1989

The Task distribution preprocessor (TDP)

Frank McCanna

The Rochester Institute of Technology
School of Computer Science

The Task Distribution Preprocessor (TDP)

by

Frank McCanna

A thesis, submitted to
The Faculty of the School of Computer Science,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

5/10/1989

Dr. Peter Lutz

5/10/1989

Dr. Andrew Kitchen

10 May 89

Peter Anderson

May 3, 1989

TITLE OF THESIS: The Task Distribution Preprocessor

I, Frank McCanna, hereby grant the Wallace Memorial Library, of RIT, and all other persons or institutions, permission to reproduce the material contained in this thesis document. I make no restriction against use of reproductions of all or part of this thesis for commercial use or profit provided that proper credit is given to the original author of the material.

Frank McCanna

Date: 5/10/89

Abstract

This document describes a software processor used to generate portable and efficient multiprocess programs that run on UNIX operating systems. The processor is designed to be a convenient method for converting single process C programs into distributed multiprocess C programs. Another goal is to have the processor used as a "back"end or "platform" for multitasking languages and code generators. Efficiency is targeted toward multiprocessor systems that provide facilities for sharing physical memory between processes running on separate CPUs. Portability is achieved with the use of the highly portable UNIX operating system and its companion C language. The C language is used as both the input and output language for the processor. Use of C as an object language gives portability to the generated multitask program. Using C as an input language, simplifies the interface with multitasking languages and code generators as well as minimizing changes necessary when converting C language single task programs to multitask programs. Initial implementation of the Task Distribution Preprocessor will generate C code that can be compiled and run on any UNIX system that provides message passing facilities.

Acknowledgements

My thanks to Dr. Peter Lutz and Dr. Andrew Kitchen for their support and ideas. I also want to thank my wife Anju S. McCanna, for suggestions concerning the writing style of the ''Purpose'' section in the introduction to this document.

Key Words and Phrases

Key Words in this document include: Language, Parallel, Distribution, Concurrency, Multiprocessing, C, Preprocessor.

Key Phrases in this document include: Multiprogramming Languages, C Preprocessors, Multiprogramming Platforms.

Computing Review Subject Codes

The major subject code is Software (D) and the secondary classification is Programming Techniques (D.1).

Table of Contents

1 INTRODUCTION AND BACKGROUND 1

1.1 Purpose 1

1.2 Previous Work 4

2 PROGRAMMING OPTIONS 18

2.1 Specifying a Task 18

2.2 Specifying Shared Data 18

2.3 Unshared Memory 19

2.4 Stack Memory 20

2.5 Library Functions 20

2.6 Synchronizing Tasks 21

2.7 Building As A Single Task 24

2.8 Second Level Modularity Conflicts 24

2.9 Other Synchronization Constructs 27

3 PROCESSOR SPECIFICATION 29

3.1 TDP Processing Diagrams 29

3.2 TDP Processing Description 31

3.3 Processor Design 36

3.4 Achieving Data Alignment 36

4 RUNTIME PLATFORM 43

4.1 Runtime Description 43

4.2 Runtime Diagrams 55

4.3 TDP Runtime Library 58

5 VERIFICATION AND VALIDATION 60

6 CONCLUSIONS 62

6.1 Problems Encountered and Solved 62

6.2 System Deficiencies 62

6.3 Suggestions for Future Extensions 62

6.3.1 Limiting the Transfer of Data 62

6.3.2 Variable Function Parameters 63

6.3.3 Attachment of Shared Memory 64

6.3.4 Exec() Overhead Reduction 65

6.3.5 Detection of Inefficient Remote Calls 67

6.3.6 Dynamic Linkage 68

6.3.7 Full Code Duplication 69

6.4 Related Thesis Topics for the Future 70

6.4.1 Automatic Task Generator 70

6.4.2 Portable Multitasking Languages 71

7 BIBLIOGRAPHY **73**

APPENDIX A: Example Program - Input to TDP **78**

APPENDIX B: Example Program - Output of TDP **81**

APPENDIX C: Running the Example Program **92**

APPENDIX D: Walk-Through for Example Program **93**

List of Figures

Figure 3-1 Program Load Module Creation without TDP 29

Figure 3-2 Multiple Load Module Creation Using TDP 30

Figure 4-1 Communication with the Fork Server 49

Figure 4-2 Creation of Child Task 50

Figure 4-3 Tightly-Coupled Configuration 51

Figure 4-4 Child Task Termination 53

Figure 4-5 Loosely-Coupled Run Time System 55

Figure 4-6 Tightly-Coupled Run Time System 57

1 INTRODUCTION AND BACKGROUND

1.1 Purpose

The Task Distribution Preprocessor (TDP) is a programming system for multiple process programs. TDP is designed to provide maximum efficiency and portability in multiple process programs running on tightly coupled multiprocessor UNIX systems. A ''tightly coupled'' multiprocessor system is one which provides parallelism in the execution of kernel code but maintains a single file system and controlling operating system [BAC]. The efficiency of the TDP system also depends on a multiprocessor system that allows the sharing of physical memory between user processes. Along with efficient use of tightly coupled multiprocessors, a major goal for the TDP system, is that multiprocess programs be able to run on any hardware platform running the UNIX operating system. Portability in TDP, will allow the use of inexpensive systems during multiprocessor application development and allow ease in porting these applications to new hardware.

The Task Distribution Preprocessor essentially provides parallel constructs to the C language [KER] without changing the language. There is only one TDP provided function to synchronize parallel code. Data sharing is provided by putting all static data into shared memory.

Why a task distribution preprocessor? What makes the multitasking platform generated by TDP different or superior to other multitasking platforms? The most important difference is that the output (and input) to TDP is standard C code that can be compiled on any UNIX system. With the success of UNIX, both industry and academia have realized the importance of software portability [LAP]. It is becoming so important that software development is very often done using UNIX and the C language for no other reason than for portability. C is very often chosen over other portable high level languages because of its efficiency and flexibility. Because UNIX is written in C, the first software tool often developed for a new hardware platform is a C cross-compiler. The cross-compiler can be used to compile the kernel and to compile an application to

run on the new UNIX platform. Often an application can be running on a new platform the same day a UNIX kernel is ported to the platform [BOD].

If efficiency can only be realized with tightly coupled multiprocessor systems, then what good is the portability? Very simply, TDP processed applications will be developed (for multiprocessor systems) on single CPU systems because single CPU systems are cheaper. Software development for multiprocessor systems can proceed on cheap reliable single CPU systems and then moved to target multiprocessor machines when the software has been thoroughly debugged and verified. When the time comes that tightly coupled multiprocessor systems are commonplace, portability between the makes and models of these machines will also be very important.

Is portability and efficiency really so important that TDP be used instead of more appropriate parallel programming languages? TDP can, of course, be used by C programmers to generate parallel programs, but TDP can also be effectively used as an object ''language'' for other high level multitasking languages. Other high level languages will ''inherit'' the portability and efficiency of TDP by ''compiling'' their

source code into C code that can then be processed by TDP.

1.2 Previous Work

Much of TDP is based on the remote call concepts presented by Nelson [NEL1,NEL2], summarized by Hamilton [HAM] and implemented by Birrel & Nelson [BIR]. A remote procedure call is best described in the words of Birrel & Nelson:

> ''The idea of remote procedure calls (hereinafter called RPC) is quite simple. It is based on the observation that procedure calls are a well-known and well-understood mechanism for transfer of control and data within a program running on a single computer. Therefore, it is proposed that this same mechanism be extended to provide for transfer of control and data across a communication network.'' [BIR]

Birrel & Nelson omitted two major features from their implementation that have been addressed by TDP. The two features are shared memory and parallel or ''asynchronous'' calls. Birrel & Nelson omitted shared address space for remote procedure calls due to a limitation of resources:

> ''In summary, a shared address space between participants in RPC might be feasible, but since we were not willing to undertake that research our subsequent design assumes the absence of shared addresses. Our intuition is that with our hardware the cost of a shared address space would exceed the additional benefits.'' [BIR]

In fact, most implementations of RPC do not provide shared memory. With the absence of shared memory facilities, remote procedure calls are limited to call-by-value or ''copy-in/copy-out'', no addresses as arguments or in data, can be shared across the RPC. This restriction may seem inordinately limiting but for some systems this limitation provided a simplicity that was very attractive. The main attraction was for use of this scheme for heterogeneous distributed systems. Falcone [FAL] found Birrel & Nelson's scheme to be very appropriate for his Lisp [LIS] based language for heterogeneous distributed systems.

