerns were motivated by the multifold speed increases coming with the new microsecond machines. He feared that the increased capacity would be wasted without new techniques to take advantage of it. To this end, he proposed a timesharing supervisor called the Director.

The purpose of the Director was to control the utilization of the cpu and its peripherals, while supporting on-line debugging and on-line maintenance facilities. The Director and its privileged instructions could be stored in a high speed, inexpensive read-only memory (ROM) to protect it from the users. The Director protected programs from potentially destructive interaction with other programs by limit registers. These would confine users to their own area in memory. When a program was selected to run, the Director would allocate memory for it and the memory bounds be placed in the limit registers. All memory references would be checked against the registers, in parallel with the memory operation. The Director would be notified of any violations.

The system would allow simultaneous operation of up to four job types. There could be a low priority, long running "base load" program, that was run when no other jobs were ready to run. The second job type was a queue of short to medium length jobs, run in first-come, first-serve (FCFS) order. These programs would have been assembled in an earlier compilation run. The third job type would be a program being debugged on-line at a special console. On-line debugging was affordable because there were other jobs to keep the processor busy while the programmer thought. A maintenance program to help service the peripheral equipment was the last job type. The Director controlled the input of programs from paper tape for later storage on magnetic tape, and low speed output to terminals, in addition to the four job types.

Strachey's ideas had varying impact on system development. The limit registers and reserved instructions became widely adopted, as the need for some form of memory protection had already been recognized. The concept of having the supervisory program in ROM was not widely accepted, with the notable exception of the Manchester University Atlas System in 1961. However, his idea was a precursor of the concept of having a separate address space for the software supervisor. Later researchers found different ways to implement it, as will be seen in subsequent chapters.

The timesharing aspects of his Director were too restrictive to be generally copied, but his work did provide a basis for further research into timesharing methods. His idea of a base load program was the forerunner of foreground/background systems. More sophisticated ways of determining job mix and job scheduling were needed to further improve system utilization.

Strachey's proposed system typified the prevalent philosophy that machine utilization was more important than programmer utilization. The major objective of

his system was to make best use of the machine, with no regard for programmer efficiency or turn around times. Jobs were run in FCFS order within program group. There was only limited on-line access planned (one special console), indicating that programmer time was not as critical as machine time.

The same year Strachey proposed a software based multiprogramming system because of its cost and flexibility, Honeywell was announcing its H800. The H800 could support up to 8 simultaneous programs, with control of the multiprogramming and scheduling in hardware, "without the use of cumbersome supervisory routines" [Lour59]. This system did represent an improvement over Strachey's system: it provided elementary round robin scheduling, that could be overridden by a user program. Like Strachey's proposed system, the H800's design was too limited in scope, partly due the inflexibility of hardware implemented multiprogramming, to be generally adopted.

1960 was a quiet year with no major announcements. Several models for hierarchical control of computer resources were proposed [Baue60 and Ryck60]. The number of high level languages was increasing greatly, hardware was changing and hardware failures were common. The environment that an operating system controlled was increasingly dynamic. Static operating system designs were phased out as the concepts of modular, flexible, layered operating systems were developed. The objective was to make addition of new languages processors easier, to remove device dependencies from the user, and to control the equipment effectively.

3.4.3. The Growth Years - 1961 to 1964

The period between 1961 and 1964 was one of tremendous interest and growth in multiprogramming supervisors.

The Manchester University Atlas system represented a great step forward. The principal goal of the operating system was to maintain a balance between the peripherals and the processing unit. The slow speed peripherals were fed into wells in the central memory and magnetic drum. These wells were adaptations of the buffers used in earlier generations of supervisors. If the program executing filled the output well, it was suspended and a more compute bound job was selected for execution. Before a job could be run, its inputs had to be available in the input well. The objective was to overlap the input and output of other jobs with the computation of the current job. The scheduling methods did not represent any advance.

The feature that made Atlas significant was its concept of one-level store [Kilb61]. This represented the first attempt to make the levels of storage and their differences in speed transparent to the user. As such, Atlas represents the first virtual memory system and these aspects will be described in the next chapter.

The Compatible Time Sharing System (CTSS), from MIT, represented the next major step forward. This system's aim was to balance computer and programmer efficiency through interactive timesharing, a radical change in approach. This project served as the starting point for most later interactive systems, and as such deserves a close examination.

A prototype of CTSS was announced in 1962 which supported four users in a foreground/background environment [Corb62]. Control of the foreground users was the main effort and interest in this system. The background job function was that of the base load program in Strachey's design.

The service objectives of the system were to provide immediate response to user keystrokes, and to respond to commands in time linearly proportional to the number of users on the system. The software was to handle user accounting, user I/O requests, and to provide a range of utility programs to simplify programming. These utilities would include compilers, editors, post-mortem dump routines and feedback methods for use during program runs. Ease of use and programmer efficiency were key goals of this operating system.

Hardware was needed to help achieve the software goals. In specific, memory protection, dynamic memory relocation, an interval clock, and special I/O trap instructions were required. Many of these features were not fully used by the prototype operating system. Only one user, in addition to the supervisory programs, would be allowed in memory at a time. The memory protection features were used only to protect the supervisor from the user. With only one user the need for the dynamic relocation capability was limited to interaction of the user program with library subroutines. The I/O trap instructions were a means to force the user to use the supervisor for I/O. The clock was used in the scheduling process.

CTSS's scheduling algorithm was one of its major contributions. It used a multi-level priority, time-sliced method to allocate the cpu. As each job entered the system, it was assigned to a priority queue based on program size. The formula used in this determination was

$$q = \lfloor \log_2 (wp/wq + 1) \rfloor$$

where wp is the number of words in the program and wq is the number of words that can be transferred between the memory and drum in one quantum.

The queue numbers were prioritized in reverse number order, with low numbers being higher priority. To determine which job to run next, the scheduler picked the first job from the lowest occupied queue and set the clock interval for

2^q quanta, where q is the queue's level number. If the job was not completed at the end of that time, it was placed at the end of the next higher numbered queue. The process repeated with the scan for the lowest occupied queue. When a new job entered the system, another scan was initiated. If the new job had a lower queue level than the currently executing job, and that job had already used as much time as the new job would receive, the current job was suspended. If a job changed its memory size at any time, its new queue number and interval was determined and applied retroactively.

The algorithm maintained an acceptable level of service even when the system reached saturation. It ensured that a program would receive at least as much time in service as it took to swap it in. That way the computational efficiency never fell below one-half. It also allowed a computer facility to determine the maximum number of users it could support, given a desired response time limit. Other advantages of this algorithm were that program size and cpu requirements were measured automatically by the system, and not estimated by the user, as was standard at the time. The algorithm also responded to changes in program characteristics.

CTSS grew from its four user prototype to a 110 console system by 1965 [Cris65]. The system was still based on the foreground/background concept, but with an obvious growth in the number of supportable foreground users. Hardware advances helped make this growth possible. The availability of large capacity, high speed disks made swapping faster, allowing more concurrent users. The increased memory sizes made it feasible to keep more than one user in memory.

On the 1965 system, there were two banks of memory, one for the operating system and one for the users. Memory protection was implemented via two protection registers, whose contents were used as limits on every user memory access.

There was a relocation register that acted as base address register, modifying every memory access to simplify dynamic program relocation. This in turn gave the system flexibility in swapping. All memory allocations were based on blocks of contiguous memory locations. The supervisor was responsible for finding a block of memory large enough to hold the program, and swap out programs as necessary to make space.

Other features that were added during the development of CTSS were the ability to run jobs in foreground mode while detached from the user terminal, passwords for accessing the system and private files, an interconsole message capability, public disk files and an on-line user manual.

The CTSS developers had hoped to eliminate the foreground/background environment, and with it the differentiation between on-line and batch jobs. They never implemented this change, as many of the early pioneers on this project had moved on to other projects.

CTSS is the most remembered system of that time period, but there were others that added to the collective knowledge. Burroughs had developed AOSP, An Operating System Program, that acknowledged the known timesharing concepts. The most interesting point about this system was that it performed confidence checks on itself when the cpu was idle. Considering the questionable reliability of computer systems at that time, it seemed an intelligent use of excess processor time.

Another system that came not long after CTSS was TSS from Software Development Corp [Schw64]. This was a batch system, unlike CTSS. TSS had an interesting scheduling algorithm which was regulated by the number of users on the system. It was based on the concept of a minimum quantum and a cycle time.

The minimum quantum was established by the installation manager and was used when the system was under maximum load. The cycle time was time available for one complete pass through all the active jobs on the system. When not under maximum load, the quantum used was the cycle time divided by the current number of active jobs. If a job did not use its entire quantum, the excess time was distributed among the other users.

Jobs were assigned priorities as they became active, based on the program size and the type of I/O devices it required. The memory was partitioned into four memory banks. Bank 1 held the operating system, Banks 2 and 3 were used for medium and low priority jobs and Bank 4 for high priority jobs. The system favored small jobs, as large jobs may have required all three user memory banks. Inter-user protection was provided only a bank basis.

To facilitate swapping jobs in and out of memory the supervisor had a space allocation algorithm for the drum. Object programs were kept as contiguous storage locations, so they could be quickly and easily be transferred into memory when needed. Each time a new job entered the system, all empty spaces left by completed programs were collected together and an inventory of the space was made before assigning the new job a location.

3.4.4. The Close of the Era - 1965 to 1968

By 1965 the principles of multiprogramming were well established. The need for protection, relocation, reentrancy, scheduling and storage allocation methods were recognized and various solutions had been implemented. The following years were years of refinement of the techniques proposed and investigation into new areas.

The only implementation project of significant interest during this period was developed in 1968. This was THE Multiprogramming System by Edsger Dijkstra in the Netherlands [Dijk]. The significance of this system was the approach taken towards programming in general. The entire system was viewed as a collection of parallel sequential processes.

These processes could cooperate through explicit mutual synchronization statements. Each process works at its own speed, and if necessary waits for another process to give it information before it continues. To implement this concept Dijkstra promoted the use of semaphores, and synchronization primitives to act upon them. The semaphores are integer variables allocated within the universe of the processes. The primitives increase and decrease the value of the semaphore via the primitives, which are uninterruptable actions. This allowed the system to define critical sections, which could be used to examine and modify state variables, without fear of the states being changed after the check by another process.

Breaking a large program into a group of smaller cooperating processes could be used as an aid in the resource allocation problem. The memory requirements for the individual pieces were fairly small, and more easily satisfied than for one large program. Also there was no special hardware required beyond that which was used to implement multiprogramming. These concepts have been used on all types of machines.

The concepts of mutual exclusion by use of semaphores has been widely adopted, as has the idea of cooperating processes, as shall be seen in subsequent chapters. The widespread acceptance of the concepts presented make THE a significant contribution to operating systems research.

It is not fair to assume that no other work was done on multiprogramming systems after 1965. The systems discussed in this chapter were primarily research projects, that contributed greatly to the knowledge base, but were not commercially successful. The honor of commercial successfulness belongs to IBM with the System 360 and OS/360 and its many offspring [Meal66]. These software products had few new technological advances, but because of IBM's stature in the computer industry, helped the ideas of multiprogramming become universally accepted.

3.5. Contributions of the Era - 1957 to 1968

This period in operating systems design contributed greatly to the collective knowledge of computer science. The techniques developed served as the basis for further research, that eventually lead to even better methods. Many concepts formulated during this period are still used today, as will be shown below.

Hardware and software contributions are sometimes difficult to differentiate. Hardware refinements were often prompted by new software demands. These same refinements made many of the software concepts feasible and lead in turn to new developments.

3.5.1. Hardware Refinements

Memory sizes and speeds increased over the period, making new ideas practical. Memory protection, in the form of limit registers and other techniques, became commonly available. Increases in the transfer rates and capacities of disks made them usable for swapping.

Interrupt capabilities were enhanced to take the burden away from the software. There were multi-level priority interrupts, that identified the nature of the device and its request. Certain registers, such as the program counter, were

automatically stored when an interrupt occurred. On many machines there was a special supervisor mode for executing certain privileged instructions. The system switched into this mode automatically when an interrupt occurred. This mode provided a separate address space for the supervisory program. These advances made the interrupt processing faster and more secure, allowing the sophistication of the operating system to grow without degrading performance.

Sophisticated methods of relocation made swapping user jobs easier, as the programs now could be placed in any available memory locations. This also made it easier to use system library routines.

3.5.2. Software Refinements

There were many general concepts developed in this period. The idea of an operating system providing a wide range of utilities grew up out of simple I/O supervisors. People started to expect an operating system to provide many services, to make using the computer less of a chore. The attitude towards computing went from computing for its own sake to computing as a workable method of problem solving. The areas of greatest contribution were in scheduling and storage allocation techniques.

3.5.2.1. Scheduling

When monitors first came out, jobs were serviced in first-come first-served (FCFS) order. Priorities, if required, were established manually by placing critical jobs first in the deck. Early multiprogramming systems approached the scheduling problem in much the same way. It was soon discovered that this was not necessarily the most efficient way in which to use the computer.

Many different techniques were developed at approximately the same time, many of which are still in use. Priority schemes include shortest job first (SJF), highest priority first (HPF), round robin and feedback queues. The first two methods are self explanatory. Round robin scheduling implies that each job is serviced in a rotating sequence. In a feedback queue, priority is determined by the amount of service received: the entering job is assigned highest priority and as it is serviced, its priority decreases. This ensures that each entering job receives at least as much service as the least serviced job in the system. There are other categorizations of scheduling methods as well.

There are non-preemptive scheduling techniques in which the current job continues until the end of its allowed time, even if a higher priority job has come into the system in that time. There are preemptive techniques in which the current job is suspended upon arrival of a higher priority job. The definition of higher priority is dependent on what priority scheme (SJF, HPF, etc.) is used. In preemptive systems there is a choice of how to handle the job being preempted. Some systems throw out the results of the preempted program, forcing it to restart later, or flushing it from the system totally. This technique is rarely used today, but in the early days hardware limitations sometimes forced this method to be used. The preferred method of handling preemption is to save the preempted job's environment, and restore it when that job is scheduled to run again.

The availability of information for scheduling is another issue. If a system uses SJF for example, how does it determine the length of a job? This information is needed to determine what job to run next. That information can be provided by the user, or it can be determined automatically by the system, with additional help from the compilers. The problem with users specifying job requirements, such as cpu time and peripheral resources required, is that they generally don't have the

information. These requirements may vary during the execution of a program, and the system would always be making its decisions based on worse case situations. Automatic techniques allow the system to respond to changes in program requirements, but they require more software sophistication. However, because there is no a priori knowledge of a program's behavior, there are no checks on a runaway program with automatic techniques.

Scheduling techniques can also be classified as static or dynamic. Static implies that the scheduling method does not vary with different loads or situations. An example of a static technique would be a round robin scheduler with a fixed length quantum. Regardless of the system load, the service relationship to each user remains the same. There are many dynamic scheduling strategies: some involve dynamic quantum determination, based on what happened in the previous quantum. Others depend on the amount of service a user has already received, or how much service a user has received compared to other users.

Some scheduling methods are designed to order jobs to minimize swapping, maximize memory utilization, or favor a particular class of job. Some methods are best suited for light to medium loads, such as pure round robin. Others are well adapted to gracefully degrade under heavy loads, such as the CTSS multi-level priority scheduler.

Almost all operating system schedulers use a combination of these techniques depending upon the anticipated user environment. This way the heart of the operating system can be adapted to handle user needs.

3.5.2.2. Resource Allocation

The primary concern of resource allocation schemes is the efficient use of memory. In a multiprogramming system memory is a critical resource and was

especially so in the days of small memories. A number of methods were developed in this period to handle this problem.

Before relocation hardware was readily available, all program addresses were bound within the program, thus locking programs into particular memory locations. This made sharing programs difficult and multiprogramming impractical. This could be partially overcome either by assigning users to particular blocks of memory or allow only one program in memory at a time and use swapping to affect multiprogramming. This was not a very efficient use of system resources.

When memory relocation hardware became standard, a number of things happened. The binding of symbolic addresses was moved from programming time to either load time or to the time of reference. Supervisory programs had more flexibility in allocating storage since programs were address independent. With this capability, it was feasible to run multiple programs in memory simultaneously, since they could be located anywhere without ill effect. If the address binding was dynamic, this afforded even more flexibility in that the programs could be moved about in memory as needed. If the address binding was at load time, the program had to use those memory locations for the duration of its run.

The hardware influenced how memory allocation was handled in another way. Almost all the systems mentioned allocated memory as contiguous blocks. This was due to the hardware protection devices. Most machines had only a pair of limit registers for a protection mechanism. Unless a program was allocated a contiguous block of memory, there was no way to implement memory protection. The need for memory protection was well established, and not to be abandoned in favor of more efficient memory allocation. Within this constraint of contiguous memory allocation, there are many ways to handle the details.

The simplest method is static partitioning, such as that used in IBM's OS/360. Each partition is a fixed size, and in a fixed location, as chosen by the installation manager. When a job is to be swapped into memory, a partition of sufficient size is found. Each partition has associated with it in-use indicators and only one job can reside in a partition. Unfortunately this technique wastes space as jobs are rarely the exact partition size, and in fact may be significantly smaller than the partition. It also presumes that the job characteristics of a user group are known in advance and are static.

Another method is dynamic partitioning, in which memory is divided into partitions each of which represents a user program. Each partition has only one job, and each has a status indicator associated with it. A list of available partitions is kept by the operating system, so that when a job enters the system a place can be found for it.

There are several ways to chose which partition to give a job. One method is the first fit algorithm, in which the list of free space is kept in address order. When a job comes in, the list is scanned, and the first available partition that will fit is used. The algorithm tends to load the low memory addresses with small jobs, leaving larger blocks in high memory for large jobs. Another technique is the best fit algorithm, in which the free list is kept in order by size. The first partition in the list that fits the job is chosen. This guarantees that the smallest possible partition is selected, tending to leave large unbroken areas for large jobs. The advantages of a dynamic approach to memory allocation are that it assumes no prior knowledge of the job characteristics, and it responds to changes in environment.

Unfortunately both these algorithms tend to fragment memory. For the free memory locations to be useful, they need to be collected together. This process is known as compaction or shuffling. Jobs present in the system are moved, as

necessary, to eliminate gaps in allocated memory. The free space is then in one block and considered one partition. Compaction, if used often, can severely impact the system performance as it is a time consuming process.