The original goal of TDP was to simply add shared address space for a Birrel & Nelson - like remote call implementation using standard UNIX and the C language. The implementation was to be similar, but more flexible than an implementation by Glasser

[GLA1,GLA2] that copied data referenced by pointer arguments. The goals were expanded to include parallel or ''asynchronous'' remote calls and an efficient implementation for tightly-coupled multiprocessor environments when the importance of this hardware platform became apparent.

Like the Birrel & Nelson RPC system, TDP allows a program to alternately run as a single process or multiple process program without changing the source code. Birrel & Nelson provide this capability by linking in function ''stubs'' for remote functions when a program is built. The function stub is simply a fake definition for a function that, when called, will interface with a run-time system that provides for remote execution of the real function. Without the function stubs, the program can run as a single process. With the stubs, the program is a multiple process distributed program. TDP provides this capability by processing the source code and replacing remote calls with calls that interface with a run-time system that provides remote execution. Without TDP processing, a program can run as a single process, with TDP processing, the program becomes a multiple process program. TDP has retained the interchangeability between single and multiple process versions of a program. This flexibility allows conversion of single task

programs to utilize the advantages of multiple threads of execution while retaining a vehicle for simplified debugging with a single process version of a program.

For parallelism and synchronization with shared memory, TDP has taken the implementation restriction adopted by the Ada language [REF]. The restriction adopted is that shared memory is only guaranteed to be updated at program synchronization points. This restriction was adopted so that TDP could easily provide shared memory for either loosely-coupled or tightly-coupled multiprocessors but retain high efficiency in a tightly-coupled environment. A quote from an article by Mundie & Fisher discussing Ada, gives a very complete account of the issue:

> ''Shared Memory. Another issue facing process-model languages is the dichotomy between shared-memory systems and message-based systems. If the language assumes that the parallel processors share memory, then it is convenient and efficient to use that shared memory for interprocess synchronization and to allow multiple processes access to shared variables. However, that approach virtually precludes extending the parallel processing scheme to loosely coupled networks, which lack shared memory. Message-based systems, on the other hand, incur a substantial run time performance penalty, but are more general in that they can be used in a wider range of configurations.

> Ada rendezvous provides a diplomatic compromise between these two environments. Multiple tasks can access shared variables, but in a restricted way that allows for loosely coupled systems. Updating is guaranteed only at synchronization points, so that shared variables can in fact reside in separate memories with copies passed back and forth only during rendezvous.'' [MUN]

TDP takes a similar approach and uses shared physical memory if available and aligned copies of memory in loosely-coupled environments. Taking the philosophy that memory is only guaranteed to be updated at synchronization points allows the flexibility for efficient implementation on tightly-coupled machines without precluding implementation on loosely-coupled machines.

Compatibility between tightly-coupled and loosely-coupled environments also has a very important role for many multiprocessor systems. We often like to categorize a machine as either tightly-coupled or loosely-coupled but in reality multiprocessor systems often have both tightly-coupled and loosely-coupled components. Current multiprocessor implementations all have a limit to the number of processors that can share the same physical memory, when systems are built beyond this processor limit, there is a

"loose" coupling of processor groups. The Butterfly GP 1000 produced by BBN Advanced Computers Inc. of Cambridge Massachusetts is a case in point [RUS]. Currently, the Butterfly GP 1000 can have a maximum of 256 processors sharing the same physical memory. Building a configuration comprising more than 256 processors requires that portions of the system communicate via a communication network. A system made up of several Butterfly GP 1000 machines would have both tightly-coupled components and a loosely-coupled component. Ideally, run time support for such a configuration would make crossing the loosely-coupled component invisible while still allowing maximum efficiency in the tightly-coupled environment. Favoring the tightly-coupled environment for efficiency is based on the assumption that with more and more processors available in each tightly-coupled unit, there will be fewer and fewer applications that will require crossing a loosely-coupled boundary to access processing units. Because applications will primarily function using the tightly-coupled component of the system, maximum efficiency is targeted to that environment.

Use of an unaltered C language as the source language for the TDP system should bring maximum portability and utility to the system. Hopefully, using a popular and well

supported language like C will allow TDP to be easily interfaced to existing applications and code generators. Going with standard C as the input language, a number of parallel implementations of the C language were explored for possible use as a basis for TDP.

The first parallel C implementation considered is ''Refined C'' implemented by Dietz & Klappholz [DIE2]. This implementation is based on concepts presented in an earlier paper by the same authors describing how any conventional sequential language can be ''refined'' to provide parallel constructs [DIE1]. Dietz & Klappholz correctly identified the three most difficult constructs to deal with in changing a sequential language into a parallel language:

> ''References to global data: since the complexity of certain analysis techniques requires flow-analysis to be localized to small regions of a program, the flow-analysis is forced to make the "safe" assumption that every function and subroutine call might alter every global variable.
>
> Pointer references: for similar reasons, most references via indirection on pointers must be assumed (by the compiler) to potentially alter large sets of variables.

References to indexed data structures: since it can be difficult to determine which element(s) of a data structure an indexed reference can access, it may again be necessary to assume that such a reference potentially touches the entire structure.'' [DIE2]

Refined C introduces a construct called a partition to replace global memory and a construct called an environment to specify limited access to data. There are two major reasons that Refined C was not chosen as a model. The first is that elimination of global memory and data structures as they are normally presented in C programs is a major change to the language. Changing C dramatically dilutes arguments for using it in the first place. If a non-standard source language is acceptable, then using a highly expressive distributed programming language like SR [AND], MML [BOA] or Linda [AHU] is a more desired approach. The second problem with Refined C is that partitions are implemented with the use of an added layer of indirection for referencing memory. Double indirection to fetch data is a valid approach for heterogeneous distributed systems but is unacceptably inefficient for program distribution among homogeneous systems.

The Concurrent C language presented by Gehani & Roome [GEH] was also discarded as a model because of extensive changes to the C language. This implementation also had the added disadvantage of not providing shared memory facilities. In Concurrent C, shared memory is left for the C programmer to implement as described in a quote from the article describing the language:

> ''Some final comments about shared memory are in order. First and foremost, we wanted Concurrent C to allow (and encourage) programmers to write portable programs that will run efficiently on multiprocessors with or without shared memory. This is why we choose a message-passing model, and why we have not added any language constructs for dealing with shared memory or for simulating shared variables in a non-shared memory environment.'' [GEH]

Leaving the implementation of shared memory entirely to the programmer and encouraging the use of a message passing model for shared memory is unacceptable for the tightly-coupled multiprocessor environments targeted for TDP.

There is an implementation of parallel C, also called ''Concurrent C'', presented by Tsujino et. al [TSU]. In this Concurrent C, shared variables are implemented using the concept of a monitor. A data object is retrieved from a monitor by use of a function call.

Although monitor access to shared variables provides the abstraction necessary for multiple process programs in loosely-coupled environments, this language implementation is inefficient for tightly-coupled systems. Always accessing global memory through a monitor will also force major restructuring in sequential C programs that are to be converted to multiple process programs.

A parallel implementation of the C language presented by Sonnenschein [SON] provides shared memory and synchronization with the use of ports and signals. Ports are special global shared memory that is referenced like any other memory but cannot be referenced until specifically assigned to a particular process. Signals are used to synchronize the assigning of ports and special functions assign() and release() are used to gain or relinquish access to a port. Objections to this implementation are two fold. One is that signals and ports are implemented by adding reserved words to the C language. Although the change to the language is not as profound as Gehani & Roome's Concurrent C and Dietz & Klappholtz's Refined C, the language is changed to the degree that a standard C compiler could not be used to compile the source code. The other problem is that there is no provision for more than one process to simultaneous access a

particular port memory. Shared memory is always of a restricted nature, real-time simultaneous access to shared memory is expressly forbidden.