The obvious problem was that the contiguous memory requirement impeded efficient use of memory. Other methods were developed to neutralize this problem, called segmentation and paging. These methods will be discussed in the virtual memory chapter. Many of the methods described are still in use, especially on minicomputers that do not have the hardware needed for virtual memory.

4. Virtual Memory Systems

4.1. Introduction

This chapter presents the development of virtual memory operating system concepts. Virtual memory systems began in the early 1960's with the Atlas System. They developed in complexity and capability through the mid to late 1960's; by the early 1970's they were an accepted design technique and research interest in them dwindled.

The concepts developed during this period centered on more effective ways to handle storage allocation, protection, and sharing of common information within a multiprogramming system. Several systems represent the major developments of the first half of the period, notably the Manchester University Atlas system and MIT's Multics system. The second half of the period was devoted to studying the techniques available to manage virtual memory, and developing models to determine which were most efficient. The major developments of each period will be discussed in further detail.

4.2. Definition of Terms

To discuss any concept, a common vocabulary must be established; some general terms are defined here to minimize confusion.

Virtual memory is the set of storage locations in physical memory and secondary storage that are referred to by a common address space. To the programmer all addresses appear to be in physical memory. A virtual memory system is one that presents this view of memory to a programmer via hardware and software techniques.

Segmentation is a method of implementing virtual memory in which a program's address space is regarded as a collection of named segments. A segment is logically a collection of information important enough to be given a name; physically a segment consists of a contiguous set of addressable locations.

Paging is a method of implementing virtual memory in which a program's address space is divided automatically into fixed length blocks of contiguous addresses. These fixed length blocks are called pages, and usually have a size between 256 and 4096 words. Page frames are pages in main memory. Paging and segmentation can be used separately or in combination.

A process is a collection of program procedures and data operating together to perform some task.

4.3. Early Influences and Motivations

This phase of operating systems development partially overlapped the development of multiprogramming systems. The factors that influenced development of the earliest virtual memory system, Atlas, were much the same as those which motivated development of multiprogramming systems. These factors were the economic necessity of efficient machine utilization, coupled with a desire to make that process transparent to the user.

Small memories, and a need to share the cpu among several programs to increase utilization, created an awareness of the need for sharing common procedures. This was not easily done using the common memory allocation scheme of assigning contiguous memory addresses to a program. That technique also presented problems in memory allocation for a multiprogrammed system. Entire programs had to be transferred to and from secondary storage if there was not enough memory for all programs to run simultaneously. If there was enough memory, but not in

contiguous locations, the free space had to be redistributed by moving programs in memory; even this technique would not work if dynamic relocation of programs was not supported.

In the early 1960's, when Atlas was developed, main memories were very small; programs could easily be larger than the available memory space. Some systems overcame this problem by requiring that the user structure his programs so that unrelated sections could write over each other. This process is referred to as overlaying. This put the responsibility for storage management within a program on the programmers; any changes in storage size necessitated a change in the overlay structure to keep machine utilization high. The increased use of high level languages made this process more difficult, in that the user was removed from many of the machine details by the language, including factors such as memory usage. The purpose of virtual memory on the Atlas was to make memory overlaying the responsibility of the operating system and transparent to the user. This would provide a level of abstraction between the details of the computer's storage levels and the users, allowing a degree of configuration and device independence previously not available.

4.4. Historical Development

4.4.1. Early Concepts

If one considers virtual memory to be the extension of a program's address space beyond the size of the physical memory available, then manual overlays represent the first virtual memory systems. Memory locations were mapped by the user to more than one program address. This was achieved by breaking the program into logical segments and having the segments brought into memory only when they were needed. Segments that were unrelated could time-share the same

memory locations.

This technique was capable of giving the illusion of almost unlimited memory, but was difficult and clumsy to use. One major problem was that the overlays were created to run in a particular minimum memory size: if the available memory was decreased, through hardware changes or an increase in the operating system size, the overlay structure had to be altered to run in the reduced space. The man and machine time involved to do this reduced the overall efficiency. Some automatic way of handling program segmentation and memory allocation was needed.

4.4.2. Early Systems

The Manchester University Atlas system in 1961 marked the start of virtual memory systems as we now know them [Howa63, Kilb61, Kilb62]. The technique was known then as one-level storage ; the term virtual memory came much later. The original purpose of the operating system, and its one-level store, was to minimize system idle time. To achieve that goal the operating system had a scheduler that tried to always to have at least 2 jobs on a list of active jobs. The small memory size did not always allow the active jobs to be entirely resident, and swapping entire jobs involved too much overhead.

The method used to solve the problem was to break all random access storage, including the drum, into pages of 512 words. An instruction address referred to the combined core and drum storage of the machine. Associated with each of the pages in memory was a Page Address Register (PAR) that indicated which logical block of information occupied that page frame. When a memory reference was made, the hardware attempted to find a match between the instruction address and an address in one of the PARs. This search, called an equivalence

check, was done through all 32 PARs simultaneously to minimize the associated overhead.

To reduce the overhead even further, the PAR mechanism was effectively bypassed on instruction fetches. It was assumed that most instruction accesses would be within the same page, so a request would be made using the word index of the instruction address and the page number of the last instruction. In parallel with the request, an equivalence check would be made, plus an additional check to be sure that the instruction was actually within the same page as the last instruction. The memory request would be canceled if either check indicated the wrong word was being accessed, and a new request for the correct word would be made. This concept of nearby accesses is known as program locality.

During an equivalence check, if the hardware did not find the address requested in one of the PARs, a non-equivalence interrupt (page fault) was generated. When that happened a drum transfer from the block indicated in the referenced address was initiated. The transfer copied the page from the drum to an empty page in main memory. An empty page was always available; if the last empty page was used, an occupied page was written out to the drum.

The choice of which occupied page to write to drum and was handled by a special monitoring routine. Each page in memory had an in use bit associated with it, and every 1024 instructions the bits were copied into a list. When a page was to be chosen for removal, the monitor calculated the length of time since each block was last used (t) and the length of the last period of inactivity for each block (T). The last period of inactivity was the length of time, before the last access, that the page was not referenced; when a page was transferred into memory, T was set to zero. The page to be swapped out was chosen by the formula

$$t > T + 1$$
or $t \neq 0$ and $(T-t)max$
or $Tmax$ and all $t = 0$

This means that the page was chosen, if possible, from pages whose time since last reference was greater than the last period of inactivity. If there were no pages that fit that criterion, the next choice would be from those pages not in use and with a maximum difference between last use and latest period of inactivity. The last choice was a page from those that were currently in use with the longest last period of inactivity. This algorithm guaranteed that if a page was chosen for swap out and was then referenced immediately, the page would not be chosen for swap out on the next cycle, since its latest period of inactivity would be zero. This technique was said to be as good as an experienced programmer in choosing which page to swap out, with the additional benefit that it would take into account the dynamic state of the machine, whereas a programmer would not have access to that information.

This addressing mechanism had many advantages. First, the users were shielded from the intricacies of the multiple storage levels of the system, and their associated speed differences. Second, only those pages being used by a program needed to be in memory; this actually allowed more flexibility in the number of jobs that could be in memory simultaneously. This flexibility, in turn, increased the choices available to the scheduler for jobs to run, so that it could do a better job of balancing the input, output and computation loads. The memory allocation routine was much simpler because programs were no longer address dependent, and could be moved to any convenient location in memory.

With this additional freedom, and the ability to run multiple simultaneous jobs came additional responsibility. User programs could not be allowed to interfere

with each other or with the operating system. Each page had associated with it a lock-out bit; when set, this bit prevented access to the page by the cpu. The supervisor kept a list of all pages belonging to a program, and all pages not on the list were locked out, thus preventing users from accessing other users' pages. The supervisor protected its main memory pages in the same way, although most of the supervisor was in read-only memory (ROM) or in subsidiary storage which could not be accessed by the user.

Atlas represented a major step forward, both in its handling of resource allocation and multiprogramming. It was the most sophisticated system of its time, and it was several years before any significant advances were made in virtual memory systems. Several systems built upon Atlas's multiprogramming ideas, as discussed in the previous chapter.

Burroughs developed the B5000 system in that same year (1961) and it too implemented a form of virtual memory [Lone61]. The B5000 was attempting to solve many of the same problems that the Atlas tackled.

The B5000 provided automatic program segmentation, allowing program size to be independent of physical memory size. The system could then handle global resource allocation and scheduling more efficiently than individual programmers. The compilers divided a program into variable length segments. Each program had its own set of program descriptors, each giving the drum location of a segment, a base address in memory and in "in core" indicator. If a reference was made to a segment not in core, it would automatically be brought in by the Master Control Program. Programs were fully relocatable since they only referenced entries in the Program Reference Table. This gave the Master Control Program more flexibility in memory allocation, and increased the number of jobs simultaneously in core.

Until about 1965 there was little visible activity in virtual memory research. The successor of CTSS at MIT heralded the beginning of the key years in virtual memory research.

4.4.3. The Growth Years - 1965 to 1970

Multics (1965) was the first major virtual memory system after Atlas [Bens72, Dale68, Orga72, Ossa65, Vyss65]. Multics' goal was to become a commercially viable operating system; it achieved this goal, and it is still used today. This goal colored the functional requirements, design, and implementation of the system. To be a commercially feasible timesharing system, it had to provide continuous service. This meant that the software had to be flexible enough to handle dynamic changes in the hardware configuration. In addition, the software itself had to be reliable.

These requirements led to a modular operating system design. The system had to allow easy addition of language processors and utility programs. It also had to be constructed so that key operating system functions, such as scheduling, memory allocation, and I/O handling could be easily modified, to keep up with anticipated changes in machine and user environments. As a research project, the designers wanted to be able to test new techniques and to take advantage of other researchers' developments, without disrupting the large user community. A modular operating system design would allow sections of code to be replaced, without affecting the operation of the rest of the system. The modular design was made easier to implement by the use of PL/I.

Reliability implied that the operating system had to be able to recover from hardware malfunctions. This required a flexible approach to device handling, so devices could be excluded or included in the configuration as their status dictated. Part of the flexibility needed was device independence, so that users would not be

affected by device changes. Reliability also implied protection, from other users changing data, and reading sensitive data. Seemingly at odds with the protection goal, Multics had to allow controlled sharing of data and programs among consenting users. There are some of the factors that led to the design of Multics.

As mentioned earlier, it is difficult for a multiprogrammed system to allocate memory if programs must be stored in contiguous physical locations. Physical contiguity makes sharing common code and data among many programs a formidable task. Either multiple copies of shared programs must be made, which wastes valuable space, and leads to problems of updating the copies, or an entire program or data area must be made accessible to the sharing programs, which is dangerous. This "all or nothing" form of protection was not acceptable any longer, given an increasingly sophisticated user population.

Multics solved these problems by using segmentation. A segment is a user-named, user-defined portion of a program or data area. It has no fixed size, and is the smallest sharable and protectable element. Segments in Multics are quite different from the segments used in the earlier B5000. Multics segments are an organizational and naming technique, rather than a memory allocation tool, as they were on the B5000. Also segmentation is user controlled on Multics, not automatically controlled by the system.

Segments are used to implement files. Segments are associated with a process via the process's Known Segment Table (KST). When a process wants to reference a common routine, it uses the routine's symbolic name. The name is searched for in the KST; if it is found, the segment number stored in the KST is used to determine the physical address. If the segment name is not found, then the directory hierarchy of segments maintained by the system is searched. When the segment is found, an entry is made in the KST of the requesting process. This

mapping procedure takes place at run-time, so there is no need to link program pieces together before they are executed.

The KST entry for a segment points to the same physical location for all processes using a shared segment. This eliminates the problems relating to updating copies of shared files or programs, as each user is directly accessing the original entity.

Segmentation seems to solve the sharing problem. Hardware protection on each segment also solves the access problem, since each segment can be read, written, executed or totally isolated from other users. However, segmentation does not solve the memory allocation problem. Granted, segments will often be much smaller than entire programs and that would help the allocation difficulty somewhat. But segments can dynamically change their size, and this could cause severe problems. Since segment behavior is not predictable, preallocation of the maximum segment size would waste valuable space, especially if restricted to contiguous memory locations.

Multics solved the allocation problem by dividing the variable length segments into fixed length pages. A page is 1024 words and a segment can contain up to 2^{18} pages. Physical memory is divided into 1024 word page frames. A page can reside in any available page frame. This allows the operating system maximum flexibility regarding use of memory, while affording the user the convenience of named segments. The paging aspects of the system are transparent to the user.

To see how these two techniques (segmentation and paging) fit together, one must examine the translation of an instruction address into a physical address. An address consists of a segment number and a word number within that segment. Figure 1 shows the address translation process used to find the physical address

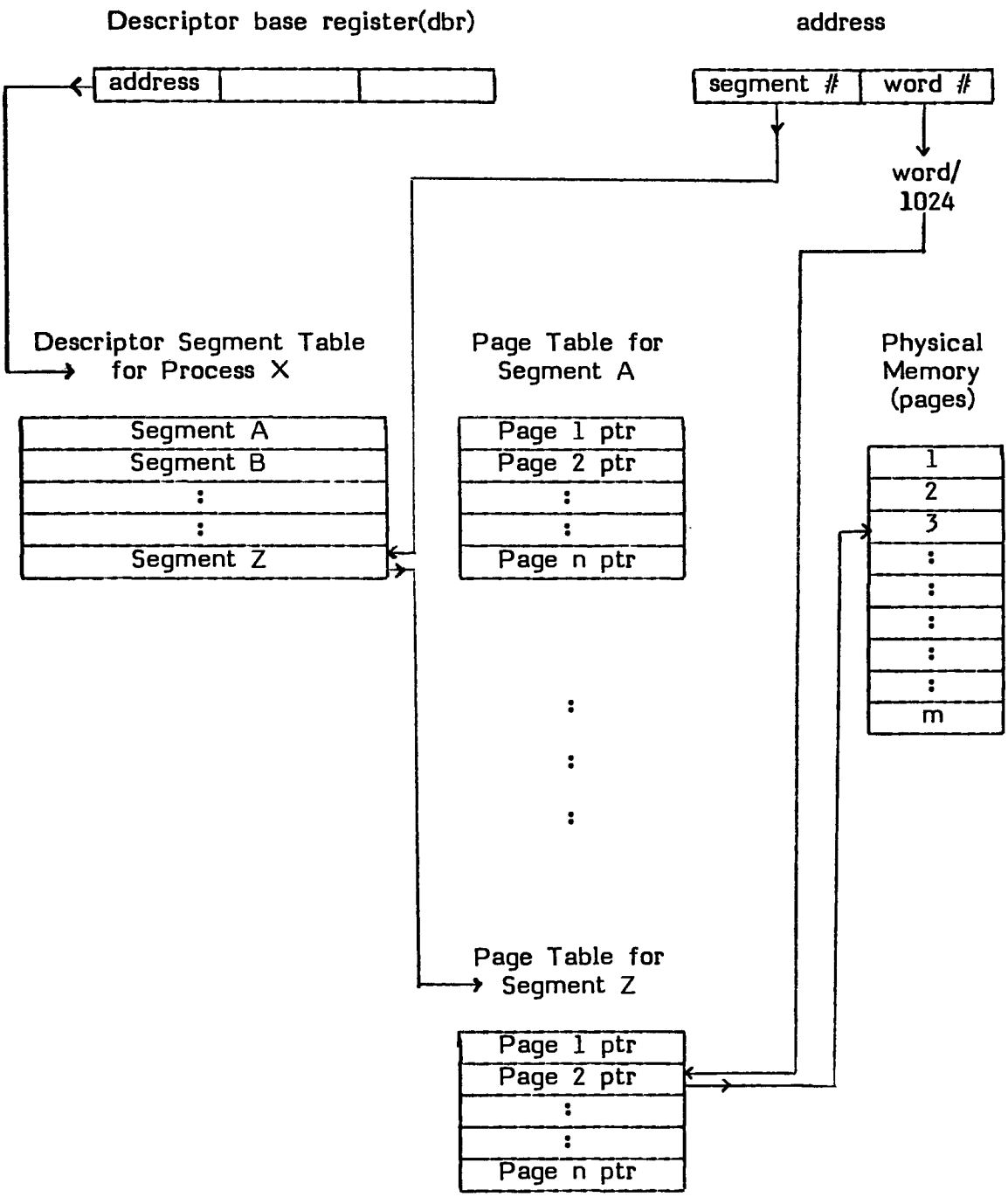


Figure 4-1
Multics Address Translation

from the segment number and word index. This addressing scheme makes sharing procedures and data very simple. The information to be shared occupies a segment that is made known to the requesting processes through the KSTs. The KST entry points to the same segment for all the sharing procedures and the descriptor segment table entries will be the same for each process. This way there is one copy of the shared information available to processes with the proper access rights. The supervisor is also a set of paged processes, and many of its routines are shared among the user processes in this way.

We have seen how pages and segments are addressed, and how segmentation facilitates sharing; the next step is to see how memory is actually allocated.

An area in memory is reserved for the Active Segment Table (AST), containing information on segments whose page tables are in memory. Another portion of memory, whose size is determined at initialization time, holds the page tables. The page tables are all the same size. The supervisor divides the page tables into two lists, a free list and a used list. When a segment not in core is needed, a page table is chosen from the free list and the segment is added to the AST. If there is no free page table, a segment and its page table are deactivated. The segment chosen is the one which had no pages in core for the longest time. There will always be at least one segment with no pages in core, since the number of active segments exceeds the number of pageable blocks of core. The segment tables and AST will be updated accordingly. If the segment is a shared segment, all corresponding segment descriptor tables will be updated.

When a page not in core is referenced, a page fault occurs. The core map is consulted to find a free page frame. This map groups page frames into a free list and a used list. A page frame is selected from the free list, and the requested page from storage is transferred into it. The page table and core map are then

updated. A pool of free page frames is maintained by selecting pages to be written out to secondary storage. The selection process uses the in use bit in the page table word to select the least recently used page for replacement.

In this scheme only referenced segments and pages are in memory. The least recently used segments and pages are selected for replacement when there is not enough memory. This minimizes the amount of paging traffic and thereby maintains an acceptable level of performance.