Binding [BIN] presents a parallel implementation of the C language that allows the sharing of all global memory without restriction. The C language is essentially unaltered satisfying another goal for the TDP environment. In Binding's parallel C, parallelism is only simulated; all program tasks reside in a single UNIX process. Context switches are not ''preemptive''; a program must explicitly relinquish control of the CPU by calling a ''ContextSwitch()'' function. A similar implementation by Cormack [COR] allows preemptive scheduling but also simulates parallelism inside a single UNIX process. Both Binding's and Cormack's implementations provide a model very close to what is provided by ''light'' processes or program ''threads'' [RUS] provided by many multiprocessor operating systems. Programs using the Binding or Cormack implementations could be easily upgraded to utilize the ''light'' process primitives of many of these multiprocessor systems. TDP does not utilize this model because it does not allow compatibility with loosely-coupled systems and is not yet common enough among UNIX systems to be widely available. By making loosely-coupled and tightly-coupled multiprocessor

implementations invisible to an application, that application will be much more portable across varying processor configurations. Currently, very few UNIX implementations provide ''light'' processes; therefore, exclusive use of them, would impact the portability of the system severely. It is also hoped that the performance impact in the creation of ''heavy'' processes can be lessened to be very near the performance impact for the creation of a program ''thread'' or ''light'' process. However, because the development of ''light'' processes in multiprocessor systems is very important, nothing in the design of TDP will preclude the use of threads in combination with TDP to achieve performance and portability benefits in both loosely and tightly coupled environments. The design of TDP will also not preclude the use of program ''threads'' to implement the TDP run time system if in the future threads become a standard feature of all UNIX and UNIX-like operating systems and can provide parallelism and shared memory across loosely-coupled and uncoupled networked distributed systems.

Due to the special goals for TDP that could not be satisfied by any published version of a parallel C language, TDP design went in a completely new direction. Adding reserved words to the language to specify parallel or remote functions reduces the

portability of the source code and does not lend well to conversion between single and multiple process versions of the program. Adding reserved words would also eliminate the possibility of having a single process version of a program, compiled without TDP, using a standard C compiler. Therefore, to specify parallelism, TDP uses the modularization feature of the C language. The feature of separate module compilation is used by TDP to define what source code will reside in separate tasks. A single source file defines a program task that will run as a separate UNIX process.

Using the C preprocessor's #include directive, program modularization can be achieved on several levels. A program can use modularization for separate compilation and information hiding as usual, by separating source code into separate files. To build the program using TDP, ''task'' source modules can be created that have nothing but #include directives that gather together source modules that comprise a task. These ''task'' source modules are the source modules input to TDP. Because TDP runs the C preprocessor on all source modules before processing, the ''task'' source modules will be expanded to include all files in a task. Details of how scope conflicts are resolved in this scenario are explained later in this document.

Shared memory in TDP processed programs, requires no specification by the user because all static memory is shared. All static memory is also aligned so that one copy of static memory can be shared by all program processes without pointer conflicts. Because TDP design is expressly aimed at tightly-coupled multiprocessor environments, this radical design can be used with the benefit of high efficiency for the target platform.

2 PROGRAMMING OPTIONS

2.1 Specifying a Task

A programmer informs TDP that certain functions are to be part of a separate program task by putting the declarations for these functions in a separate C source file. TDP assumes that the code in each C source file is code that will reside in a separate process for the program.

When converting single process C programs to TDP multiprocess programs, functions should be rearranged in the source files so those that can run in parallel are in separate files. Because TDP runs the C preprocessor on all source files before processing, several source files can comprise a task by using the #include preprocessor directive.

2.2 Specifying Shared Data

All static memory in a program except that which is declared in a library function, will be shared between program tasks. Any changes to static memory will be reflected in all tasks. Not sharing library function static memory allows the proper execution of many standard UNIX library functions that are not designed to be re-entrant. Any data declarations preceded by the C reserved word ''static'', will also be shared but only the task that declared it will have access to the label describing the data. String data falls into the same category as local static memory. All string data is shared between tasks. Because local static memory and string data is also aligned, pointers to this memory can be passed between tasks.

2.3 Unshared Memory

To declare memory that is not shared, a programmer must declare an automatic variable or call *malloc()* or *sbrk()* to get dynamically allocated memory. Pointers to unshared memory cannot be shared between program tasks.

2.4 Stack Memory

The stack memory for each task is not shared, any alterations to stack memory will not be reflected in other tasks. Therefore, pointers to automatic variables must not be passed as arguments to functions in other tasks.

2.5 Library Functions

Library functions are linked into program tasks as the last step in creating individual task modules. Every library function that is used in the entire program is linked into EVERY task module whether a particular task calls that function or not. This prevents data alignment problems caused by static memory declared inside a library function. If one task linked in a library function that declared static memory, and another task did not link in that function, the static memory may destroy the memory alignment of the two tasks. Static memory declared in library functions is the only static memory in a TDP created program that is not shared. Not sharing library function static memory allows the proper execution of many standard UNIX library functions that are not designed to be re-entrant.

2.6 Synchronizing Tasks

When the user calls a function that is defined in another source file, he/she is starting a parallel task. The remote function call will retum a child task descriptor and this descriptor can be used to wait for the child task at any point in the execution of the parent. An example of how a parent waits for the child is as follows:

```
func1()
{
      int child_descr;
      int retval;

      child_descr = func2();

      . . . . . .

      parallel code

      . . . . . .

      retval = TDPwait(child_descr);
}
```

func1() and func2() are defined in separate source files and the code between the call to func2() and TDPwait(), is code that will be executed in parallel with the code in func2(). The return value of func2() is returned from TDPwait(). TDPwait() is identical to the *absorb()* function in the parallel C implementation presented by Cormack [COR]. The child descriptor is a value that is unique to a particular call and is different for every call even if multiple calls are made to the same function.

If by the time TDPwait() is called, the child task has already terminated, then TDPwait() simply returns the value returned by the remote function. If the child task is still running when TDPwait() is called, the parent task will be suspended until the child

task terminates. The call to TDPwait() can be anywhere in the parent task and it is not necessary to call TDPwait() at all. There is nothing in the TDP platform that requires that a parent task wait for its children. Children can terminate long after their progenitors have terminated.

2.7 Building As A Single Task

An application that uses the TDPwait() function can be built and run without TDP by simply creating a function called TDPwait() that returns its argument. The return value of a function would simply be passed through the TDPwait() function unaltered. This may be useful if there is a suspected bug in the parallel constructs of a program or if it is desired to use a debugger that can only function on a single process version of the application.

2.8 Second Level Modularity Conflicts

It is expected that some programmers will define TDP tasks by simply creating files that include groups of files for a program using the C preprocessor #include directive. If a program source file is normally compiled separately but is now merged with other files, a number of conflicts can arise:

1) Two source modules that are normally compiled separately may have a variable with the same name declared with the keyword ''static'' outside of a function definition (therefore giving it scope for the entire source file). If these two source files were merged together, the variable would be multiply defined. TDP solves this problem when it renames all variables declared with the ''static'' keyword. Each declaration of the static variable would be given a separate new name when moved to the global data file, thereby resolving the conflict.

2) The scope of a function name can also be restricted to a single source file using the ''static'' keyword when the function is declared. It is possible that two source files may use the same ''static'' function name and if the two source files were merged the function would be multiply defined. TDP will generate an error message and terminate processing if this problem occurs. It is up to the programmer to choose a separate name for multiply defined functions in this situation.