Many virtual memory concepts were developed during or influenced by the Multics project. It established a precedent for segmentation and paging, and the mechanisms to implement them were the bases of future work. Computer vendors were marketing virtual memory systems by 1966. IBM used the ideas of segmentation and paging in the design of the S/360 Model 67 in 1966. It was made to compete in the timesharing market against the General Electric 645, the first machine used in the Multics project. The 360/67 was the first model of the 360 series to have hardware support for virtual memory, and a new operating system, TSS/360, was developed for the system. A process could have 16 segments of up to 256 pages each. The page size was fixed at 4096 bytes. A high speed associative memory sped up the address translation process.

These early systems showed many ways that virtual memory could be implemented. The next few years produced studies of the effects of virtual memory on program performance, and methods to optimize overall system performance. Very little was written in that period about specific implementations.

By 1968 various groups were modeling page replacement policies, program behavior, and resource allocation strategies. Belady was the first to present a comprehensive comparison of the myriad replacement algorithms [Bela68]. He

grouped replacement algorithms into three major classes based on the type of knowledge the system used to implement them.

Class 1 algorithms are those that do not retain any information on memory usage and assume that all pages are equally likely to be referenced. A random replacement algorithm, in which the page to be replaced is chosen totally randomly, and a first-in first-out (FIFO) algorithm, in which the oldest page in memory is replaced, are examples of Class 1 algorithms.

Class 2 algorithms are ones which classify pages by their history of most recent use. These generally keep lists of status bits indicating if a block had been used or modified since the last check. Belady described a number of different methods of chosing the replacement page; the most effective, named AR-1, is based on a logical grouping of memory blocks based on the "in use" and "modified" bits. The blocks are grouped as "not used-not modified", "not used-modified", "in use-not modified", and "in use-modified", in that order. The algorithm indicates that the page for replacement is chosen at random from the lowest ordered (starting from "not used-not modified") non-empty group. The "in use" bits are automatically reset when the last unused page is taken. This ensures that there will always be a page not in use at replacement time.

Class 3 algorithms keep information on blocks in external storage as well as those in main memory. An example of this class of algorithm is the replacement scheme used on the Atlas, which always kept a free block in memory by chosing one and copying it out onto drum before it's needed.

Belady concluded that the optimal replacement algorithm required knowledge about future references, rather than previous references. He developed a model that did optimal replacement for programs with known paging sequences, for use as

a base of comparison for the other algorithms. He concluded that the random selection, FIFO and AR-1 algorithms justified implementation. A good algorithm strikes a balance between simplicity in randomness and complexity in information accrual. A purely random scheme may not be the most efficient, but it does provide a consistent level of performance and it is easy to implement. A more efficient algorithm may be very complex because of the information that it must maintain to make the best choice, and that overhead may offset the benefit received.

Belady wasn't the only one studying virtual memory and paging strategies in 1968. That year Peter Denning published several papers describing the working set model of program behavior, developed to help solve resource allocation problems [Denn68]. The working set of a process is defined to be the minimum collections of pages to be loaded into main memory in order for the process to operate efficiently. Denning estimated that it consisted of those pages that have been referenced in time Δt.

The working set principle states that a process may be run if and only if its working set is in memory, and a page may not be removed if it is a member of the working set of a running program. The working set principle was seen as a way to implement the principle of optimality. It states that the page to be replaced should be the one with the longest time until next reference, and if that is not exactly known, the page whose expected time to reference is longest. The working set principle maintains in memory a pool of pages that have been recently referenced, and those are the pages most likely to be referenced in the near future, since programs follow the principle of locality. The principle of locality states that references to a page tend to cluster and that programs tend to operate within a few pages over a short period of time.

The working set principle represents a resource allocation and paging policy. That a program may run only if its working set will fit in memory ensures that the program will be able to accomplish some useful work during its time slot. If there is not enough memory for a program's working set, the working set of another program must be deactivated to make room for it. The pages deactivated will be written out to disk as they are needed by the new process.

An objective of the working set theory was the prevention of the situation known as thrashing. Thrashing is a collapse of performance caused by over committing memory to too many programs. The system spends the greatest percentage of its time moving pages in and out of memory and doing almost no useful work. In a system not using a working set policy, one process can force the removal of another process's critical pages. When the second process is given the processor, it will soon reference the pages it lost, causing more paging I/O. If memory is still full, it may remove pages from another process's working set, ad infinitum. If there are enough processes in memory, it is possible to have all pages members of some working set, and trashing will result. The practice of allowing a process to page out any process's pages is called a global paging policy; a local paging policy allows a process to page out only its own pages. With a local policy, it is not possible for one program to degrade the performance of another.

Sharing is still possible using the working set policy, in that the shared pages would be members of multiple working sets and the "in use" bit would almost always be on, so that a shared page would rarely be replaced.

The working set policy is predicated on the belief that balancing memory demands is required to balance processor demands. Denning suggested that scheduling and memory management be closely related to achieve overall system balance. This was very different from all that had come before: until this time processor

utilization had been the overriding concern of most operating systems. That policy may also be the reason that virtual memory had received a bad name, since thrashing was not an uncommon problem with many implementations. The working set principle became widely accepted as a virtual memory control policy, and is used by many systems today.

In 1970 Denning expanded upon his initial work on virtual memory [Denn70]. In addition to further investigation of the working set principle, studies were done to determine optimum page size. The two factors that effect the page size are the amount of fragmentation of memory and the efficiency of the paging devices. Fragmentation is the inability to assign physical locations to logical addresses because the available memory is too small to be usable. Fragmentation can take two forms: external and internal. In a non-paged system fragmentation is external: many small blocks of unused memory are scattered about. If these small blocks are combined into a single block, the memory may be usable. Paged systems suffer from internal fragmentation. All pages in memory may be allocated, but there can be a significant amount of space wasted within the pages.

The page size has a direct affect on the percentage of memory lost to internal fragmentation: a small page size leads to less fragmentation. Denning found that the optimum page size, from a fragmentation viewpoint, was

$$(2*c*s)^{.5}$$

where c is the cost of fragmentation due to page tables vs. cost of fragmentation within a page, and s is the segment size. Having many small pages requires larger page tables, whereas large page sizes increases the number of words wasted within a page. Assuming a c of 1, and that the segment size is less than 1000 words, the optimum size is about 45 words or less.

However, minimizing fragmentation is not the only consideration in choosing page size. Pages have to be moved to and from secondary storage as part of the paging process. Small page sizes do not make effective use of the disk, since the seek time is significantly greater than the transfer time. The optimum page size for a disk transfer was greater than 500 words, based on the disk technology of the time. Most current virtual memory systems chose a page size that is a multiple of the disk sector size, as this is the most efficient way to use the disk.

Denning also discussed paging policies with regard to the time when pages were brought into memory. If a program's entire working set is brought into memory at the time it is to run, this is a prepaging system. If the pages are not brought into memory until they are referenced, it is known as a demand paging system. He concluded that the demand paging scheme is more efficient, because only those pages actually needed are brought in. He viewed prepaging as likely to be futile, and the time to swap in the entire working set is not worth the overhead. However, prepaging could be beneficial if it is done when the I/O channels are idle, and there is free memory available. The relative merits of prepaging and demand paging are still open to debate.

Denning and his contemporaries established a solid base of virtual memory management techniques. Research after this time concentrated on refinements of the techniques, using analytical models of program and system behavior.

4.5. Contributions of the Era - 1961 to 1970

Virtual memory originated as a means to run large program in a small space, without burdening the programmer. Hardware and software designs changed radically to support this new structure. Hardware designs reacted to software demands for faster addressing mechanisms that made virtual memory feasible. Software

developers were then able to concentrate on optimizing their management techniques. Never before had there been as strong a dependency between hardware and software designs.

4.5.1. Hardware Refinements

Virtual memory systems forced significant changes in computer architecture. Programs no longer addressed a simple linear array of storage locations. Program addresses were grouped into logical blocks, and words were referenced by a block number and an index within the block. These blocks might be segments, in which case an instruction address refers to a segment number and word within that segment. The logical groups might be pages, in which an address refers to a page and a word within a page. In either case, the hardware must map the block number and index into the specific word in the physical memory. If segmentation and paging were used, the hardware must provide even more support to map logical addresses to physical addresses. Software could have done the job, but it would be prohibitively expensive from a performance standpoint.

Hardware assistance is also needed in selecting pages or segments to be replaced. Over time the hardware aids have grown from a single "in use" bit, to a series of indicators about access rights and page modification. Page replacement requires transferring pages to and from disk or drum. The frequent need to replace pages promoted different disk designs. Fixed head disks with a head per track were developed to minimize the time to transfer a page from the disk.

Associative memories were developed during this period to speed up address translation. An associative memory is addressed by a pattern. The pattern, for example a segment number, is used to determine which cell in the associative memory to access. All cells are checked in parallel, so searching is very fast.

The associative memories are used to store some of the most recently used table entries. The Atlas PARs were the first associative memories.

Another technique used to improve virtual memory performance is cache memory. Cache memory is a small, high speed buffer for main memory. Memory requests are given to both the cache and the main memory; if the information is in the cache, it is returned to the processor immediately, and the main memory request is canceled. With careful design, most requests are satisfied by the cache, reducing the effective memory access time. The cache is updated with main memory information whenever it can not satisfy the request. Many caches also prefetch the next several words from main memory anticipating the next request. Prefetching is based on the assumption that most instructions and data are accessed sequentially.

For a period of time in the mid to late 1960's Extended Core Storage was a popular concept. An ECS was a slower, cheaper, random access memory that could be used instead of a paging disk or drum. At the time, ECS was faster than the equivalent mechanical storage devices. When LSI memory technology became widely available in the very early 1970s, ECS use declined, since the large fast memories were now relatively inexpensive. Also faster, cheaper disks helped eliminate ECS memories.

The development of larger memories did not eliminate the need for virtual memory. While small memory sizes stimulated virtual memory development, virtual memory provides too many other benefits, such as ease of sharing and more flexible memory allocation, to be phased out by large memories. In fact, larger memories helped virtual memory systems perform better, since more of the critical tables could be made memory resident, and the system could afford to keep pools of free pages available.

4.5.2. Software Refinements

Virtual memory had a profound impact on memory allocation techniques. Prior to its development, a program was allocated a single sequential block of memory; moving other programs if necessary. The granularity of allocation was an entire program. With virtual memory systems, the units of allocation were either segments or pages depending on the method used. A program was a collection of these smaller units, making it easier to find blocks of memory when necessary. Also, the entire program need not be resident to be run.

The addresses within a process on a virtual memory system are logical addresses, which are mapped at run time to some physical memory cell. This address independence makes resource allocation easier, in that any part of any program can be placed where it is most convenient. Address independence also makes sharing much easier since the shared code no longer must be part of the physical space of a program. The entire system becomes more device and memory independent, because there is no predefined relationship between logical and physical addresses.

The working set model established a method of determining how to allocate memory to different programs in order for the system to run efficiently. The working set policy ensures that the system is able to perform useful work, regardless of the number of users demanding service. The working set principle also puts restrictions on the page replacement scheme: when a process requests a new page, the replacement page can not be a member of another process's working set. Replacement pages must be selected from the process's own working set.

A host of algorithms to handle page replacement were developed. The one most commonly seen today is the least recent used algorithm (LRU): it chooses for replacement the page that has not been used for the longest time. This is based

on the idea that programs exhibit a tendency to use the same small set of pages over a short time interval. Therefore, the one used least recently is the least likely to be referenced in the immediate future. This approximates the optimum paging algorithm, which requires that the paging sequence be known in advance, which is not attainable in practice. This is the algorithm used in Multics.

Other paging algorithms developed were the random replacement and first-in first-out schemes described by Belady. Others were biased first-in first-out, in which the page chosen was from the process's own pages and selected from those via FIFO. The least frequently used (LFU) choses that page which has been used the least over some defined period of time for replacement. Other schemes involved user or system defined priority; the page to be replaced was selected from the lowest priority process. Within that process the page was chosen using one of the other algorithms. The choice of paging algorithm depends on the hardware support available and the complexity desired in the software.

5. Security and Protection Systems

5.1. Introduction

This chapter presents the development of security policies and protection mechanisms as they relate to operating systems. Many aspects of security and protection will not be covered; this chapter focuses on those developments that have had significant effect on the design of operating systems. Security and protection are still active research topics, so the chapter is necessarily incomplete.

Concern about the protection of system and user programs and data has been evident since operating systems were first developed. The earliest systems worried about protecting the system supervisor from modification by faulty user programs. When multiprogramming (especially interactive multiprogramming) became available, the protection problems became more acute. At the same time, increasing demand for sharing data and programs complicated the protection problem. As research progressed, security features were incorporated into operating systems' design at their inception. Eventually security was deemed to be a fundamental function of operating systems. The design was modified so that the heart of the system was the security kernel, with all other functions built on top of this kernel.

5.2. Definition of Terms

Basic terms used in the chapter are defined here; other terms are explained as they appear in the body of the text.

Access refers to the authorization to use information stored in the computer; it also refers to attempts to use that information.

Principal refers to that entity in a computer system to which authorizations are granted.

Privacy is the right of an individual to decide what personal information may be released, to whom it may be released and when it may be released. Privacy issues are beyond the scope of this chapter.

A process is a collection of program procedures and data operating together to perform some task.

Security refers to the policies and mechanisms that control who may use the information stored in the computer, and the ways they may use it. A policy is a guiding principle governing who may obtain or modify information. A protection mechanism is a device designed to enforce the security policy.

5.3. Early Influences and Motivations

Protection mechanisms have been the concern of hardware and software system designers since automatic control systems were developed in the mid 1950s. The most primitive resident supervisors provided standard I/O routines to users. As soon as this service was made available it became apparent that an erroneous user program could modify the supervisor, causing it to act erratically or not at all. The problem became more acute when the supervisor functions grew to include job sequencing and job set-up. If a user compromised the supervisor's code or data, man and machine time were lost in restoring the supervisor and rerunning the affected jobs.

The protection problems grew significantly with the change to multiple user systems. Not only did the system have to be protected from the users, but the users had to be protected from each other. Security policies had to be extended to secondary storage devices, as disks became more common and users started maintaining data files on-line. To complicate issues, users wanted to share data and code, so total isolation of users was not an acceptable security policy. These

were the motivating factors behind research into operating system protection mechanisms.

5.4. Historical Development

5.4.1. Early Concepts

In the late 1950's software supervisors provided job sequencing and set up functions, as well as libraries of commonly used subroutines and I/O functions. The systems were capable of overlapping the output of one job with the execution of a second job and the input of a third job. It became unacceptable to allow one user's undebugged program to compromise the operation of the supervisor or other user programs.

In response to these increasing pressures for protection mechanisms, hardware designs began to include special aids. Bounds or limit registers were used in the late 1950s to define a program's upper and lower physical memory limits [Stra59]. On every memory access, the hardware checked the address against the limits, and reported illegal access attempts to the software supervisor.

This approach solved the immediate problem adequately. However, the computer industry was growing very rapidly, and the drive was towards multiprogramming systems. With these systems came new protection problems, and more sophisticated techniques were needed to solve them.

5.4.2. Early Systems

The Manchester University Atlas, announced in 1961, was a pioneer in many areas of computer design [Kilb61]. It was among the earliest multiprogramming systems, it was the first virtual memory system, and we shall see that it developed a novel solution to the protection problems of the times.

The operating system was protected in several ways. The most critical routines were implemented in read only memory, where they were safe from meddlesome user programs. The computer's main memory was divided into fixed length pages, as were user programs. The supervisor maintained a directory which defined which pages belonged to which users. Users were only allowed to access pages which belonged to them. This policy was enforced with help from the hardware. Each main memory page had a register associated with it, and one of the pieces of information kept in it was a lockout indicator. The supervisor set the indicator on or off depending on whether the page was in the current user's list of owned pages or not. The cpu blocked any access to pages that had the lockout indicator on. When another user was scheduled to run, the supervisor altered the lockout indicators to reflect which pages belonged to the new process and locked out all other pages. The portions of the operating system that were run in main memory were protected from the user programs in the same manner.

The system had rudimentary protection for secondary storage as well. The main input and output facilities were magnetic tapes; the tape units could be configured as required to perform specific operating system services. To prevent problems associated with mounting the wrong tape on the wrong drive, and destroying potentially valuable system information, the system performed tape label checking. This appears to be the only special protection function for auxiliary storage.

The Compatible Time Sharing System, CTSS, appeared in preliminary form in 1962, and was in development for several more years [Corb62, Corb65]. This system represented the first major attempt at an interactive multiprogramming system. A method of identifying authorized on-line users had to be developed. Users were required to "logon" and identify themselves. Each user had his own private file tape on which he kept his data and programs; logging on made the tape

available to the user. Memory protection was provided via limit registers.

Other improvements were made during the early 1960s. Many machines had two operating modes, one for the supervisory software and one for users. When operating in user mode, all memory references were checked against the limit registers; in supervisor mode, the checks were ignored and certain privileged instructions, such as the I/O instruction or limit register instructions, could be executed. This scheme protected the limit registers from modification by users, thereby maintaining system integrity.

The concepts presented were soon adopted by other systems under development at the time. The field was dormant for several years before the next major breakthrough.

5.4.3. The Growth Years - 1965 to Present

Multics (1965) was the successor of CTSS, and represents a major milestone in operating system's design [Grah68, Orga72, Salt74]. It has been discussed in detail in the previous chapter, so only the protection related portions will be described here.

The protection mechanisms implemented in Multics are governed by a predefined security policy. There are five basic principles employed. First, protection is to be based on a default authorization of no access; permission has to be expressly given for access to allowed, rather than excluding permission on a selective basis. Second, every access to every object has to be checked against the current authorizations, since access privileges are allowed to change. Third, the protection system must not depend on the ignorance of the attackers for its effectiveness. Fourth, the principle of least privilege should be followed, which states that a process should be granted only those privileges necessary and sufficient to perform the

task assigned it; if the nature of the task changes, the privileges should change accordingly. Last, the human interface must be easy to use to encourage users to use the protection systems, rather than circumvent them.

One of the protection mechanisms relates to the addressing scheme. Processes are made up of named segments of arbitrary size. A segment represents any collection of information, such as data or subprograms, important enough to have a name; the segment is the smallest unit of protection on the system. Associated with each process is a table of segment descriptors, giving the segment size and the applicable types of access. If an illegal access is attempted, or a reference is made outside the bounds of the segment, a hardware trap occurs.