3) Each source module that is normally compiled separately will have its own set of #include statements for structure definitions and manifest constants. If these source files are then merged with others, many #include statements will be duplicated. TDP defines the manifest constant ''TDP'' so when there are multiple #include statements they can be surrounded by conditional preprocessor directives:

```
#ifndef TDP
#include <stdio.h>
#endif
```

4) A manifest constant can be given one value in one source file and a different value in another. If the two files are merged there may be a conflict. Some C preprocessors allow redefinition of a manifest constant, but for those that do not, a programmer should use the #undef preprocessor directive at the end of a source file for manifest constants that may conflict.

2.9 Other Synchronization Constructs

TDPwait() is the only TDP function provided to synchronize parallel tasks. If a programmer needs more elaborate communication or synchronization between tasks, the entire UNIX C library is at the programmer's disposal. Because TDP processes are true UNIX processes, any UNIX system call that can be used to synchronize separate UNIX processes can be used to synchronize TDP tasks. TDP reserves for its own use, only one message queue identifier and one signal. Both TDP's message queue identifier and signal number can be changed by the user to eliminate conflict with their use in a particular application.

3 PROCESSOR SPECIFICATION

3.1 TDP Processing Diagrams

Figure 3-1 Program Load Module Creation without TDP

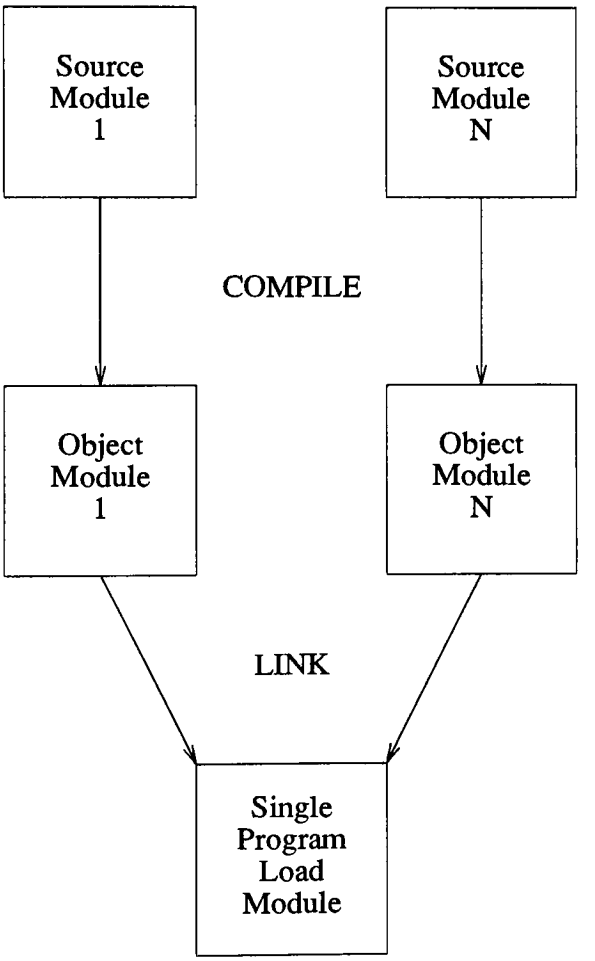
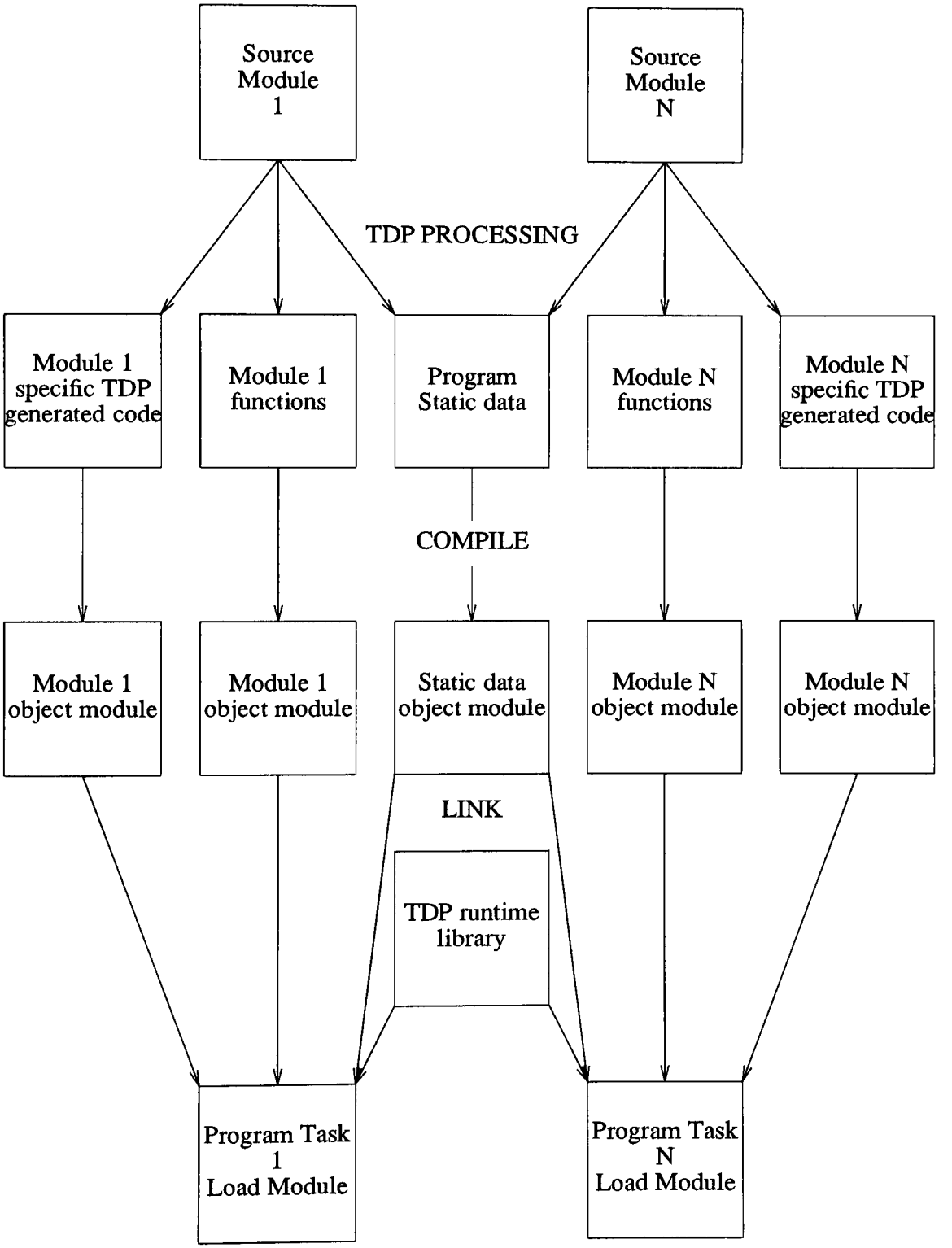


Figure 3-2 Multiple Load Module Creation Using TDP



3.2 TDP Processing Description

TDP accepts as input, all the C source files that comprise a program. TDP will assume that each source file is to be a program task. A program task process will be started whenever one of the functions defined in the task, is called by a function in another task.

TDP processes the source files in two passes. Before the first pass, all the source code is processed by the C preprocessor (cpp). TDP processing in pass one proceeds in the following manner:

o If TDP encounters a function declaration in a source file, the function name and its arguments are saved in a linked list for later reference. The name of the file and the number of the function declaration is also stored for later reference.

- If TDP encounters a string constant, the string is saved in a linked list to be later used to initialize an array that will contain the data for all strings in the program.

- If the C reserved word "static" is encountered, TDP saves the declaration to be used later in a global declaration that will replace the static variable declared.

When all the program's source files have been read, TDP outputs an array declaration that will replace all the string constants encountered in the program. The string constants in the program will be replaced by offsets into this string array. Declarations are also generated for variables that will be used to replace the static variables in the program. Unique names are chosen for all static variables simply by appending the value of a counter to a unique string. The declarations are all output to the single TDP generated global data file.