Segments can be shared among processes: each authorized process has an entry in its descriptor table pointing to the same segment. The creator of the segment is the owner, and determines the type of access permitted to other users. Each segment has associated with it an access control list (ACL), which specifies which users have what permission to the segment. The types of accesses allowed are combinations of read, write, execute, and append. When a process accesses a segment for the first time, the ACL for the segment is consulted to obtain the access rights for the segment descriptor. Through the ACL it is possible to completely isolate a segment from all other users, or to share it in any way desired.

Since groups of users may be working on a common project, or a user may develop a utility program to be made available to all users on the system, a more convenient way of forming the access control list was needed, to conform to the last protection principle mention above. To this end, users and processes are identified by a three part principal identifier. The components are the user name, a project identifier, and a compartment within the project. If a user wants only "Jones" to access his segments, he would specify in the ACL "Jones.*.*", giving

Jones access to the segment from any group or compartment. If the user wants all members of the Payroll group to access the segment, he would specify "*.Payroll.*". The type of accesses (read, write, execute, append) to be allowed would be listed. In this way a user can protect his files from other users, or even himself.

A user's segments are entered in a directory. An ACL is defined for each directory, either by the user or the system, to serve as a default for every segment entered in the directory. The user is free to change the default if desired, and to change the particular access list for any segments in the directory.

Multics was a long lived development project, and over time other protection features were added. One mechanism that began in software and was eventually implemented in hardware is the ring structure. [Grah68, Orga72]. The rings are numbered from 0, with 0 being the most privileged ring. The original design called for up to 32 system rings and up to 32 user rings.

The purpose of the rings is to provide intraprocess protection. Each segment has one ring associated with it; a process executing that segment thereby has that ring number associated with it. Any attempt to access a segment in a different ring, generates a ring crossing fault. On a ring crossing, a process is allowed to access segments with a larger (less privileged) ring number than its current ring number. An access to a lower numbered (more privileged) ring causes a trap to the operating system.

The concept was implemented differently to minimize the system overhead. Instead of having a single ring associated with a segment, it has a list of adjacent rings that it can access and from which it can be accessed, called the access bracket. This was done so that common service routines could be accessed by the supervisor and users without causing a ring crossing fault on most references. If a

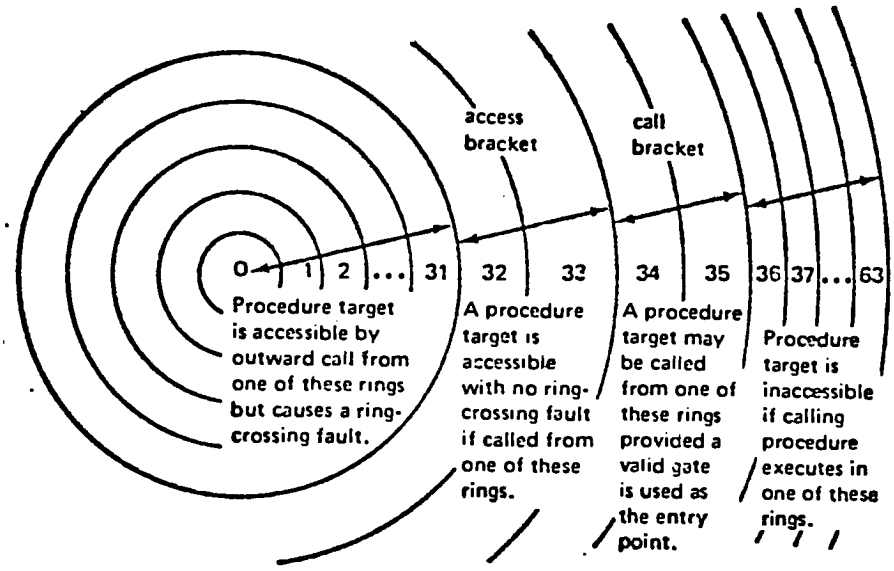
service routine was implemented in a single ring, only segments in that ring could access it without generating a fault.

There is also a call bracket , which represents a band of rings outside the access bracket that the segment is permitted to call. With each segment is a list of valid ring entry points called a gate list. When a call is made to a segment in a lower ring number, a fault occurs. A process called the gatekeeper checks if the call is to an entry point in the gate list. If it is, the call is allowed, otherwise it fails. Only calls can transfer control to a more privileged segment. A segment can not obtain more privileges than it was originally granted, but it can have a more privileged segment do work on its behalf. An example of the operation of the Multics ring structure is given in Figure 5-1.

The gatekeeper also pushes the return address and the ring number of the calling segment on the stack, and if an attempt is made to return to the segment through a different path, a fault is generated. Thus entry into and exit from the interior rings is carefully monitored. In addition to these checks, any arguments passed to inner ring segments are checked to be sure that the calling segment has access privileges to the argument segments. This keeps users from tricking the operating system into disclosing information that the user does not have rights to.

The protection mechanisms in Multics are sophisticated, and some of them have been incorporated into other operating systems, including current commercial operating systems such as DEC's Vax/VMS and Bell Labs Unix.

In 1966, not long after Multics was announced, Dennis and Van Horn published a paper called "Programming Semantics for Multiprogrammed Computations" [Denn66]. In it they described a number of protection concepts, one of which was the idea of capability lists (C-lists). Each capability in the list located some



Example:

A target procedure, possibly a system utility, has an access bracket consisting of rings 32 and 33, and a call bracket of rings 34 and 35. If the target procedure is called from either of the rings in the access bracket (32 or 33) the access is allowed, without a ring crossing fault. If the target procedure is called from a procedure executing within the call bracket (34 or 35), a ring crossing fault occurs, and the access is allowed only if the calling procedure is entering through a valid entry point. If the target procedure is called from a procedure executing within a lower numbered (higher privileged) ring, access is permitted, but a ring crossing fault occurs. If a procedure executing withing rings 36 through 63 calls the target procedure the entry is blocked since the call is from a less privileged ring that is not within the call bracket.

Figure 5-1
Multics Ring Structure and Operation

---

object in the system and indicated the actions that could be performed on the object. An object was any part of the system that required specific protection. An object was only accessible through a capability. A set of processes having a common C-list was considered a computation, and operated within a sphere of protection defined by the C-list. The segment descriptor list in the Multics system is an

example of a C-list, in which the only type of object is a segment.

The idea of a set of processes forming a logical computation unit had interesting ramifications. Dennis proposed a number of primitives designed to control parallel programming, in which a process may spawn other processes, each proceeding in parallel and perhaps joining later. Creating a new process involved granting capabilities. The parent process could selectively grant capabilities from its own capability list, creating a less powerful offspring if desired. The parent process had access to the state of the child process for exception handling. This could be very useful in debugging, where the debugger could place a protective sphere around the program being tested. An even more promising application would be to make the operating system the parent process and have it selectively grant capabilities to user processes.

It took several years for the ideas presented to be fully explored. The articles for the next several years concentrated on privacy issues, physical security measures and data encryption techniques, rather than secure operating system designs. The only articles relating to operating systems were methods of user identification and techniques for protecting user passwords.

In 1969 Butler Lampson expanded upon the ideas of Dennis and Van Horn [Lamp69, Lamp74]. The problem he tackled was the issue of protecting the protection mechanisms. Unless the mechanisms, such as the capability lists, were themselves protected from user modification, the security of the system could be compromised.

He defined a capability to be the protected name of an object. A domain was defined to be a group of capabilities, and a process executed within some domain, and could exercise the capabilities that belonged to that domain. When an object

is created, a capability is created by the system to name it. Capabilities belong to the system, and can only be manipulated, not created or destroyed, by the users. Capabilities should be hidden inside the supervisor to prevent users from changing them. Lampson proposed a tagged hardware architecture to help protect the capabilities. Each word has a tag field, indicating if users or only the supervisor can modify the word; capabilities would be marked as supervisor only, and the hardware would be responsible for the integrity of the capabilities.

Several methods were proposed to describe the relationship between domains, objects and capabilities. One possibility was to list all the domains as rows and all the objects as columns in a matrix A. The element A[I,J] specified the access rights that domain I had to object J. However since there could be many objects and domains, and most elements in the matrix would be empty, it was not practical to store the information in matrix form. Capability lists consisting of object name, access attribute pairs were used to describe the access matrix in a more compact form.

A computation consisted of several processes, each running in their own domain. Since each process ran in its own domain, small programs could be collected together to form larger programs without extensive advance planning about a common environment. Each piece did its function with only that collection of capabilities required for the job. If data needed to be passed between processes, capabilities for the data objects could be passed.

If control was to be passed from one domain to another, care had to be taken that security was not compromised. Lampson proposed an idea of protected entry points for domains, similar to the gates in Multics. A gate was a capability for a particular domain. Gates could be passed as capabilities, giving the calling program access to the routine entered by the gate. Like Multics' gate system, the domain

and return location were saved on a stack and the return must be through the same gate.

There was also a need to relate capabilities, which the system used to name objects, to symbolic names for the users. Another object type called a directory, contained information for mapping the external user names into capabilities.

The problem with a system based on capabilities is that the implementation and the logical design are complex. If an object, such as a file, is to be shared, a capability for that object must be passed from the object's owner to all those that want to use it. A method of making the procedure easy for the user, while maintaining the integrity of the capabilities was proposed by Lampson. An access key is an object that identifies other objects as belonging to a particular user. For example, if a user's name is Jones, his access key would be Jones*, where Jones* is a system provided, uniquely encoded id. To give Jones access to another user's files, "Jones" would be added to the access list associated with the file. When Jones' computation accessed that file, the computation would have the capability Jones* which defines the object Jones, which would be in the access list of the file. To grant access to another user, only the user's name need be known, not the user's capabilities or unique id. The idea of access keys can be expanded to groups of the type described in the discussion of Multics.

There are other problems with capability based protection systems. Some mechanism has to be provided to allow an owner of an object to revoke the capability to that object from selected users. Once a capability has been granted there is no easy way to determine which users have that capability. Several methods have been suggested that could solve this problem. One is that a segment would be associated with each user, holding all the capabilities for a user. Only these segments, which would be known to the system, need to be examined to revoke a

capability or to take inventory of the users having a capability. Another method proposed uses indirect objects as a means of granting and revoking capabilities. Capabilities would be given only for an indirect object that points to the real object. There would be an indirect object for each principal given access to the real object. To cancel the privilege, the indirect object would be deleted, only affecting the one user associated with it [Need72].

The basic problem of a system having only capability based protection mechanisms is that the privileges to objects are bound to users when the capability is copied, and there is no information retained to reverse the binding. A capability is like the key to an unguarded locked door: having a copy of the key is sufficient to open the door. Once the key has been given out, there is no easy way to recover it. Even if the key was returned, there would still be some question of how much protection the lock now gave. What is needed is a way to guard the door.

For this reason many systems make use of a combination of capabilities (keys) and access control lists (guards). The capabilities are for use by the system, and the access control lists provide a way for the user to keep track of his own objects and the authorizations given to other users.

In 1970 a totally different approach to protection was taken, as a byproduct of an attempt to provide a more flexible multiprogramming system [McCu70]. CP-67/CMS, from IBM, was a product that allowed multiple operating systems to run simultaneously on one machine. Each user could run under his own operating system, which had the effect of totally isolating the user from any other users. Several users could run under one operating system copy and share programs and data subject to the limitations of that operating system. They would still be separated from any other users on the system. The problem with this approach was that the multiple levels of control software resulted in poor performance.

However the idea of providing users with their own virtual machine had potential as a protection mechanism.

The concepts presented to this point were topics of discussion until about 1974. By that time government pressure for secure computing had been building and culminated in the Privacy Act of 1974. Also the Multics project, which was the dominating system of the late 1960s had peaked out by the first third of the 1970s. Researchers were branching out into new territory. The new research trend was towards designing a very small, security oriented operating system nucleus that could be verified by formal techniques.

The early basis for this new design philosophy was work done by Per Brinch Hansen in 1970 [Hans70]. He had developed a system nucleus, which provided only primitive functions, on which different operating systems could be built to satisfy various needs. The nucleus governed dynamic creation and control of processes, input and output, and interprocess communication. Interprocess communication was through messages, sent through the nucleus, within which the senders and receivers of messages were identified for protection. The nucleus provided these services as a logical extension of the hardware, and operating systems were to be implemented on top of it. The operating system would provide scheduling, file services, language processors and other utility functions.

The security kernel concept developed in 1974 was similar to Hansen's nucleus, in that it represents the lowest layer in a hierarchy of system service layers. The difference is that the security kernel consists of only security functions and encompassing all security functions. The rest of operating system is built using the kernel primitives as a base.

The advantages of this approach are that the security related code is all in one place and thereby less subject to penetration. This is especially true if the primitives are implemented as uninterruptible actions, so there is no data integrity problem during the time between authentication and authorization. If an interruption is allowed between the check for rights to privileges and the granting of privileges, it is possible for the system to check one process and then grant that process's privilege to a different process. A user could gain system privileges by switching places with a system task after the authentication step.

Also, since the kernel is the lowest level, it does not rely on the correct operation of any other portion of the system for its own proper operation. Therefore if the security kernel is correct, the outer layers of the operating system can not degrade the security of the system. Since the kernel is compact, it can be subjected to formal verification techniques to prove its correctness.

Several systems used the kernel approach to implement operating systems. One system implemented in 1974 using a security kernel was the UCLA-VM system by Popek and Kline [Pope74]. The security policy was based on the concept of objects and capabilities. The kernel provided mechanisms to manipulate objects, but had no concept of the internal structure of the objects. The kernel included the interrupt handlers, which provided basic control over the hardware. The kernel was kept very small, as the maximum size program verifiable at that time was about 2000 instructions. To verify the kernel two things were required: the explicit definitions of security had to be translated into predicates for mathematical interpretation, and properties of the primitives had to be verified. This second phase included tests that there be no order of invoking the primitives that would cause a security breach.

The system implemented on top of the security kernel was a virtual machine. A virtual machine was the major object of the system. Since the kernel couldn't provide intra object protection, there was no protection within a virtual machine. However, there was protection between virtual machines, since each was a protectable object. Each user ran his own copy of the operating system on his own virtual machine, which provided total isolation from other users. This approach was taken because of its ease of implementation. This allowed the researchers to concentrate of the kernel portion of the project rather than on the higher level operating system services.

A different approach to the security problem was taken by MITRE Corp in 1974 [Lipn74]. Their system was based on the premise that

> current operating system software is incapable of preventing access by any program to any information accessible to the processor, thus information that is to be made inaccessible to a user must be isolated from the processor.

To this end, they used a minicomputer front end to act as a reference monitor. The monitor controlled all communications to and from the central processor, and checked accesses to all objects in the system. The monitor was implemented in a front end processor to conform to the three following premises. First, every access to every object was to be checked by the monitor. Second, the monitor and its data base were to be protected from the users and third, the monitor had to be small enough to be verified. Putting the reference monitor in hardware between the user and the main processor ensured that these constraints were met. This represented an interesting approach, providing the security needed with the technology available, but it was not adopted elsewhere.

In 1978 and 1979 more security kernels were developed. Popek and Kline were working on a new project, developing a security kernel for Unix [Pope78,

Pope79]. As part of the project they investigated general issues in kernel design. Basic factors affecting kernel design were determined to be the security policy, the variety of primitive functions, and the hardware characteristics.

The security policy determines broad aspects of security, such as the granularity of the objects, operations on the objects, whether the sharing is supported and whether data security is the only concern or if denial of service is considered as well. The granularity of objects refers to the precision with which objects can be defined. For example, is a file the lowest level object or can records or data fields within the file be defined as individually protectable objects? Given the objects of the system, what type of operations are to be provided for their manipulation? Can users define their own object type and have the system provide protection for them? What types of devices will be supported? (The more unique device types supported, the more complex the kernel becomes.) Are the primitives sufficient to easily build an operating system using them? Are all the primitives necessary? (The larger the kernel the harder it will be to prove it correct.) These are all policy questions relevant to kernel design.

The hardware can have an effect on the design: especially in the area of I/O primitives, certain hardware characteristics can make implementation more difficult. If the hardware provides support for capabilities, such as a tagged architecture or capability registers, the kernel can be made much smaller.

Popek and Kline proposed a layered approach to kernel design as a means of reducing the size of the most critical areas of the kernel. They distributed functions among trusted and distrusted processes. Trusted processes are not part of the kernel, because their function is peripheral to the main task, but their correct operation is essential to the security of the system. Trusted processes depend, for their correct operation, only on the correctness of their own implementation and

that of lower levels in the kernel. They can not be compromised by distrusted processes, because they do not rely on correct operation of the distrusted processes. An example of an operation that could be moved into trusted code is the user authentication process that is activated at logon.

Examples of operations that can be moved out of the kernel and into distrusted processes are memory management and process scheduling. In the case of the scheduler, distrusted processes would determine what job to run next, and then call the kernel to perform sensitive functions, such as context switching. The distrusted processes would be able to move the objects it managed, but can not examine or change their contents. That way the security of the objects is intact, while minimizing the direct responsibilities of the security kernel.

The Unix security kernel was implemented as a multi layered system. The kernel itself implemented abstract object types and operations on those types, and it was the only portion of the system that could issue I/O instructions. Outside the kernel were some trusted processes upon which the system security depends. The policy manager was capable of providing extensions to the kernel defined object types, such as file system object types. (The kernel itself only supported four types: processes, pages, devices and capabilities.) The other trusted process was the dialoguer which owned all terminals and was responsible for user authentication. The dialoguer indicated to the policy manager what user was to be associated with each terminal process. Between the kernel and the user process was a Unix interface that provided an environment for the process, and handled communications between the user and the other portions of the system.

Other groups were working on a security kernel for Unix at the same time [McCa79]. One was KSOS, Kernel Secure Operating System, from Ford Aerospace. The system consisted of a security kernel, a Unix Emulator and non-security system

software. Like the UCLA kernel, the KSOS kernel supported only a limited number of object types; these were processes, process segments, files, devices and file subtypes. The only object type that was Unix specific was a file subtype for Unix style directories. The system was designed to be very general, so that other operating systems would be built to run on it via different emulators. The Unix emulator provided user functions, and gave Unix specific properties to the kernel defined objects to provide a Unix compatible user interface.

An interesting protection mechanism was used on this system to prevent users from masquerading as the logon process and stealing passwords. The terminal was treated as two separate devices, one secure and one insecure device. The secure device was only accessible through privileged software. When a special attention character was typed, the device was switched to the secure device. This attention character was used as part of the login process.