In the second pass, TDP generates a task output file for each task source file. TDP processing in pass two proceeds in the following manner:

o If a global declaration is encountered, it is appended to the single data file for the program. If the declaration does not have an initializer, it is given one. Initializers are given to prevent misalignment that would occur when uninitialized static data is put in the BSS section of memory.

o If a local static variable declaration is encountered, a ''#define oldname newname'' is appended to the task output file. The ''oldname'' is the original identifier name in the task input file and the ''newname'' is the replacement name given to the static variable in pass one. When the end of the current block is encountered, (i.e the end of the scope of the variable) a ''#undef oldname'' is appended to the output file.

o If a string constant is encountered, it is replaced with a pointer offset into the string array that was put into the global data file after pass 1. The correct offset into the string array for this string constant was calculated in pass 1.

o When a function call is encountered, TDP looks to see if the function declaration

was encountered in pass 1. If the function call is not defined in the program, the function name is put in the list of library routine calls for the program. If the function call was defined but in a different task, the function name is put in a list of remote function calls for the task. The remote function call is replaced in the source file with a call to a special TDP function that initiates a remote call using the original function call arguments and the function number. The function number for this function was assigned in pass 1 when the function declaration was encountered. No information is saved for functions that are called and defined in the same task; this code will just make a normal local call. The code for a local call is just echoed to the output file.

- o If the function main() is encountered in a source file, TDP renames the function and records the fact the current task is the startup task for the program. At the end of this function, TDP will output a call to a special TDPexit() function that allows the program to exit with notification given to the fork server.

- o If a call to *exit()* is encountered in the user's source code, TDP replaces it with a TDPusrexit() call which will exit with proper notification given to the fork server.

o All code not mentioned above, is copied directly from the the input source file to the task output file.

In a separate file, TDP generates a declaration for a global structure that contains all information about the functions defined in the program. This structure is initialized with the task name, function name, function pointer and the size of function arguments. The function number is an index into this array of function structures.

TDP also generates a single global structure that contains all the necessary run time information about the program and about each particular task. This structure also contains a pointer to a linked list that describes each instance of a function call for a particular task.

Both the function list and the run time global structure are kept in private static memory. Every task has its own copy of these structures, they are not shared between tasks.

3.3 Processor Design

The TDP parser/code generator has a main() function that contains two loops that operate on all the files on the command line. The first loop is pass one of the processor and the second is pass two.

The scanner used by TDP is created using the UNIX utility *lex*. It has the capability to scan all C declarations and calls a number of special functions for parsing and output. The same scanner is used for both pass one and pass two of the processor.

TDP parsing and output routines manipulate two linked lists. One linked list is used to store information on function declarations in pass one. The other linked list is used as a general place to save declared identifier and their types.

3.4 Achieving Data Alignment

To provide shared memory functionality, TDP aligns global memory in all tasks so that it can be accurately copied from task to task without relocating memory references. TDP puts all the static data declared in a program into a single source file and creates the load modules for each task in such a way as to give the static data identical virtual addresses in all the load modules for the program. Data can be directly copied between the tasks in the program because all tasks operate on identical copies of the data. More importantly, shared physical memory is feasible because pointer values stored in memory are valid for all processes in the program.

To accomplish memory alignment there are some general characteristics of UNIX linker/loaders upon which TDP relies. When a UNIX linker/loader allocates memory for a program load module, the addresses assigned to code and data objects are virtual

addresses. When memory is accessed by a program at run time, the virtual pointer to a variable is remapped to a physical memory address.

TDP puts all static data declarations into a single C source file for a program. This source file is compiled and the object module is linked into every program task. This assures that all static data will be placed in a single block of memory space in all programs. This block of "memory space" or "section", is not guaranteed to start at the same address in all the program tasks. Typically the linker/loader will put this data section after the code section in virtual memory. The exact placement of the data section varies from system to system but will always partially depend on the size of the code section. This means that, although the data objects are the same distance from each other in each task, the addresses are still different in every task depending on the size of the code section for that task. To achieve data alignment, the virtual addresses of the data must be shifted so that the virtual address of the start of data objects is the same in all tasks. TDP solves this problem by telling the linker/loader to leave the task load module relocatable after building it. TDP then goes through all task load modules looking for the one with the largest start address for the data section. TDP then tells the linker/loader to put a hole

in the data section equal to the difference between the starting address of the module with that largest data section start address and the start address of the current load module's data section. This hole in the data section shifts the addresses for the static data in the task load modules so that they all begin at the same location.

The UNIX System V Release 3.0 Programmer's Guide gives a complete description of how to manipulate load modules using the Link Editor Command Language for the linker/loader: *ld*. An *ifile* is a file which contains Link Editor Command Language commands to manipulate load modules. The following is the default *ifile* run by *ld* when executed on an Intel 80386 System V Release 3.X UNIX system:

```
SECTIONS
{
    .text 0xd0 : { *(.init) *(.text) *(.fini) }
    GROUP BIND( NEXT(0x3ff000) +
        ((SIZEOF(.text) + ADDR(.text)) % 0x2000)) :
    {
        .data : { }
        .bss : { }
    }
}
```

Most of the detail in the code above is unimportant to this discussion but is included for completeness. The meaning of each construct is explained in the UNIX System V Programmer's Guide [UNI]. The following *ifile* is an example of one that was used to build one of the load modules in a TDP processed program. Notice that it is identical to the default specification except for the increment value placed in the .data section specification:

```
SECTIONS
{
    .text 0xd0 : { *(.init) *(.text) *(.fini) }
    GROUP BIND( NEXT(0x3ff000) +
        ((SIZEOF(.text) + ADDR(.text)) % 0x2000)) :
    {
        .data :
        {
            . += 0x318;
        }
        .bss : { }
    }
}
```

In this particular case, the ''hole'' in the data section is 318 (hex) bytes in length. This increment command will cause *ld* to start allocating data for declarations after skipping 318 (hex) bytes. By looking at this *ifile* we can surmise that there is a load module for this program that has a data section with a start address that is 318 (hex) larger than this load module's data section start address.

If a link/loader on a particular system does not provide a means for putting a hole in

the data section, TDP will generate a dummy declaration equal to the size that the hole would be. The declaration is made so that it is the first data allocated by the linker/loader and all of the ''real'' data will follow it.

4 RUNTIME PLATFORM

Simply aligning data is not all there is to the story, TDP must also provide an efficient means for task spawning and synchronization. A library of run time routines is linked into all TDP processed program tasks. The TDP library serves to manage all aspects of the TDP run time environment. The only visible evidence to the user that this library is used, are calls that he/she makes to the TDPwait() function that allows a parent task to wait for a child task.

4.1 Runtime Description

TDP generates a program that runs in the following manner:

- To start a program, the user starts the fork server. The fork server does a fork() exec() to start the first task, then waits on a message queue. A UNIX message is sent to the fork server whenever a task calls a function in another task. Upon receipt of the message, the fork server executes a fork() exec() according to the specifications in the message. Therefore, the fork server is the process parent of all program processes.

- Upon startup of the first task in the system, any necessary initialization is performed and the program's original main() function (now renamed) is executed.

- If a function is called that is defined in the current task then nothing unusual occurs; the function is executed like any function in a single process program.

o If a library function is called, again, nothing unusual occurs; the task's copy of the function is executed in the same way as in a single process program.

o If a program function is called that is defined in another task, the following sequence is performed:

1) A special TDP function is called. TDP has placed a call in the program code that appears as follows:

TDPcall(funcnum,arguments, . . .);

The funcnum is a constant integer representing the function number calculated by TDP. The arguments are the exact arguments placed by the user for this function call in the original source code. During preprocessing, TDP has replaced the original text of the call:

func(

with:

TDPcall(funcnum,

The TDPcall function is declared using the variable parameter macros defined by the UNIX System V macro package varargs (see the UNIX System V Programmer's Reference manual for a complete description of these macros):

```
TDPcall(va_alist)
va_dcl
{
    va_list pvar;
    int funcnum;

    va_start(pvar);

    funcnum = va_arg(pvar);
```

The variable ''funcnum'' will be an index into the function list structure placed (by TDP) into the global data file.