Another system developed in 1979 was PSOS, Provably Secure Operating System, by SRI [Feie79]. This system was also based on capabilities, but the protection for them and operations on them were intended to be in the hardware. The system was not kernel based; it was a collection of hierarchical modules, each defining new abstractions composed from lower level abstractions. Capabilities were the lowest level abstraction, and were used to uniquely name the abstract objects created at higher level. Processes, pages, and segments are examples of higher level abstractions in the system. Protection policies were implemented using subsystems of abstract objects. Each subsystem would have to be proved correct, for the entire system to be verifiably secure. The system has not been implemented on any currently available commercial hardware.

5.4.4. Current Activity

The concept of verifiable security kernels is still under investigation. The concept has not yet caught on in commercial products. However the research is bound to continue, motivated by the need for secure systems, secure networks and government pressure for privacy controls.

5.5. Contributions of the Era

Since this topic is still an area of active research, the list of contributions can be expected to grow over time.

5.5.1. Hardware Refinements

It is difficult to distinguish hardware changes influenced by protection from those motivated by multiprogramming, since the two were so closely related in their early phases. Hardware limit registers, privileged instructions, and dual operating modes were all a result of pressures from these two areas.

The original concepts of segmentation were developed to help make multiprogramming easier to implement, but it had added benefits related to protection. Access control information was added to the segment descriptors, and the hardware became responsible for access checking. These checks were more involved than the simple memory limit checks on earlier machines. These checks validated not only memory limits but also the type of access being made.

In the early 1960s and for many years afterwards there was a supervisor and user mode on many machines. As protection systems became more complex, it was useful to have multiple levels, such as kernel, supervisor and user modes. Attempts by processes to execute instructions or processes in more privileged modes generate traps, so that the operating system can intervene and determine what action is

appropriate. Having multiple modes makes it easier to implement rings of protection, with hardware traps helping to enforce gatekeeping between rings.

When capability based protection mechanisms came into use, special hardware was desired to speed up access checking. Some machines, including the Honeywell 6180 on which later version of Multics were implemented, were equipped with a number of capability registers, that kept information on what access rights the running process had. These registers were maintained as part of the operating environment of a process.

Tagged architectures, where words in the system have a field indicating under what operating modes the word can be accessed, were also influenced in part by capability based protection systems. With increasing concern over security and the number of modern systems that use capabilities for protection, this type of architecture may become more common.

5.5.2. Software Refinements

There have been many software techniques developed as a result of research into protection mechanisms. Logon procedures used on most modern interactive system were developed during this period. There are numerous encryption techniques and methods to prevent passwords from being stolen. Directory structures were developed as a means of organizing the growing amount of online user data, and to provide a way to protect that data. The directory encapsulated the user files and provided a gateway to the data. Within the directory a user could selectively control access to his files.

Two basic protection architectures were developed: ticket and list controlled. If one develops a general model of protection in which there is an impenetrable wall around each object needing protection, with a guard posted at the gate in the

wall, then ticket and list controlled systems define the action of the guard.

In a ticket controlled system, each user having access to an object is given an unforgeable ticket to that object. When an access is attempted, the user presents the ticket to the guard. The guard matches the ticket against the identifier it knows about, and if there is a match, the access is allowed. The guard knows nothing about the users, just that anyone with a ticket can be allowed to access the object. Capabilities are a form of ticket; passwords are another form of ticket. The key concern in a ticket based system is that the tickets must be unforgeable and that they cannot be copied and passed indiscriminately. Copy bits in the tickets can be used to indicate whether the ticket can be propagated.

In a list controlled system, the guard has a list of all users who can access the object. The Multics access control list is an example of this type of mechanism. The problem with this type of system is that access checks involve searchs through the list, which can be time consuming, if done on every access. Users are often grouped so that the lists are more easily controlled by the owner of the object. The lists make it easy to keep track of what users have access to each object. Many systems use a combination of ticket and list controls to combine the performance advantages of a ticket system with the user interface of a list controlled system.

Virtual machines were developed as a means of allowing one machine to run many operating systems. The original motivation was to provide flexibility, by allowing users to run earlier operating systems on new machines. The concept provides a secure environment for users of the virtual machines. There may be security problems within the operating systems on the virtual machines, but if each user is using his own copy of the operating system, the technique is secure. Unfortunately there is a heavy performance penalty associated with the multiple

layers of complex system software.

The organization of protection mechanisms within the operating system has changed over time. Protection mechanisms until fairly recently have been "ad hoc". They were not an integral part of the design, or if they were, their implementation was such that holes in the protection system were inevitable. Making a system secure consisted of patching known leaks and running tests to try to break the security. This approach has been replaced, in the research community, by the security kernel concept. In that approach the security policies are thoroughly defined and verified by formal techniques. The kernel consists of all and only those aspects of the system relating to security. Once the kernel has been proved secure, the rest of the operating system can be built on the security base without fear of breaking security.

6. Distributed Processing Systems

6.1. Introduction

This chapter traces the development of operating systems for distributed computer networks. The primary focus will be on software for networks of autonomous computers rather than for multiprocessing systems. Multiprocessing systems will be briefly discussed in their role as the base of modern computer networks.

The desire for multiple cooperating processors arose early in the development of computers. Multiprocessor systems first appeared with the Univac LARC in 1956, and have been available in many forms ever since. A different trend in the design of cooperating computers developed in the early 1970s. Functionally complete computers were linked by communications lines to share storage and computational resources. The computers in these networks were often from different vendors, each running its own operating system.

Early network software handled physical communication and primitive data transfer; users worked directly with each host operating system. More recent network software has attempted to make the network operations transparent to the user by providing a standard interface to the network resources. Methods for providing this user environment are still an area of intense research.

6.2. Definition of Terms

In the area of distributed computing there are many terms used. To establish a common base for the coming discussion, some terms are defined here.

Coupling refers to the logical and physical relationships between processors in a network. A closely coupled system is one in which the individual processors work as a unit. A loosely coupled system is one in which the processors are independent

units that can function by themselves, or can cooperate with the others to perform some function.

A distributed computer network is composed of multiple, fully functional computers. In this chapter, this will usually mean the opposite of a multiprocessor system.

A guest operating system provides a standard command interface to the users by acting as a mediator between the user and existing operating systems of host computers.

A multiprocessor is a computer system composed of multiple processing units sharing a common memory and controlled by a single operating system. Typically, the processors do not have enough resources to function as standalone computer systems.

A node refers to one or more computers connected, as a unit, to a network.

6.3. Early Influences and Motivations

The roots of computer networks extend back to the middle 1950s. At that time computers were expensive and slow. They were being used for large scientific and mathematical problems; a computation could take hundreds of hours to complete. This caused two problems: first, the machine was not available to other users for the duration of the computation, and second, the machines were unreliable and rarely remained operational for several hundred consecutive hours. Increased speed and reliability were needed to improve throughput. The technology had not advanced enough to increase the speed through faster circuitry, and component reliability was still unsatisfactory. As a result multiprocessor systems were developed, starting in the late 1950s, that increased the speed through parallel operation of

the processors, while the redundancy aided reliability.

These concepts represent the early motivations for distributed processing. The motivations and techniques changed as computers became more common and sophisticated.

The development of distributed systems was limited to multiprocessing systems until about 1970. By that time many changes had occurred in the hardware and software technology that affected the progress of distributed processing. Hardware costs had decreased dramatically since the 1950s. As a result, many corporations had more than one computer. Also, computers had diversified over the years: manufacturers had different product lines for commercial processing and scientific computing. In response, corporations started to decentralize computing by providing different groups with machines suited to their specific needs. Minicomputers had been available for many years, and were starting to be more common. These systems had small memories and limited secondary storage, but were inexpensive and easy to justify for single projects. The computing environment changed from one large computer in a central location to many smaller computers in diverse locations, each with different capabilities and resources. This was to be a key factor in the development of distributed computer networks.

6.4. Early Concepts

Many software changes had taken place since the multiprocessors of the late 1950s. Interactive and batch multiprogramming systems were common. Users were accustomed to sharing the system hardware with others, and often used the facilities to share code and data as well.

Other significant changes had taken place in the design of programs. Programs had been viewed previously as sequential groups of instructions, to be

executed serially. Dijkstra had proposed in 1968 that programs be viewed as collections of cooperating sequential processes. Processes could run in parallel, each at its own speed, synchronizing as necessary with the other processes in the computation. This had several ramifications. If a computation consisted of many parallel processes, these processes could run on different processors. All that was required was a way to pass synchronization information between the processors running the cooperating processes.

It was also recognized that an operating system could be constructed as a collection of cooperating processes. Rather than being a single monolithic program, an operating system could provide a nucleus of primitive functions, with a collection of service processes built on the nucleus. This decentralized approach made it possible to implement the functionally distributed operating systems used in computer networks.

The philosophy of an operating system nucleus, with cooperating processes providing higher level system functions, coupled with the resource sharing philosophy of timesharing systems led to the development of computer networks.

6.5. Historical Development

6.5.1. Early Systems

In 1970 the ARPA computer network (ARPANET) was announced. This was an experimental system developed with the support of the Defense Department's Advanced Research Projects Agency (ARPA). The goal of ARPANET was to provide a base for research into network hardware, software and communications technology [Robe70]. ARPA recognized that a wealth of software existed throughout the computer community, but transporting that software to different sites and different machines was costly. Timesharing systems like the GE Mark II service

demonstrated that resources could be shared by many diverse users on one machine. The next step was to extend the timesharing concept across machine borders, thereby eliminating the need to transport software.

The network consisted of existing independent computer systems at various sites throughout the nation. There were about 20 sites in the original network; these were chosen to represent different application areas and machine types. An objective of the ARPANET was to make all network resources available to every node, in such a way that programs could be used remotely. Any program should be able to request the resources of a remote computer much as it would call a subroutine.

There were some difficulties for the users, because sites were running different machines with different operating systems. To access remote resources, the user had to be aware of where those resources were located; there was no automatic routing of requests along the network. The easiest way to use a remote system was to logon to it. To do this, the user logged on to the local system, and used the network to access the remote system. Commands to the remote system were be routed by the local system through the network.

There were many problems associated with this method of remote access. First, it was burdensome: a user had to be familiar with the operating system of each different host, and he had to have a separate account on each machine he used. Each of these accounts could have data files and programs, but there was no coordination of the accounts. The user was responsible for remembering where all his files resided, which was difficult if he had accounts on many machines.

Some interesting software issues arose from the remote logon concept. The user was connected to two systems simultaneously, with the local system acting as

a messenger between the user and remote system. However, the user had to communicate with the local system when the remote system failed or the remote's input buffers were filled. Also there had to be some way transfer files between systems; therefore, the local system had to examine each user request to determine the action to take, requiring a way to differentiate local and remote requests. The TELNET protocol provided these services.

TELNET's remote logon and file transfer capabilities made network use somewhat easier for (but not transparent to) the user. In 1973, RSEXEC was developed to provide a transparent interface to the ARPANET PDP-10's that ran TENEX [Thom73]. RSEXEC consisted of a group of programs residing at each host. The host systems executed RSEXEC programs as user tasks. These tasks intercepted user commands and host responses, and converted them to a standard form. Through this mechanism, RSEXEC attempted to make the TENEX hosts appear to be a single timeshared system. It extended the affect of user commands to all TENEX hosts, so users did not need to know which system they were using. One account gave access to all of the systems.

To coordinate user activity, the system maintained a profile for each user. The profile described the user's subdirectories on each TENEX host. The profile directories formed a composite directory listing all the files belonging to the user. File references were checked against the composite directory; if the file was not in the composite directory, RSEXEC requested the file from the local host.

RSEXEC also facilitated sharing of computational resources. Resident on each host was an RSEXEC server process that communicated with the RSEXEC server processes on the other hosts. The server was responsible for monitoring job requests for its host, accepting or rejecting requests from other servers based on the local load. Thus, resource sharing was not done at the expense of local users.

In summary, ARPANET provided a testbed for many ideas on computer networks. It was the first collection of diverse computers connected over long distances to achieve resource sharing. Many basic concepts for communications protocols and communication links came out of this project. However, ARPANET was just the beginning of user oriented network software.

By 1974 a different type of network, C.mmp, was under development at Carnegie Mellon University [Wulf74, Wulf78]. C.mmp was a homogeneous minicomputer network. While each processor was capable of operating by itself, the memory was shared among all the processors through a common memory switch. The kernel software for C.mmp, Hydra, was designed to provide a flexible environment upon which to build an operating system. Hydra provided I/O device support, capability based protection, and limited scheduling services. Most traditional operating system services, such as a file system, were developed by the users on the Hydra base. Hydra represented one of the first attempts at a truly distributed operating system.

6.5.2. Growth Years - 1976 to Present

Research on network operating systems continued to develop on two divergent paths. Hydra represented an operating system designed from its inception to control a network of processors. Other network operating systems were designed to provide network control services for computers with existing operating systems. This second class of network operating systems are referred to as guest operating systems.

The goal of a guest operating system is to provide a standard interface to a variety of existing host operating systems [Kimb76]. A basic assumption made is that the host operating systems can not be modified. Given this, the guest operating system must translate user requests into a sequence of commands acceptable to

the hosts, and it must translate host messages back into a standard form. The network operating system is also responsible for balancing resource requirements against the resources available, while allowing host operating systems some control over the demands from the network. In particular, the hosts must be able to ensure that local users are not penalized for the system's participation in the network.

To accomplish these goals, a guest operating system has to provide a set of primitive functions. These functions include some form of user communication with remote processes. Included are message processing primitives to create, forward and coordinate messages.

Other services required are data migration functions, which allow users to create, access and move files on remote systems. The ideal form of data migration is transparent access to remote and local files. This requires a means of retrieving records from the remote file, performing any translations required, and reconstructing the logical record required by the user. This presents difficulties because the network operating system must be aware of the internal structure of the host's file system, adding an extra level of complexity. Most real systems transfer whole files, or severely restrict the types of files supported by remote record access.

Guest operating systems must also provide some network job execution functions. The functions allow a user to run jobs on different systems in the network and possibly execute separate job steps in parallel on separate machines. The specific functions required would depend on the user environment the guest presents: transparent or user controlled network operation.

Finally, the guest must provide some primitives to implement the network operating system job control language (JCL). In this way, users can employ a common network JCL instead of the JCL of each host. Since the guest operating system must translate its JCL into the host operating system commands, the network operating system JCL is limited by the capabilities of the hosts on the network. To access the full power of a given host, users have to issue commands in that host's language.

The National Software Works (NSW), developed in 1976, represented a step forward, as it was the first major attempt at a guest system for a heterogeneous network. [Mill77]. Unlike RSEXEC, which provided a guest operating system for only the TENEX hosts, NSW provided a user environment for all ARPANET hosts.

NSW attempted to make the tools (programming utilities) on different hosts available to all users as if they were on the local system. This had to be achieved with no changes in the host operating system and with little or no modification of the individual tools. A Foreman process at each host was responsible for host specific details. For example, there would be a Multics Foreman to translate NSW commands to Multics commands and vice versa.

A different version of the Foreman existed for each unique host operating system. The Foreman acted as a spokesman for the tool, intercepting all communications between the tool and the rest of the network. The Foreman delegated the tool's requests to the appropriate network components. This allowed the output of a tool on one system to act as the input of a tool (ala a Unix pipe) on another system. Thus users could combine tools on various systems to perform a particular job.

The project was successful in providing a program production capability through the use of tools on diverse hosts. However, it did not provide more general forms of resource sharing, such as remote job execution or network file services. For this reason it did not achieve all the goals of a network operating system.

The next attempt at a guest operating system for the ARPANET came in 1978. The National Bureau of Standards developed XNOS, an Experimental Network Operating System, to run on all the interactive systems on the ARPANET [Kimb78]. XNOS was implemented in specialized processors, called Network Interace Modules (NIMs), that were used to interface the host machine to the network. The NIM handled all the communications with the network, passing to the host only those messages directly involving it. This approach minimized the impact of the network operating system on the host, because the network functions were placed in a separate processor. What is more, the common code of the network operating system, not related to the host operating system interface specifics, could be used on each NIM without any recoding, since the NIMs were identical. Host specific code would be developed just once for each unique host operating system.

The centralized design of the network operating system achieved through the homogeneous nature of the NIMs reduced the projected costs, and the design facilitated incremental expansion of the network. To add a new host, a NIM with the host specific software would be added to the network. If the host ran the same operating system as some host already on the system, there was no new software required. If the host was different from all the others, the only new software was the host specific interface to the NIM.

The XNOS user interface was also implemented on the NIM. Users logged onto the NIM, and XNOS coordinated their interactions with the network. XNOS

maintained a network wide directory system for each user, defining the files he could access, where they resided on the network, and the types of access permitted. XNOS mapped user names in the network directory to the host system's name for that same file. The user was unaware of the host names or internal representations of his files.

XNOS supported a form of network job execution. The user specified the system he wanted to use; if necessary XNOS would transfer data files to this system, prior to execution. Also supported was remote access, allowing run time access to records on other systems. An XNOS translator overcame the differences between host file structures. It was possible to set up a network job where different steps were executed in parallel on different hosts; however the user had to explicitly set up the job that way. To coordinate the parallel processes, XNOS supported interprocess communication across processors. The functions provided were simple signal and wait primitives.

XNOS came closer to meeting the requirements of a network operating system than any previous system, but it only worked with the interactive hosts on the ARPANET, excluding the batch systems. Network operation was not totally transparent to the user, as evidenced by the network job facilities. However, it did provide a comfortable user environment for resource sharing, especially through the network file system and common command language.

About the same time that XNOS was developed, Carnegie Mellon University began working on a new multiprocessor system called Cm* (see Fig. 6-1) [Jone79, Oust80]. Cm* consisted of multiple DEC LSI-11 processors, grouped in clusters. Each computer in a cluster was connected through a memory switch (Slocal) to a map bus, controlled by a Kmap. Each Kmap was a processor that mediated intra- and inter-cluster data transfers.

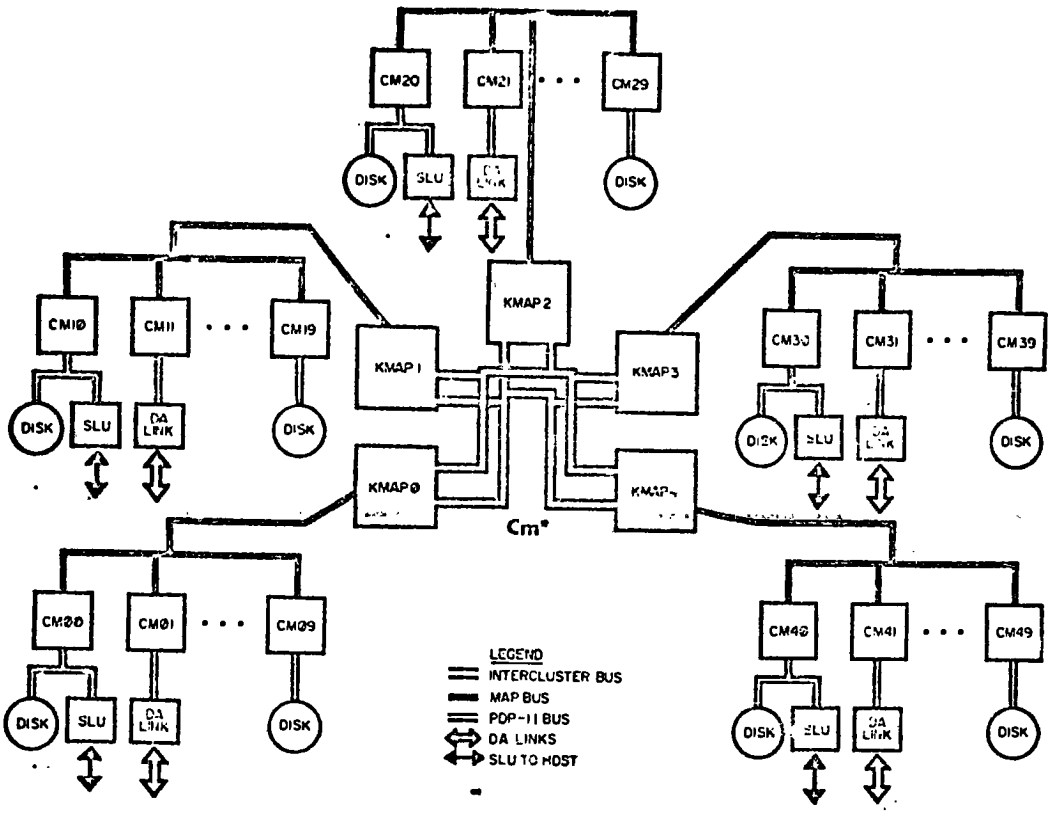


Figure 6-1
Cm* Configuration

The memory of Cm* was the union of the local memories for each processor. The Kmap and Slocal interfaces provided access to the memory of the other processors. Because of this multilevel memory switching, there were substantial time penalties associated with non-local and cross-cluster memory accesses.

The first operating system for Cm* was StarOS. The nucleus consisted of primitive functions called instructions. Instructions were short, uninterruptible

sequences of processor commands. Much of the StarOS nucleus was implemented in Kmap microcode; instructions not in the Kmap resided on each LSI-11 and were executable only in the kernel mode. Since a copy of the system nucleus was kept on every processor, the individual processors would be able to function even if part of the system failed.

StarOS implemented a two layer scheduling system. At one level were routines that determined global scheduling policies, computed job priorities, assigned jobs to particular processors, and set the execution quantum. This scheduler was responsible for ensuring concurrent execution of parallel processes when possible.

The Multiplexor formed the second level of the scheduling system. This program determined which process to execute next on the processor. Several low level scheduling policies were possible, such as first-come, first-serve or round robin; the policy was set at system initialization.

The concept of a task force was fundamental to the StarOS design. A task force was a collection of many small cooperating processes, whose code and data combined to accomplish a single task. Each of the task force processes implemented a specialized function, and therefore each needed access to only small amounts of data or code. The number of active processes in a task force could vary with the number of processors available. Some processes could be replicated and their data partitioned; individual copies worked in parallel on a small subsets of the data. The task force was the unit used for global scheduling and authorization for resources in the system.

Except for the small nucleus, the operating system itself was implemented as a task force. The processes of the operating system task force communicated with each other through mailboxes. The body of the operating system was distributed

over the cluster, with system processes assigned to processors by the scheduler. Each cluster was controlled by a functionally complete operating system. Dynamic reconfiguration of the StarOS task force allowed the system to recover from hardware or software failures. If the hardware failed, the system processes would be recreated on functional equipment, and if software failed the faulty process would be destroyed and a new copy executed. This system was still under development and test as of 1980.

6.5.3. Current Activity

The Cambridge Model Distributed System (CMDS), from the University of Cambridge, took a different approach to network software [Wilk80]. CMDS is the network operating system for the Cambridge Digital Communications Ring shown in Figure 6-2. CMDS provides interactive timesharing facilities to its users via the network. The system is made up of servers and computational machines. The servers are microcomputers that implement a specific operating system function for all users; the computational machines are minicomputers that can be allocated to a user upon request. Communication is accomplished by sending messages on the ring, with each module checking all message headers for messages sent to it.

The operating system is physically as well as logically distributed. The initial configuration has six servers providing basic services. The Name Server maps the symbolic names for processes, servers and users to the ring addresses used in the message headers; users are never aware of the ring addresses. The File Server is responsible for handling all system and user file requests, and is the only processor with access to the system disks. The Printing Server spools user output to the line printer, and the Time Server provides time and date stamps for other servers. Finally, the Boot Server is responsible for the initially loading the ring interfaces.

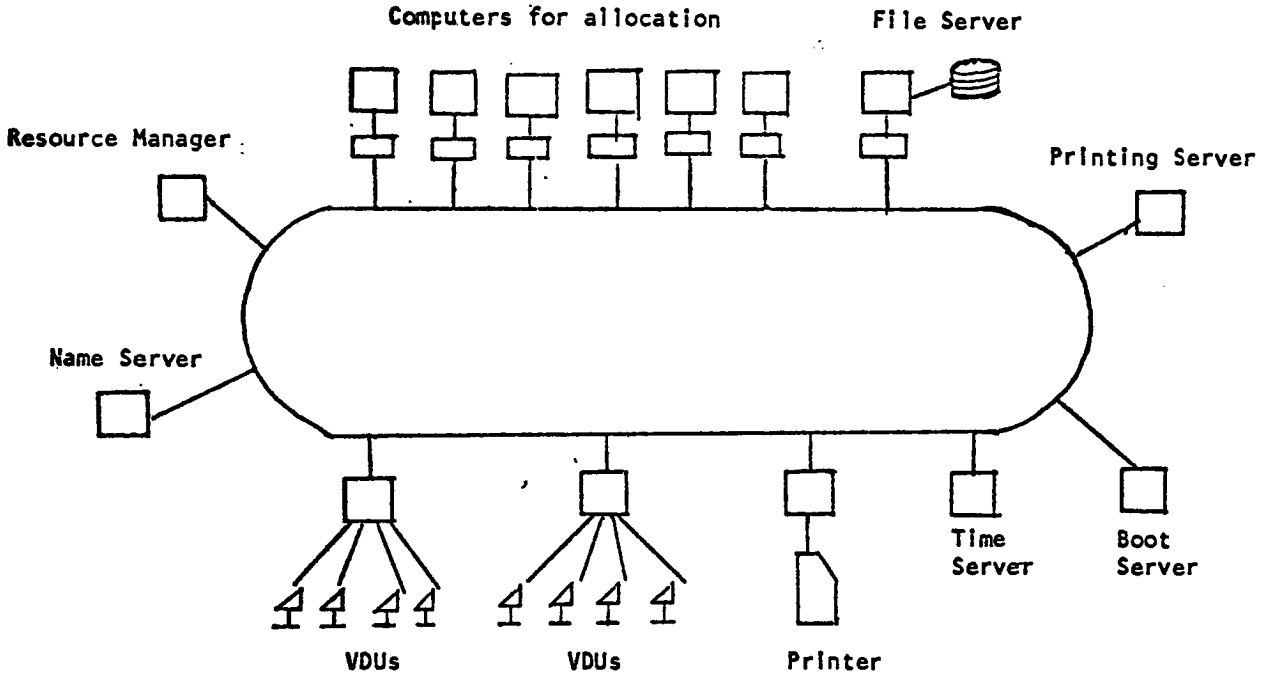


Figure 6-2
Cambridge Model Distributed Ring

The most critical portion of the system is the Resource Manager. Through it users are granted access to the network. The user terminals are connected to concentrators that in turn are attached to the ring; these concentrators send user authentication information to the Resource Manager. Once a user's identity is verified, he can use the network as an interactive timesharing system. When a user wants to run a program, he requests one of the six computational machines; the Resource Manager arbitrates these requests. The Resource Manager can reboot the computational machines if user software running on them fails irrecoverably.

Users are not allowed indiscriminate use of all resources on the network. A user's privileges are maintained by the Resource Manager, who then grants access to other servers if permitted. Other protection features are implemented in the message handling software. Each ring station maintains a list of stations from which it will accept messages. In this way, a station can ignore messages it does not expect. The acceptance list for each station is maintained with the aid of the Resource Manager.

The server approach of CMDS has advantages and disadvantages. One of the advantages is flexibility in choosing the server functions. Any commonly used program, such as an editor, can be implemented as a server. Once this is done, editing requests will be handled by this server. Without the editor server, editing requests would generate requests to the Resource Manager to allocate a computational machine, load the editor into the memory, and run it. While this is transparent to the users, it ties up computational resources and has a long set up time.

Another advantage is that if a bottleneck develops, another copy of the server can be added to the system. This can be done temporarily by commandeering a computational machine, or permanently by adding another station. The latter is a feasible approach, because the stations are inexpensive microprocessors.

The primary disadvantage to the approach is that critical facilities are located in single stations. If the Resource Manager or Name Server fails, the entire network fails. To alleviate this problem, CMDS permitted redundant critical servers. This system is still in early stages of development, and research with it continues.

While CMDS represents one approach to the problems of network operating system, many other techniques are being investigated. A group at MIT turned away from the classical concepts of computer organization in their network operating system, TRIX [Ward80]. They believed that the traditional view of memory as the primary commodity of computers was not appropriate for implementing distributed networks.

The basic commodity in TRIX is a stream. A stream is a full duplex path for passing control or data messages between processes. Stream communications are asynchronous, so a process can operate several streams at a time. All system objects, such as files and devices, are implemented as processes attached to streams. For example, a file appears to be a stream that responds to control messages such as read, write and seek. The message semantics are associated with the stream, rather than the mechanism at the end of the stream that implements them. Programs use streams without regard to the underlying implementation.

An advantage of streams is their flexibility. For example, if new protection services are required, processes to perform protection protection checks can be inserted into the path to the protected object. If data translation services are needed, the translation processes can be placed in the path between the dissimilar objects. Users are unaware of such changes.

Since streams separate the abstract service or object from its implementation, it does not matter where an object or service resides. While the multiple

processor aspects of TRIX have yet to be tested, the MIT researchers expect the organizational structure will lend itself to network implementation.

The LOCUS system at UCLA is a recently developed network operating system [Pope81]. LOCUS is built on a local area network connecting PDP-11s on a high speed ring. The objective is to provide a highly reliable system giving the illusion of a single machine.

Each computer in LOCUS is a functionally complete system; each runs the same software, configured to account for hardware differences. In this way, if the network is divided into disjoint pieces by a failure, the individual components can still function. The underlying philosophy of the system is that each site is master of its own resources, so it cannot be swamped by network requests.

One of the key components of LOCUS is the implementation of a distributed global file system. The system provides a hierarchical name space compatible with that of Unix. File objects in the system have a globally unique name in the hierarchy. For example, pathnames /group1/user1/file1 and /group1/user2/file1 define two different files, each of which appear to the owner as file1. The path associated with user 1 is /group1/user1/ and the path associated with user 2 is /group1/user2/ so there is no conflict.

The file system is logically partitioned into file groups. For each file group, one site is arbitrarily designated the Current Synchronization Site (CSS), and is responsible for coordinating access to all files in the group. As there may be several copies of a file, the CSS is responsible for locating the latest version of the file. A Storage Site (SS) is any site that has a copy of the file. For each new file request, the CSS chooses an SS to handle the request. Each user works with one copy of a file. Changes to the file are not made known to other users

until the file is closed; this guarantees file consistency to each user. When a file is closed a copy of it is distributed to all SSs with a copy. However, conflicts can occur if a file is updated at the same time by two users. When the file is closed, and the updates are distributed, there is problem regarding how to merge the changes, or how to decide which copy will overwrite the other. Resolution of such conflicts is still an open question.

Network efficiency is a primary goal of LOCUS. The penalty for non-local access must be low so that processor without local file storage can be added. What is more, local file access should not cost more because of the network. This latter goal is met by bypassing the network when the using site, SS and CSS are the same. The performance measurements for file accesses show that LOCUS has approximately the same throughput as standard Unix when operating on local files. For distributed files, access time is slightly longer than Unix for large block transfers, and about twice as slow for single block transfers. These results support the concept of using distributed processing for timesharing applications.

Distributed computing and its affect on operating systems is still a lively, current topic. This will probably continue for many years, because the issues are only partially understood, and they encompass so many areas of computer science.

6.6. Contributions of the Era

It is difficult to identify the contributions of this era, as it still in its early stages of development. What will be described are techniques that have been developed (but not fully tested), and issues that have yet to be resolved.

6.6.1. Hardware Refinements

Most of the hardware developments for distributed computing involved communications equipment not related to operating systems: they will not be presented here. Only those changes that influenced or were influenced by operating systems design will be discussed.

Memory switches were developed to facilitate high speed methods of communications among closely coupled processors. Multilevel switching mechanisms like those found in Cm* are available, as well as simple multiport memories, that allow multiple processors to access a single bank of memory locations in a controlled way.

Various methods of connecting the processors were developed to accommodate different approaches to network software. The earliest distributed network, ARPANET, used common carrier facilities available through the phone company. These facilities were slow and subject to data transmission errors. Such low bandwidth would not be acceptable in a system like the Cambridge Ring, which used the network to implement an interactive timesharing facility. Local networks require higher speed, more reliable communications links.

The need to reduce operating system overhead associated with communications protocols led to the development of intelligent communications devices. There are many network interfaces that will handle the physical and data link functions: detection of errors, requests for retransmission, and the formatting and ordering of message packets. These reduce the overhead associated with standard network operations.

6.6.2. Software Refinements

Network operating systems are still in their infancy. Unlike many other areas in operating systems, little has been firmly established. However, there are a variety of approaches to network operating systems, each with its own set of advantages and disadvantages. These will be presented here.

Networks can be classified as loosely coupled or closely coupled. In a loosely coupled system the nodes are independent units, but they can cooperate to achieve larger goals. The user is aware that the network exists, and has facilities available to make the network work for him. The primary design questions in a loosely coupled system are the allowable levels of sharing, and the degree to which users are shielded from the host specific features.

A closely coupled system is one in which the nodes normally work together to achieve a common goal. The users are a single community to be serviced, rather distinct groups associated with specific machines. In a closely coupled system, there are many policy choices regarding the relationship among the processors. The operating system can be server oriented, with certain functions residing only on different machines. Alternatively, the system can be controlled by a master, with other processors acting as slaves; or all processors can be equals. These are some examples of the issues in the design of a closely coupled system.

Network operating systems can be broadly classified as guest and non-guest systems. A non-guest system runs directly on the hardware with no intermediary software. A guest coexists with a host operating system, working with it to provide network services.

Guests are developed to preserve the investment in existing host operating systems and the associated software, while allowing a network to be built. The

problems associated with designing a guest relate to interactions with existing software. The host operating systems will have their own file systems, command languages, protection mechanisms, and internal structures; these can vary dramatically. The network designers are constrained by the limitations imposed by the hosts in the network. Most guest systems to date either limit the range of capabilities over different hosts, or they are operate on a single type of host.

In any network implementation, the effects of the number of different computer types must be considered. Homogeneous networks are easier to design and implement, as there are fewer variables involved. However, they are not practical in many instances, since the machines to be included in the network already exist. Heterogeneous systems present more difficulties when the hardware or software architectures differ significantly. Fundamental problems can occur, that are not easily resolvable, for example inadequate hardware support for the protection mechanisms. One method used to get around some of the problems presented by heterogeneous systems is to implement the network functions on a front-end processor that acts as an intermediary for the host on the network.

A difficult issue in distributed software involves the treatment of files. General questions arise regarding the types of accesses allowed on remotely stored files. In a network wide file system, synchronization is a key problem if duplicate copies of files are kept. Conflicts can occur when updating duplicate files, and problems resulting from partitioning the network need to be resolved. Some of these problems stem from a more basic problem caused by the time delays inherent in distributed systems. The problem is that it is not possible for any processor to know the state of the entire network at the time it has to make decisions. The state of the system could have changed between the time a processor checked on the status of some resource and the time it allocates that resource to a user.

7. Summary

Computer science is a very young field. In the 30 years that computers have been in existence, the field has changed dramatically. Because of this it is difficult to follow all the developments, and to correctly identify the factors influencing them. This is especially true in the recent years, because the effects of the research are as yet unknown.

What has been presented is a selective history of the development of one area of computer science: operating systems. The topics discussed represent major themes that pervaded the professional literature of the time. These topics have been identified as multiprogramming, virtual memory, security and protection, and distributed processing systems. Multiprogramming systems were a major research area from the late 1950's through the late 1960s. Virtual memory systems dominated the late 1960s through the middle 1970s. Security and protection, and distributed systems whose origins are in the 1960s and 1970s, are still being researched, and will be for many years to come. Figure 7-1 summarizes the timing of the major developments for each subject area.

There are major subjects which have contributed to operating system design that have been knowingly omitted. Some of these are the development of file systems, synchronization and program methodology. While these would require discussion in an exhaustive history of operating systems, they are beyond the scope of this thesis.

Year	Early	Multiprogramming	Virtual Memory	Security	Distributed
1950- 1955 1956 1957 1958 1959	Assemblers symbolic addressing simple supervisors interrupts batch systems	 GAMMA-60 TX-2 Strachey	 overlays		 multiprocessors
1960 1961 1962 1963 1964 1965 1966 1967 1968 1969		 Atlas H800 CTSS TSS THE	 Atlas B5000 Multics W.S. Model Belady	 Atlas CTSS Multics Object Model Lampson	
1970 1971 1972 1973 1974 1975 1976 1977 1978 1979			Denning	Virt. Mach. Security kernels UCLA-VM MITRE Unix security KSOS PSOS	ARPANET RSEXEC C.mmp NSW XNOS Cm*
1980 1981					CMDS Trix Locus

Figure 7-1
Summary of Major Developments

Annotated Bibliography

This bibliography is arranged by chapter.