2) TDPcall() creates a string to be sent in a message to the fork server. The string will contain the target task name, the called function number, the instance number of the function call, the current process id number, and the name of the shared data file.

3) TDPcall() copies the function arguments into the message to be sent to the fork server, placing them immediately after the string describing the call. The number of words to copy into the message is extracted from the function list

describing all functions declared in the program. The function list's value for the number of words for arguments in a particular function, was determined when the function declaration was parsed during program building. The following code is executed in TDPcall() to copy the function arguments into the message (Again, a variable number of function arguments is extracted using **varargs**):

```
for (i=0; i < TDP.funclist[funcnum].numarg_words; i++ )
message.args[i] = va_arg(pvar,int);
```

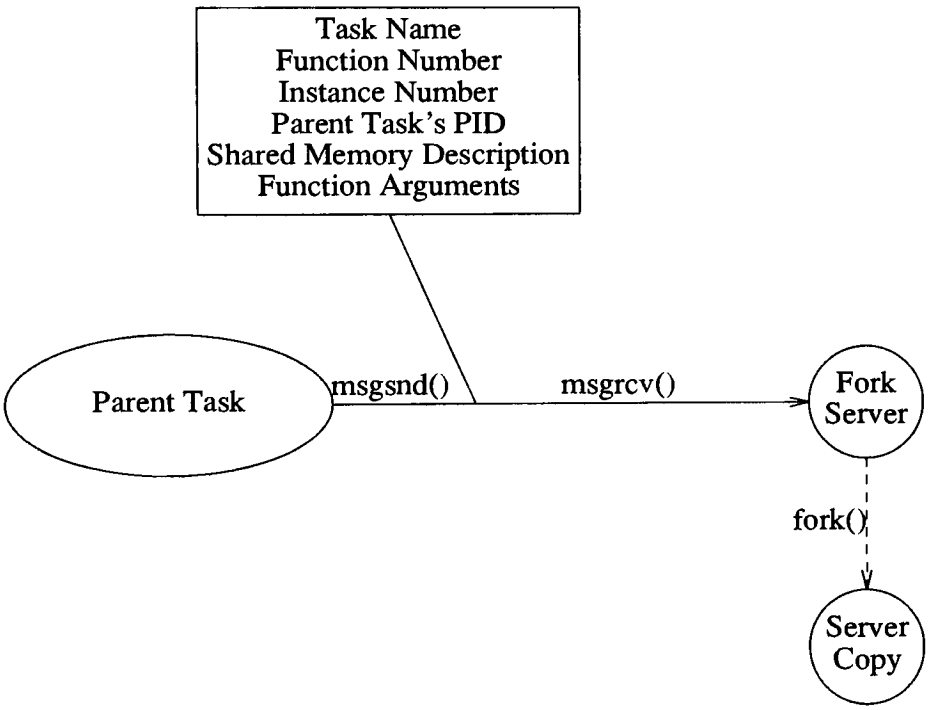
The variable ''TDP'' is the global structure that references all TDP run time structures. The structure member ''funclist'' is the structure that contains information describing each remote function. The structure member ''numarg_words'' is the number of memory words needed to contain all the function arguments for a particular function. The variable ''funcnum'' was assigned in step 1 with the first argument to TDPcall(). The function *va_arg()* is the **varargs** macro that allows extraction of a variable number of arguments in a function call.

4) TDPcall() copies all shared data to a temporary file or arranges for attachment of shared physical memory. (attachment of shared physical memory is not implemented in TDP Version 1.0). TDPcall() then sends the message containing the task parameters and function arguments to the fork server.

5) After sending the UNIX message to the fork server, TDPcall() returns a child process descriptor to the caller. The child process descriptor is used later as an argument to the TDPwait() function. The call to TDPwait() allows the parent task to wait for the completion of the child task.

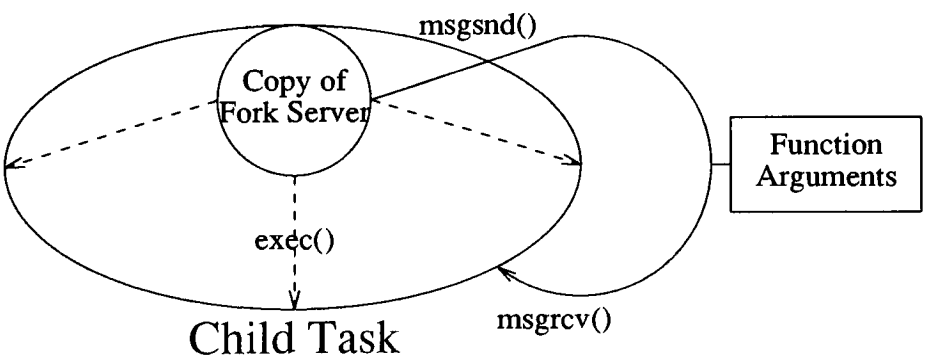
6) Upon receipt of a UNIX message, the fork server executes a fork() and the parent fork server process waits again on the message queue.

Figure 4-1 Communication with the Fork Server



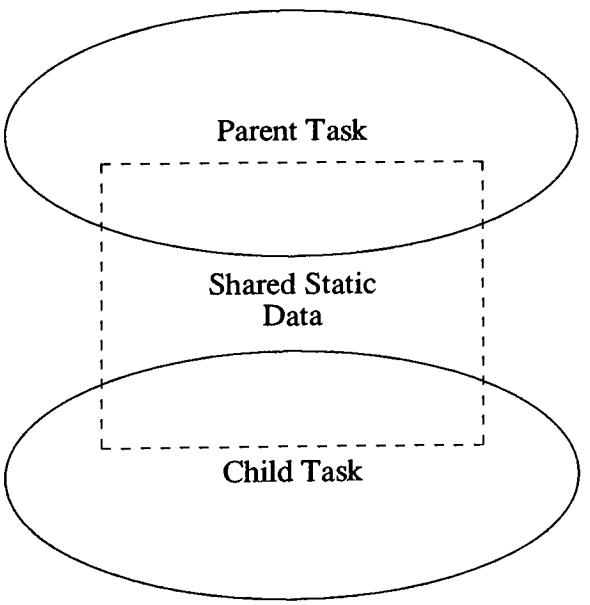
7) The child of the fork server sends a message containing the function arguments to itself (to a process with its own process id). The child fork server then executes an exec(), passing the string in the UNIX message to the exec() function. The child of the fork server has now "become" the child task (same process id). The child task's main() function receives the function arguments sent in a message before the exec().

Figure 4-2 Creation of Child Task



8) The TDP generated main() function in the child task has access to the four arguments passed on the command line. The function number, function call instance number, and parent task's process id are the first three arguments. The temporary file containing the shared data (or the shared memory description) is the last argument. The child task copies in shared memory from the temporary file created by the parent task (or attaches to shared physical memory).

Figure 4-3 Tightly-Coupled Configuration



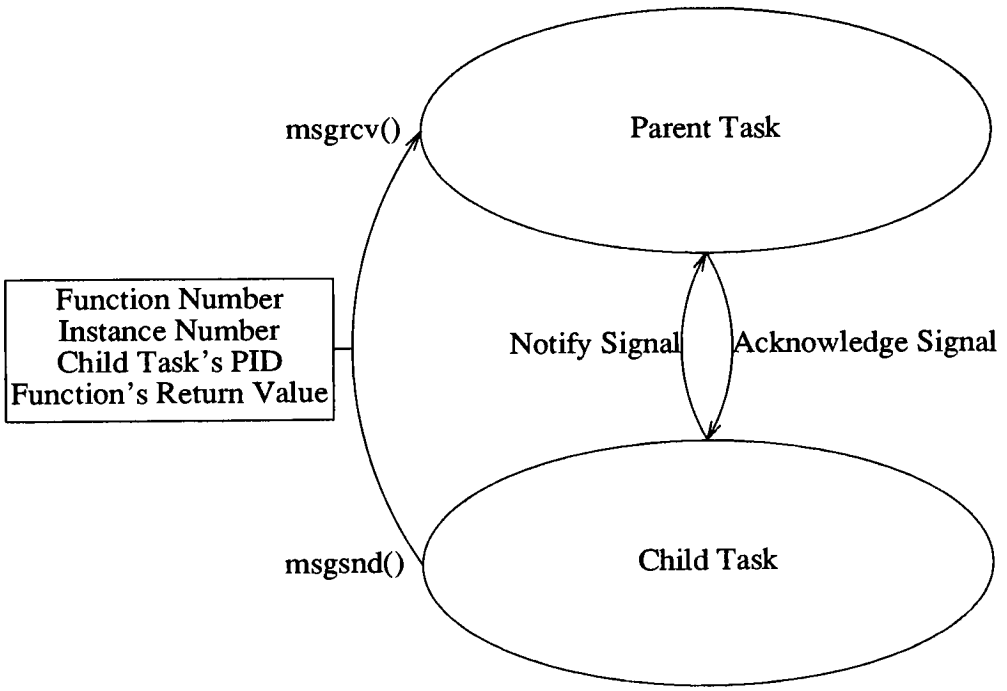
9) The task's main() function then pushes the function's arguments on the stack and calls the function specified in the first command line argument. The length of the arguments for this particular function is obtained from the TDP generated function list structure.

10) After function execution, the main() routine calls a function that will copy shared memory to another temporary file for return to the parent task. If shared physical memory is used, the child task simply detaches from this memory.

11) After the shared data is copied back to a temporary file, the main() function of the child task sends a UNIX message to the parent task. The child task also sends a signal to the parent task using the kill() UNIX function. The process id of the parent task, necessary for sending the parent the message and signal, is obtained from the third argument on the childs task's command line. The child task then waits for verification from the parent task that it has received the sig-

nal. If no verification comes, the child task again sends a signal to the parent. The parent could have missed the original signal for various reasons, the most likely being that it was busy processing the return of another child task.

Figure 4-4 Child Task Termination



12) All tasks have a signal handling function assigned to handle the termination of child tasks. When the child task sends the parent the termination signal, UNIX will branch the parent task to the signal handling function. The signal han-

dling function in the parent task will look on the TDP message queue for messages with a type equal to its process id. The UNIX message explains which function has terminated, its instance number, the process id of the child task, the return value of the function and the name of the temporary files containing the shared data to be copied back to the parent. The parent tasks sends a signal back to the child (using the process id in the message) notifying it that the message has been received. Receiving this response signal will cause the child to exit normally.

13) The parent task compares the bytes in the original shared data file sent to the child task, and the altered shared data file returned by the child. If a byte has changed, it is copied back to the parent tasks address space. This processing does not take place if shared physical memory is being used between tasks (attachment of shared physical memory is not implemented in TDP Version 1.0).

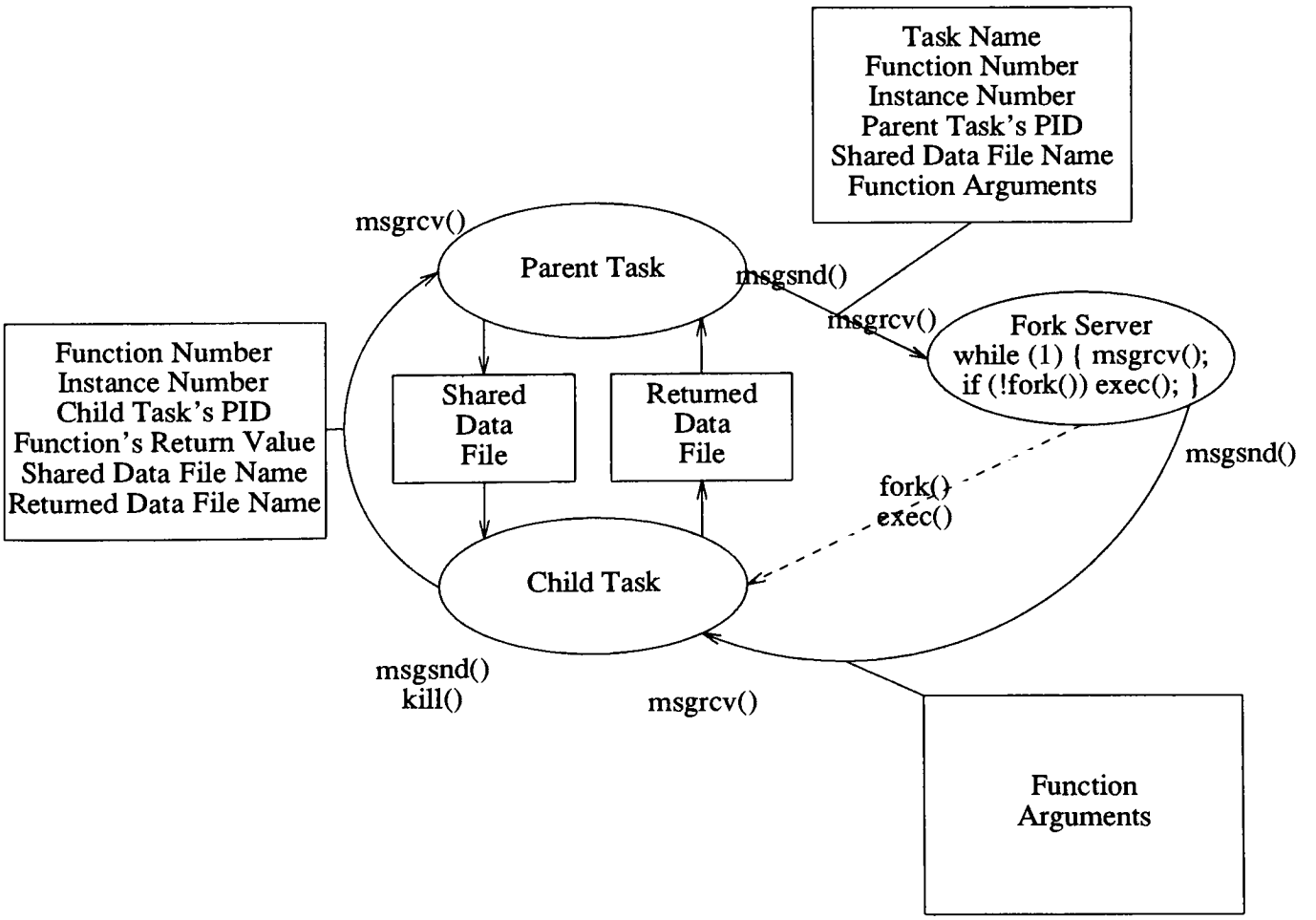
14) After copying the shared data back into its address space, the parent task continues execution from the point it had been interrupted. If the program was

suspended in TDPwait(), waiting for this function call to return, the TDPwait() function returns with the return value of the function. If the parent has not yet called TDPwait() for this call instance, the return value of the function call is saved and will be returned by TDPwait() when it is called for this call instance.