Annotated Bibliography

Early History

1. F.P. Books; "A Program Controlled Program Interruption System"; Proceedings of the Eastern Joint Computer Conference, 1957; Institute of Radio Engineers

   A description of the IBM Stretch computer system. The interrupt system is described.

2. Willard G. Bouricius; "Operating Experience with the Los Alamos 701"; Proceedings of the Eastern Joint Computer Conference, 1953; Institute of Radio Engineers

   An article describing early experiences with common libraries.

3. J.H. Brown, et al.; "Prevention of Machine Errors in Long Problems"; Journal of the ACM; Vol. 3, Oct. 56

   This article describes the Large Program System (LPS) for the MIDAC computer that performs run time error checking and environment preservation in the event of machine checks. The major motivation behind this project was the poor machine reliability that lead to wasted machine time in job restarts.

4. Wesley Clark; "Lincoln TX-2 Computer Development"; Proceedings of the Western Joint Computer Conference, 1957; Institute of Radio Engineers

   A hardware description of the TX-2 multiprogramming computer system.

5. S.W. Dunwell; "Design Objectives for the IBM Stretch Computer"; Proceedings of the Eastern Joint Computer Conference, 1956; American Institute of Electrical Engineers

   An announcement article describing the preliminary design of the Stretch system. Primarily hardware oriented, although an automatic programming system is alluded to.

6. J.P. Eckert; "Univac-LARC, The Next Step in Computer Design"; Proceedings of the Eastern Joint Computer Conference, 1956; American Institute of Electrical Engineers

   An announcement article describing the hardware features of the system.

7. Jules Mersel; "Program Interrupt on The Univac Scientific Computer"; Proceedings of the Western Joint Computer Conference, 1956; American Institute of Electrical Engineers

A very interesting article describing the first use of interrupts. The interrupt capability was requested by a user, but the author wasn't sure if they would be generally useful or not.

8. Bruse Moncreiff; "An Automatic Supervisor For The IBM 702"; Proceedings of the Western Joint Computer Conference, 1956; American Institute of Electrical Engineers

Description of a supervisory program written to facilitate job set up and tape handling. An example of the types of systems put together by early users.

9. Robert F. Rosin; "Supervisor and Monitor Systems"; Computing Surveys; Vol. 1, No. 1, March 1969

An excellent paper giving a history of supervisor software from the days of manual scheduling through to early timesharing systems. It gives the motivations and some of the hardware developments that pushed the software movement.

10. W.F. Schmitt and A.B. Tonik; "Sympathetically Programmed Computers"; Information Processing. Proceedings of the International Conference on Information Processing, 1959; UNESCO

A primarily hardware description of the Univac-LARC system, that described in fair detail the two computer subsystems and their intercommunication. An I/O supervisory program is proposed to handle the I/O computer and to do polling (faster than interrupts) to see if there is any device needing attention.

11. Donald L. Shell, et al.; "The SHARE 709 System"; Journal of the ACM; Vol. 4, 1962

A series of articles on the SHARE project of the late 1950s for the IBM 709 computer.

12. J. Svigals; "IBM 7070 Data Processing System"; Proceedings of the Western Joint Computer Conference, 1959; Institute of Radio Engineers

    The hardware description of this system is not very exciting, especially after reading about the other IBM 700's. This article does described briefly the software supervisor (of sorts) that is provided with the system.

1. Charles W. Adams; "Trends in Design of Large Computer Systems"; Proceedings of the Western Joint Computer Conference, 1961; Western Joint Computer Conference

This provides a general overview of hardware, and some software, trends of the period. He mention some representative computers. Overall, this is not a very exciting article if you've read much else from the period.

2. James P. Anderson, et al.; "D825 - A Multiple Computer System for Command and Control"; Proceedings of the Fall Joint Computer Conference, 1962; Spartan Books

An article introducing the hardware and operating system (AOSP) of the Burroughs D825 computer. This article is of interest in that this is a fairly functional operating system that was commercially available.

3. Walter F. Bauer; "Horizons in Computer System Design"; Proceedings of the Western Joint Computer Conference, 1960; Western Joint Computer Conference

This article describes a model of computer control hierarchy that is similar to that used in later operating systems. He also discusses the use of microprogramming as a method of tuning different computers to specific application areas.

4. G.A. Blaauw, et al.; "The Structure of SYSTEM/360"; IBM Systems Journal; Vol. 3, No. 2, 1964

This article is primarily on the hardware design of the System 360 but there are bits and pieces regarding the software design.

5. Charles P. Bourne and Donald F. Ford; "The Historical Development and Predicted State-of-the-Art of the General Purpose Digital Computer"; Proceedings of the Western Joint Computer Conference, 1960; Western Joint Computer Conference

This is an interesting study of architectural trends in computer design, with an initial directory of the world's computers at the time. This is more appropriate to a study of computer architecture than software.

6. Wesley Clark; "Lincoln TX-2 Computer Development"; Proceedings of the Western Joint Computer Conference, 1957; Institute of Radio Engineers

A hardware description of the TX-2 multiprogramming computer system.

7. John Cocke and Hardwood G. Kosky; "Virtual Memory in the Stretch Computer"; Proceedings of the Eastern Joint Computer Conference, 1959; Eastern Joint Computer Conference

The meaning of "virtual memory" has changed over time, and this article is about the instruction prefetch and pipelined instruction capability of the IBM Stretch computer, not virtual memory as we now think of it.

8. E.F. Codd; "Multiprogram Scheduling: Introduction and Theory"; Communications of the ACM; Vol. 3, No. 6, June 1960

An early paper on scheduling considerations. There is little real meat in this article.

9. E.G. Coffman and Leonard Kleinrock; "Computer Scheduling Methods and Their Countermeasures"; Proceedings of the Spring Joint Computer Conference, 1968; Thompson Book Co.

This is a fairly complete survey article on scheduling techniques for either batch or timesharing systems, and how to get around them. Good references.

10. Fernardo J. Corbato, et al.; "An Experimental Time-Sharing System"; Proceedings of the Spring Joint Computer Conference, 1962; National Press

This is a classic article describing in detail the experimental version of Project Mac's CTSS system. This is the single most referenced article in this area.

11. F.J. Corbato and J.H. Saltzer; "Some Considerations of Supervisor Program Design For Multiplexed Computer Systems"; Proceedings of the IFIP Congress, 1968; North-Holland Publishing Coo.

The differences between multiplexing and sharing programs are discussed in this paper, with regard to how these properties affect supervisory programs.

12. P.A. Crisman; The Compatible Time-Sharing System: A Programmer's Guide; MIT Press; 1965

This is the user's guide to the CTSS system. As such it concentrates on how to use it, rather than on the system design. The prefaces to the 2 editions by F.J. Corbato are interesting.

13. A.J. Critchlow; "Generalized Multiprocessing and Multiprogramming Systems"; Proceedings of the Fall Joint Computer Conference, 1963; Spartan Books

For the most part this article is so generalized that it is of limited use. It does have good references however.

14. Peter J. Denning; "Effects of Scheduling on File Memory Operations"; Proceedings of the Western Joint Computer Conference, 1967; Spartan Books

This article discusses the advantages and disadvantages of several disk and drum scheduling algorithm.

15. Edsger W. Dijkstra; "The Structure of THE Multiprogramming System"; Communications of the ACM; Vol. 11, No. 5, May 68

A classic paper describing an operating system composed of simultaneous sequential programs that cooperate via explicit mutual exclusion statements. Unfortunately there are no references.

16. Phillippe Dreyfus; "System Design of the GAMMA-60"; Proceedings of the Western Joint Computer Conference, 1957; Institute of Radio Engineers"

A description of France's GAMMA-60 multiprogramming system. One of the first systems with a vendor supplied supervisor to help handle multiprogramming.

17. S.W. Dunwell; "Design Objectives for the IBM Stretch Computer"; Proceedings of the Eastern Joint Computer Conference, 1956; American Institute of Electrical Engineers

An announcement article describing the preliminary design of the Stretch system. Primarily hardware oriented, although an automatic programming system is alluded to.

18. J.P. Eckert; "Univac-LARC, The Next Step in Computer Design"; Proceedings of the Eastern Joint Computer Conference, 1956; American Institute of Electrical Engineers

An announcement article describing the hardware features of the system.

19. J. Forgie; "A Time- and Memory-sharing Executive Program for Quick Response, On-Line Applications"; Proceedings of the Fall Joint Computer Conference, 1965; Thompson Book Co.

A description of the updated TX-2 computer and the APEX executive system. No startling new advances.

20. Charles T. Gibson; "Timesharing in the IBM System/360 Model 67"; Proceedings of the Western Joint Computer Conference, 1966; Spartan Books

A short hardware and software description of the Model 67, that was designed to be a multiprocessor, multiprogramming, multiaccess system. The author would have you believe that IBM invented all these ideas.

21. A.C.D. Haley; "The KDF.9 Computer System"; Proceedings of the Fall Joint Computer Conference, 1962; Spartan Books

This is primarily a hardware description of English Electric's KDF.9 computer.

22. C.A.R. Hoare; "Communicating Sequential Processes"; Communications of the ACM; Vol. 21, No. 8, Aug. 78

This article proposes a language, similar to that developed by Dijkstra, that supports concurrent processes. No implementation details are dealt with. Not directly applicable.

23. John H. Howard; "Mixed Solutions For the Deadlock Problem"; Communications of the ACM; Vol. 16, No. 7, July 73

This article gives a quick overview of the deadlock problem and how to apply the principles of detection, avoidance and protection to a hierarchical system. There is no new material in this article.

24. D.J. Howarth, et al.; "The Atlas Scheduling System"; Proceedings of the Spring Joint Computer Conference, 1963; Spartan Books

This is a fairly detailed article describing the Manchester University Atlas system.

25. Butler W. Lampson and Howard E. Sturgis; "Reflections on an Operating System Design"; Communications of the ACM; Vol. 19, No. 5, May 76

The article is a reflection on an unsuccessful operating system project for a CDC 6400. The article is interesting, in that the operating system was of an unusual design, but it is not of a general enough nature to be very useful.

26. A.L. Leiner, et. al; "Concurrently Operating Computer Systems"; Information Processing. Proceedings of the International Conference on Information Processing; UNESCO

A hardware description of the PILOT computer developed for the National Bureau of Standards. This system had 3 computer based subsystems and was designed to handle a primitive form of multiprogramming. Not much emphasis is placed on software to help the poor programmer.

27. J.C.R. Licklider and Welden E. Clark; "On-Line Man-Computer Communications"; Proceedings of the Spring Joint Computer Conference, 1962; National Press

The introduction of this article discusses the justification for on-line processing, but the rest of the article deals mostly with CAI applications.

28. N. Lourie, et. al; "Arithmetic and Control Techniques in a Multiprogram Computer"; Proceedings of the Eastern Joint Computer Conference, 1959; Eastern Joint Computer Conference

The article discusses the features of the Honeywell 800 which has implemented a simple multiprogramming system in hardware, "without the use of cumbersome supervisory routines".

29. W.C. McGee; "On Dynamic Program Relocation"; IBM Systems Journal; Vol. 4, No. 3

Discussion of dynamic program relocation using various methods, with examples from ATLAS, IBM System/360, B5000.

30. G.H. Mealy; "The Functional Structure of OS/360: Introductory Survey"; IBM Systems Journal; Vol. 5, No. 1, 1966

This article was a very general overview of the System 360, with little real information regarding the operating system design.

31. Ascher Opler; "Current Problems in Automatic Programming"; Proceedings of the Western Joint Computer Conference, 1961; Western Joint Computer Conference

This paper primarily describes compiler writing techniques, such as intermediate languages and compiler bootstrapping.

32. J.F. Ossanna, et al.; "Communications and Input/Output Switching in a Multiplex Computing System"; Proceedings of the Fall Joint Computer Conference, 1965; Thompson Book Co.

This article describes the hardware and some of the software aspects of the Multics I/O system.

33. Allen Reiter; "A Resource Allocation Scheme for Multi-User On-Line Operation of a Small Computer"; Proceedings of the Spring Joint Computer Conference, 1967; Spartan Books

This article discusses the development of a timesharing operating system for an information retrieval system. The design is primarily concerned with the I/O bound nature of the job load.

34. Saul Rosen; "Programming Systems and Languages 1965-1975"; Communications of the ACM; Vol. 15, No. 7, July 72

An excellent article giving a general history with some very notable examples. It also has an excellent list of references for anyone interested in the subject.

35. Saul Rosen; "Hardware Design Reflecting Software Requirements"; Proceedings of the Fall Joint Computer Conference, 1968; Thompson Book Co.

A historical perspective on hardware advances influenced by software technology. A good article, but limited in scope.

36. Robert F. Rosin; "Supervisory and Monitor Systems"; Computing Surveys; Vol. 1, No. 1, March 1969

An excellent paper giving a history of supervisory software from the days of manual scheduling through to early timesharing systems. It gives the motivations and some of the hardware developments that pushed the software movement.

37. George F. Ryckman; "The Computer Operation Language"; Proceedings of the Western Joint Computer Conference, 1960; Western Joint Computer Conference

This is one of the first articles promoting a modular approach to operating systems design, so that new languages and utility processors can be easily added. He proposes a machine independent operations language to achieve this goal, but does not give much detail.

38. B.L. Ryle; "Multiple Program Data Processing"; Communications of the ACM; Vol. 4, No. 2, Feb. 1961

The author discusses the requirements of a multiprogrammed system and compares current systems against his ideal.

39. H. Sackman; "Timesharing vs. Batch Processing - The Experimental Evidence"; Proceedings of the Spring Joint Computer Conference, 1968; Thompson Book Co.

The introduction of this article has some perspectives on the origins of timesharing systems, but the rest of the article is not very useful.

40. Jules I. Schwartz, et al.; "A General Purpose Time-Sharing System"; Proceedings of the Spring Joint Computer Conference, 1964; Spartan Books

This is an often referenced article describing the TSS operating system and its language processors. TSS is an early timesharing system for an IBM system.

41. Stephen Sherman, et al.; "Trace Driven Modeling and Analysis of CPU Scheduling in a Multiprogramming System"; Communications of the ACM; Vol 15, No. 12, Dec. 72

This article compares several scheduling techniques. Up until this time, most of the experimental work was based on dynamic predictors of activity.

42. Thomas B. Steele; "Multiprogramming - Promises, Performance, and Prospect"; Proceedings of the Fall Joint Computer Conference, 1968; Thompson Book Co.

This is an interesting historical perspective on multiprogramming, although the author is rather opinionated. It does have good references.

43. David F. Stevens; "On Overcoming High Priority Paralysis in Multiprogramming Systems: A Case History"; Communications of the ACM; Vol. 11, No. 8, Aug. 68

A short article on studies done on a CDC Chippewa system to maximize utilization. No startling results.

44. C. Strachey; "Timesharing in Large Fast Computers"; Information Processing. Proceedings of the International Conference on Information Processing; UNESCO

This article describes a very early timesharing supervisor that is implemented in ROM. The concept of privileged instructions and inter-process protection is proposed.

45. V.A. Vyssotsky, et al.; "Structure of the Multics Supervisor"; Proceedings of the Fall Joint Computer Conference, 1965; Thompson Book Co.

The name of the article is misleading, as this article is a general discussion of supervisor design, with some application to Multics. There is not enough detail to be useful.

46. B.I. Witt; "The Functional Structure of OS/360: Job and Task Management"; IBM Systems Journal; Vol. 5, No. 1, 1966

Another meatless article on OS/360 by IBM.

47. Mildred Wilkerson; "The Jovial Checker - An Automatic Checkout System for Higher Level Language Programs"; Proceedings of the Western Joint Computer Conference, 1961; Western Joint Computer Conference

    This paper discuss a real debugger for the Jovial language for an IBM 7090.

Annotated Bibliography

Virtual Memory Systems

1. Alfred V. Aho, et al.; "Principles of Optimal Page Replacement"; Journal of the ACM; Vol. 18, No. 1, Jan. 1971

   A very formal papaer on modeling paging algorithms.

2. B.W. Arden, et al.; "Program and Addressing Structure in a Time Sharing Environment"; Journal of the ACM; Vol. 13, No. 1, Jan. 1966

   This article discusses problems and solutions relating to program relocation, including a lengthy discussion of paging strategies.

3. V.A. Belady; "A Study of Replacement Algorithms for a Virtual Storage Computer"; IBM Systems Journal; Vol. 5, No. 2, 1968

   A classic article in the area. Replacement algorithms are grouped into 3 classes and each class is evaluated against a theoretically optimal algorithm.

4. A. Bensoussan, et al.; "The Multics Virtual Memory: Concepts and Design"; Communications of the ACM; Vol. 15, No. 5, May 1972

   This article describes in detail the segmentation and paging operations of the Multics system; by the time this paper was done, Multics had been running for 5+ years.

5. D.D. Chamberlin, S.H. Fuller, L.Y. Liu; "Analysis of Page Allocation Strategies for Multiprogramming Systems with Virtual memory"; IBM Journal of Research and Development; Vol. 17, No. 5

   A semi-interesting article, although not skimmable for salient points.

6. E.G. Coffman and L.C. Varian; "Further Experimental Data on the Behavior of Programs in a Paging Environment"; Communications of the ACM; Vol. 11, No. 7, July 68

   This is a moderately interesting article with no startling conclusions.

7. F.J. Corbato and J.H. Saltzer; "Some Considerations of Supervisor Program Design for Multiplexed Computer Systems"; Proceedings of the Internation Conference on Information Processing; , ; 1968".

The article discusses some complexities of operating system design due to multiplexing and sharing; the use of segmentation and paging is discussed as potential techniques to minimize these complexities.

8. Robert C. Daley and Jack B. Dennis; "Virtual Memory, Process, and Sharing in Multics"; Communications of the ACM; Vol. 11, No. 5, May 68

The article describes the addressing scheme that allows for the ability to easily share code on Multics. The article is quite detailed and interesting.

9. Peter J. Denning; "Thrashing: Its Causes and Prevention"; Proceedings of the Fall Joint Computer Conference, 1968; Thompson Books Co.

One of the first of Denning's working set model articles.

10. Peter J. Denning; "The Working Set Model for Program Behavior"; Communications of the ACM; Vol. 11, No. 5, May 68

A very good, classic paper describing the definition and usage of the working set concept for handling memory allocation in a virtual memory system.

11. Peter J. Denning; "Virtual Memory"; Computing Surveys; Vol. 2, No. 3, Sept. 70

This paper is a survey of virtual memory principles tracing the development of the various techniques. It is an excellent, very detailed article of the subject with many references.

12. Jack B. Dennis and Earl Van Horn; "Programming Semantics for Multiprogrammed Computation"; Communications of the ACM; Vol. 9, No. 3, Mar. 1966

This is a classic article in which the authors describe capability lists as a means of access control.

13. Charles T. Gibson; "Timesharing in the IBM System/360 Model 67"; Proceedings of the Spring Joint Computer Conference, 1966; Spartan Books

    The earliest IBM system with virtual memory capability. The author would have you believe IBM invented it and multiprogramming.

14. D.J. Howarth, et al.; "The Atlas Scheduling System"; Proceedings of the Spring Joint Computer Conference, 1963; Spartan Books

    This is a fairly detailed article describing the Manchester University Atlas system.

15. T. Kilburn, et al.; "The Manchester University Atlas Operating System"; Computer Journal; Vol. 4, 1961

    This article describes the I/O wells, how they are maintained and used, for the Atlas system. This is the least detailed and unique of the Atlas papers.

16. T. Kilburn, et al.; "The Atlas Supervisor"; EJCC, 1961; MacMillan

    A very complete article describing the structure of the Atlas system and its virtual memory system; it is must reading for anybody interested in computing history or virtual memory.

17. T. Kilburn, et al.; "One Level Storage System"; IRE Transactions on Electronic Computers; Vol. 11, No. 2, April, 1962

    A classic article on the Atlas virtual memory system.

18. C.J. Keuhner and B. Randell; "Demand Paging in Perspective"; Proceedings of the Fall Joint Computer Conference, 1968; Spartan Books

    No enlightening discoveries or conclusions in this article.

19. Alexander Lett and William Konigsford; "TSS/360: A Time-Shared Operating System"; Proceedings of the Fall Joint Computer Conference, 1968; Thompson Books

    A very general discussion of timesharing systems and IBM's TSS operating system in particular.

20. William Lonegran and Paul King; "Design of the B5000 System"; Datamation; May, 1961

A brief description of the overall goals and design of the Burroughs B5000. This is an early virtual memory system that is largely overshadowed by Atlas.

21. W.C. McGee; "On Dynamic Program Relocation"; IBM Systems Journal; Vol. 4, No. 3, 1965

The author discusses methods of program relocation using segments, and uses several systems as examples.

22. R.A. Meyer and L.H. Seawright; "A Virtual Machine Time Sharing System"; IBM Systems Journal; Vol. 9, No. 3, 1970

The article gives some details of the IBM CP-67/CMS system for the 360/67, but it looks like a plug for the product, which sounds awful.

23. Elliott I. Organick; The Multics System: An Examination of Its Structure; MIT Press; 1972

This book gives an overall description of the Multics system. Unfortunately it does not give a detailed description of the implementation of the facilities presented. It is designed to provide an overview of the design to curious users.

24. J.F. Ossanna, et al.; "Communications and Input/Output Switching in a Multiplex Computing System"; Proceedings of the Fall Joint Computer Conference, 1965; Thompson Book Co.

This article describes the hardware and some of the software aspects of the Multics I/O system.

25. Saul Rosen; "Programming Systems and Languages 1965-1975"; Communications of the ACM; Vol. 15, No. 7, July 1972

An excellent article giving a general history, with some very notable examples. it also has an excellent list of references for anyone interested in the subject.

26. V.A. Vyssotsky, et al.; "Structure of the Multics Supervisor"; Proceedings of the Fall Joint Computer Conference, 1965; Thompson Book Co.

    The name of the article is misleading, as this article is a general discussion of supervisor design, with some application to Multics. There is not enough detail to be useful.

Annotated Bibliography

Security and Protection Systems

1. Peter S. Browne; "Computer Security - A Survey"; Proceedings of the National Computer Conference, 1976; AFIPS Press

   The author provides a brief overview of the subject with an excellent annotated bibliography.

2. J.M. Carroll and P.M. McLelland; "Fast 'Infinite-Key' Privacy Transformations for Resource-Sharing Systems"; Proceedings of the Fall Joint Computer Conference, 1970; AFIPS Press

   This article was only moderately interesting; the only new part was their 2 phase random number generator used to build the cipher key.

3. P.A. Crisman; The Compatible Time-Sharing System: A Programmer's Guide; MIT Press; 1965

   This is the user's guide to the CTSS system. As such it concentrates on how to use it, rather than on the system design. The prefaces to the 2 editions by F.J. Corbato are interesting.

4. F.J. Corbato, et al.; "An Experimental Time-Sharing System"; Proceedings of the Spring Joint Computer Conference, 1962; National Press

   This article describes the experimental version of MIT's CTSS system. It is mostly about the multiprogramming aspects of the system.

5. Jack B. Dennis and Earl Van Horn; "Programming Semantics for Multiprogrammed Computation"; Communications of the ACM; Vol. 9, No. 3, Mar. 1966

   This is a classic article in which the authors describe capability lists as a means of access control.

6. Richard J. Feiertag and Peter G. Neumann; "The Foundation of a Provably Secure Operating System"; Proceedings of the National Computer Conference, 1979; AFIPS Press

   The article describes the advantages and disadvantages of security kernel designs based on the authors' experiences with PSOS.

7. R. Stockton Gaines; "An Operating System Based on the Concept of a Supervisory Computer"; Communications of the ACM; Vol. 15, No. 3, Mar 1972

An article describing an early reference monitor.

8. Robert M. Graham; "Protection in an Information Processing Utility"; Communications of the ACM; Vol. 11, No. 5, May 68

Good overview of the protection policies and mechanisms of the earlier Multics system.

9. Per Brinch Hansen; "The Nucleus of a Multiprogramming System"; Communications of the ACM; Vol. 13, No. 4, April 1970

This is a classic article briefly describing the RC4000 system's nucleus.

10. Lance J. Hoffman; "Computers And Privacy: A Survey"; Computing Surveys; Vol. 1, No. 2, June 1969

An old overview article of the problems of computer security. The article itself is not very enlightening, but it does have an extensive annotated bibliography.

11. A.K. Jones; "Protection Mechanisms and Enforcement of Security Policies"; Operating Systems: An Advanced Course; R. Bayer (ed); Springer-Verlag; 1978

A good overview of security policies and mechanisms, and how they can be implemented using the object model.

12. A.K. Jones; "The Object Model: A Conceptual Tool for Structuring Software"; Operating Systems: An Advanced Course; R. Bayer (ed); Springer-Verlag; 1978

An introduction to the object model and its use in operating system design.

13. T. Kilburn, et al.; "The Atlas Supervisor"; Proceedings of the Eastern Joint Computer Conference, 1961; MacMillan

A very complete article describing the structure of the Atlas system and its virtual memory system; it is must reading for anybody interested in computing history or virtual memory.

14. Butler W. Lampson; "Dynamic Protection Structures"; Proceedings of the Fall Joint Computer Conference, 1969; AFIPS Press

This article describes the Berkeley Computer Corp. Model I operating system whose protection is based on capabilities.

15. Butler W. Lampson; "Protection"; Proceedings of the 5th Annual Princeton Conference, 1971; Princeton

This article discusses protection is general and how various techniques have been discussed to implement protection mechanisms.

16. Richard R. Linde; "Operating System Penetration"; Proceedings of the National Computer Conference, 1975; AFIPS Press

This article is based on a study of generic operating system security flaws and a method of identifying them.

17. Steven B. Lipner; "A Minicomputer Security Control System"; Compcon, 1974; IEEE

This paper presents MITRE's front-end approach to security control. It presents a very different approach to the problem that has not been adopted.

18. E.J. McCauley and P.J. Drongowski; "KSOS - The Design of a Secure Operating System"; Proceedings of the National Computer Conference, 1979; AFIPS Press

A description of a security kernel for a Unix look alike; it is an object based system, with network capabilities.

19. R.A. Meyer and L.H. Seawright; "A Virtual Machine Time Sharing System"; IBM Systems Journal; Vol. 9, No. 3, 1970

The article gives some details of the IBM CP-67/CMS system for the 360/67, but it looks like a plug for the product, which sounds awful.

20. Lee M. Molho; "Hardware Aspects of Secure Computing"; Proceedings of the Spring Joint Computer Conference, 1970; AFIPS Press

A disappointing article that would have been more aptly titled "How to detect hardware failures to effect secure computing."

21. R.M. Needham; "Protection Systems and Protection Implementations"; Proceedings of the Fall Joint Computer Conference, 1972; AFIPS Press

This article discusses protection as it applies to central memory only. The author presents pros and cons of several techniques, and builds a method of his own based on indirection.

22. Peter G. Neumann; "Computer System Security Evaluation"; Proceedings of the National Computer Conference, 1978; AFIPS Press

This article outlines the categories of security design flaws and factors influencing good security. It includes an evaluation of PSOS, Multics and Unix as they relate to these flaws

23. Elliott Organick; The Multics System: An Examination of Its Structure; MIT Press; 1972

This book describes the Multics system in many aspects, including a chapter on the access mechanisms and protection rings.

24. H.E. Petersen and R. Turn; "System Implications of Information Privacy"; Proceedings of the Spring Joint Computer Conference, 1967; Thompson Book Co.

A rather simplistic approach to privacy is presented; He does present many forms of security problems.

25. Gerard J. Popek and Charles S. Kline; "Verifiable Secure Operating System Software"; Proceedings of the National Computer Conference, 1974; AFIPS Press

This is a detailed article on the UCLA security kernel and its verifiability.

26. Gerald J. Popek and Charles S. Kline; "Issues in Kernel Design"; Proceedings of the National Computer Conference, 1978; AFIPS Press

An excellent article providing an overview of the constraints and principles of security kernel design.

27. Gerald J. Popek, et al.; "UCLA Secure Unix"; Proceedings of the National Computer Conference, 1979; AFIPS Press

This article describes the security kernel that was finally implemented at UCLA. There are many details on the design of the kernel and its security objects, and comparisons to standard Unix.

28. Jerome H. Saltzer; "Protection and the Control of Information Sharing in Multics"; Communications of the ACM; Vol. 17, No. 7, July 74

This article discusses the design goals of the Multics protection system based on its state of development in 1973. The article is very general in nature; Graham's paper gives a better overview.

29. Jerome H. Saltzer and Michael D. Schroeder; "The Protection of Information in Computer Systems"; Proceedings of the IEEE; Vol. 63, No. 9, Sept. 75

An excellent tutorial on security problems and protection techniques, with an extensive bibliography.

30. C. Strachey; "Timesharing in Large Fast Computers"; Information Processing. Proceedings of the International Conference on Information Processing; UNESCO

This article describes a very early timesharing supervisor that is implemented in ROM. The concept of privileged instructions and inter-process protection is proposed.

31. W. Wulf, et al.; "Hydra: The Kernel of a Multiprocessor Operating System"; Communications of the ACM; Vol. 17, No. 6, June 74

    This article describes the design philosophy of the Hydra kernel. The article focuses on the concepts of objects and protection as they apply to Hydra. A good example is given that makes the concepts presented clear.

32. Charles R. Young; "A Security Policy for a Profile Oriented Operating System"; Proceedings of the National Computer Conference, 1981; AFIPS Press

    The article is a very general overview of concepts of a security policy to meet DoD standards. There is not a lot of meat to the article.

1. Stephen D. Crocker, et al.; "Function-Oriented Protocols for the ARPA Computer Network"; Proceedings of the Spring Joint Computer Conference, 1972; AFIPS Press

This article discusses the higher level protocols being developed for the ARPANET to perform user directed functions.

2. J.A. Devereaux; "An Application Oriented Multiprocessing System Control Program Features"; IBM Systems Journal; Vol. 06, No. 2 1967

"Third in a series of articles on the IBM 9020 multiprocessing system. Not much real information on the multiprocessing aspects. Not a very enlightening article.

3. Carla S. Ellis and P. Rajaram; "Issues in the Design of a Network-Wide File System"; University of Rochester; 1981

An interesting article on considerations in a network-wide file management system being developed at the U of R. Good references.

4. P.H. Enslow; "Multiprocessor Organization - A Survey"; Computing Surveys; Vol. 9, No. 1, March 1977

A survey of techniques to interconnect processors. Very little regarding software requirements.

5. John G. Fletcher and Richard W. Watson; "Service Support in a Network Operating System"; Compcon Spring 80; IEEE

This article described the overall structure of a network operating system based on the Octopus system of Lawrence Livermore Labs. In specific, the service support layer was described.

6. Per Brinch Hansen; "The Nucleus of a Multiprogramming System"; Communications of the ACM; Vol. 13, No. 4, April 1970

This is a classic article briefly describing the RC4000 system nucleus.

7. Anita K. Jones, et al.; "StarOS, A Multiprocessor Operating System for the Support of Task Forces"; Proceedings of the 9th Symposium on Operating System Principles; ACM, 1979

The authors discuss the design and implementation of StarOS, which is the network operating system for Carnegie Mellon's Cm* multiprocessor system.

8. Stephen R. Kimbleton and Richard Mandell; "A Perspective on Network Operating Systems"; Proceedings of the National Computer Conference, 1976; AFIPS Press

This article tackles the network operating system as a mediator among existing host operating systems. Lots of references.

9. Stephen R. Kimbleton, et al.; "Network Operating Systems - An Implementation Approach"; Proceedings of the National Computer Conference, 1978; AFIPS Press

An interesting article on a layer network operating system approach used by XNOS from the National Bureau of Standards.

10. David L. Mills; "An Overview of the Distributed Computer Network"; Proceedings of the National Computer Conference, 1976; AFIPS Press

This article describes the DCN project at the Univ. of Maryland. Very little appears to be discusses about the actual network capabilities.

11. Robert E. Millstein; "The National Software Works: A Distributed Processing System"; Proceedings of the 1977 Annual Conference of the ACM; ACM

The article discusses the design of NSW, which is a guest operating system for the ARPANET.

12. John K. Ousterhout, et al.; "Medusa: An Experiment in Distributed Operating System Structure"; Communications of the ACM; Vol. 23, No. 2, Feb. 1980

A long, detailed article on the structure of the second operating system for Cm*. This is a highly distributed operating system for a network of micros.

13. M.A. Padlipsky, et al.; "KSOS - Computer Network Applications"; Proceedings of the National Computer Conference, 1979; AFIPS Press

    An overview of network security problems and the use of KSOS, Kernel Secure Operating System within a network. It is a disappointing article.

14. Richard Peebles and Thomas Dopirak; "ADAPT: A Guest System"; Compcon Spring 80; IEEE

    This is a description of a network operating system being developed to run on top of a host operating system by Digital Equipment.

15. G.J. Popek, and C.S. Kline; "Design Issues for Secure Computer Networks"; Operating Systems: An Advanced Course; R. Bayer(ed); Springer-Verlag; 1978

    This paper discusses the security aspects of networks, and concentrates on the use of encryption to achieve security.

16. G. Popek, et al.; "UCLA Secure Unix"; Proceedings of the National Computer Conference, 1979; AFIPS Press

    This article describes the security kernel for Unix built at UCLA; there is some discussion of its potential use in a network.

17. G. Popek, et al.; "Locus: A Network Transparent, High Reliability Distributed System"; UCLA; 1981

    This is a very interesting article giving a different view of local network architecture from the software viewpoint. More details on the design would be helpful, but it does have some good references.

18. Ronald J. Price; "Multiprocessing Made Easy"; Proceedings of the National Computer Conference, 1978; AFIPS Press

    An introductory article discussing the use of monitors as a way to implement multiprocessor software. This is not a research paper.

19. David L. Retz; "Operating System Design Considerations of Packet Switch Environment"; Proceedings of the National Computer Conference, 1975; AFIPS

This article discusses features required in a host to append network communications over a network like ARPANET. Not very interesting.

20. Lawrence G. Roberts and Barry D. Wessler; "Computer Network Development to Achieve Resource Sharing"; Proceedings of the Spring Joint Computer Conference, 1970; AFIPS Press

This article was one of several introducing the ARPANET; it describes the motivations and the initial configuration.

A description of the distributed network operating system from the University of Wisconsin.

21. Marvin H. Solomon and Raphael A. Finkel; "The Roscoe Distributed Operating System"; Proceedings of the 9th Symposium on Operating System Principles; ACM, 1979

A description of the distributed network operating system from the University of Wisconsin.

22. Robert H. Thomas; "A Resource Sharing Executive for the ARPANET"; Proceedings of the National Computer Conference, 1973; AFIPS Press

This is a fairly detailed report on the design goals and implementation of RSEXEC, a network operating system for the TENEX hosts on the ARPANET.

23. James E. Thornton; "Backend Network Approaches"; Compcon 80 Spring; IEEE

A quick overview of backend networks.

24. Anand R. Tripathi, et al.; "An Overview of Research Directions in Distributed Computing"; Compcon, Fall 1980; IEEE

This article is mostly about abstractions of distributed systems; there is little real information, but it does have extensive references.

25. Stephen A. Ward; "TRIX - A Network-Oriented Operating System"; Compcon Spring 80; IEEE

    The author describes a different approach to operating system design in general, that can be extended to include networking fairly easily. There is not much detail on the actual network operations.

26. M.V. Wilkes and R.M. Needham; "The Cambridge Model Distributed System"; Operating System Review; Vol. 14, No. 1, Jan. 1980

    A discussion of the operating system for the Cambridge Ring, which is an example of a server system.

27. W. Wulf, et al.; "Hydra: The Kernel of a Multiprocessor Operating System"; Communications of the ACM; Vol. 17, No. 6, June 74

    This article describes the design philosophy of the Hydra kernel. The title is misleading in that little is mentioned about the multiprocessor aspects of the system. The article focuses on the concepts of objects and protection as they apply to Hydra. A good example is given that makes the concepts presented clear.

28. William A. Wulf and Samuel P. Harbison; "Reflections in a Pool of Processors - An Experience Report on C.mmp/Hydra"; Proceedings of the National Computer Conference, 1978; AFIPS Press

    An interesting article describing the successes and failures of Carnegie-Mellon's earliest multiprocessor system, from the developers' viewpoint.