4.2 Runtime Diagrams

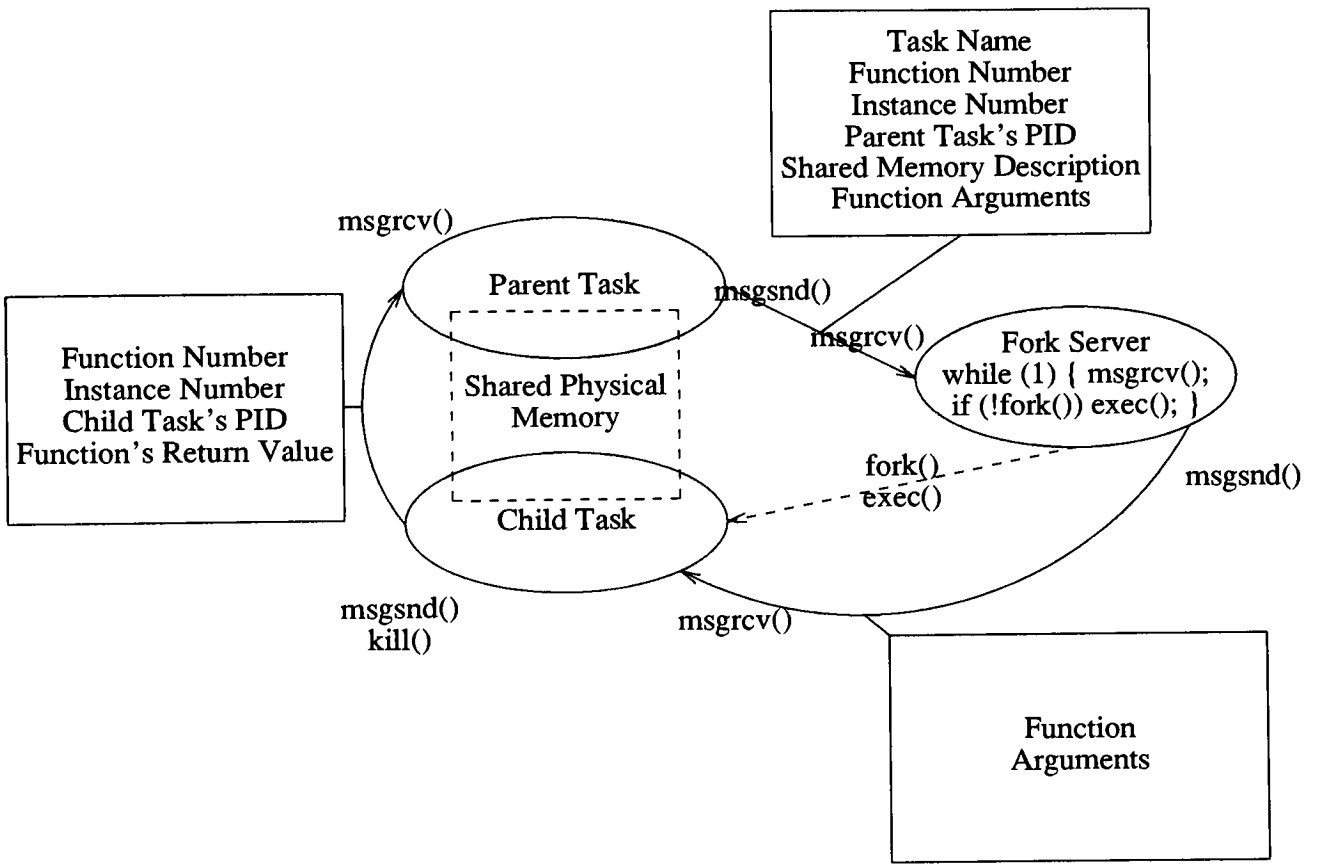
The following diagram summarizes the processes and communication involved in a remote function call when physical shared memory is not possible:

Figure 4-5 Loosely-Coupled Run Time System



The following diagram summarizes the processes and communication involved in a remote function call when it is possible to share physical memory:

Figure 4-6 Tightly-Coupled Run Time System



4.3 TDP Runtime Library

The TDP run time library is linked into all program tasks. The library provides all code necessary to manage parallelism in the C language. The only visible interface to this library in the user's source code are calls to TDPwait() to synchronize parallel tasks.

The call to TDPwait() can be easily ''stubbed out'' by the user when TDP is not used to build the program. A sequential version of the user's program can be built by just providing a function called TDPwait() that returns the argument passed to it. In this manner, a programmer can easily switch back and forth between sequential and parallel implementations of a program when necessary for debugging purposes.

Although the only visible interface to the TDP run time library in the user's source code is TDPwait(), in the code produced by TDP there are a number of calls

visible. A call to TDPcall() is made wherever the program calls a remote procedure and TDPusrexit() is called wherever the user has specified a call to *exit()*. At the end of the user's *main()* function TDP will also place a call to TDPexit() which will properly notify the fork server of the termination of the user's main function.

As well as specific calls placed in the user's source code, TDP also links in a number of routines used by these calls. Each task is also provided a *main()* function that plays the executive roll in setting up the stack for a procedure call, executing a call request, and receiving and returning shared data to a parent task.

5 VERIFICATION AND VALIDATION

The important elements of TDP that should be tested are the capabilities of the scanner/parser and the correctness of the run time environment. The capabilities of the parser can be tested by processing a program that contains all possible C data declaration syntax. All possible function declaration types, local static declarations and function call syntax should also be tested. Code or data that is not used in a running program must be visually verified.

The run time environment can be tested on a program by program basis. A program's results when run after processing by TDP should be identical to results obtained when the program is not processed by TDP. The only exception to this, is that parallelism in a program that affects output will only be visible in a TDP processed program. For comparison of output, there will be a TDPwait() function call immediately after all remote function calls to eliminate parallelism. Parallelism will be demonstrated by testing a program that generates terminal output in separate

tasks. Parallelism is demonstrated by the random interleaving of the output.

6 CONCLUSIONS

6.1 Problems Encountered and Solved

TDP was first written as a one pass processor and this imposed a number of complications and limitations on the system. TDP was redesigned as a two pass processor and memory restrictions were eliminated with the use of linked lists and dynamic allocation of memory in the program. The result is a slower processor but one that is much simpler and one that produces more correct and efficient code.

6.2 System Deficiencies

- The initial implementation of TDP will not handle typedef declarations.
- The sharing of physical memory between tasks will be left for future implementation.

6.3 Suggestions for Future Extensions

6.3.1 Limiting the Transfer of Data

When data is transferred using secondary memory (a file), data transfer could be improved by not copying data that will not be used by the destination function. Copying data back to the parent task could also be reduced by not copying data that was not changed by a remote function call. TDP could be easily amended to have a user interface that would instruct TDP to generate conditional code that limited data transfers with calls to specified functions.

6.3.2 Variable Function Parameters

Some C functions commonly operate with a variable number of function parameters. Common examples are the printf() and scanf() functions. To pass a variable number of parameters to a remote function, TDP can calculate the size of the parameters in each call during processing or generate code to calculate the size at run time. To calculate the size of arguments during processing is the preferred method because it improves run time efficiency. To calculate this at processing time however, requires that TDP parse all data declarations in a function and calculate the size of each identifier declared. The run time solution is less efficient but much easier. All that is required is that TDP generate code that adds sizeof(identifier) statements for each of the identifier used in the call.

6.3.3 Attachment of Shared Memory

Tightly-coupled multiprocessor systems allow the sharing of physical memory between processes. The alignment of TDP task static memory makes the TDP run time system ideally suited to fully capitalize on this capability. Data pointers in all tasks are equivalent so they do not need to be relocated. Data pointers can be shared without restriction in real time. Because nothing needs to be copied or relocated, there is no reduction in the speed of memory access.

In addition, many tightly-coupled multiprocessor systems also share a file system with processors with which they cannot share physical memory, TDP should be able to make a decision at run time, whether to start a process sharing physical memory or to start a process in a loosely-coupled manner.

6.3.4 Exec() Overhead Reduction

When shared physical memory is used between program tasks, the fork server exec() call overhead can be reduced by "dummying up" the object task's data section headers. If physical memory is shared, there is no need for any task except the startup task to load data sections into memory. The only data section that is required is that which defines the stack for the program. More often than not the stack section is not part of a load module but is allocated dynamically when a process starts. Because TDP tends to increase the amount of static data and more importantly, makes all static data initialized, there could be a great increase in child task startup speed if this data was not loaded.

Some UNIX systems also allow shared and/or locked text sections. If the text sections for all program load modules could be either shared or locked, multiple in-

stances of the same program task would start up much faster.

6.3.5 Detection of Inefficient Remote Calls

A profiling mechanism can be added to TDP so that inefficient function calls can be identified by measuring time variables for remote call efficiency. Code can be generated that creates a variable that stores the time it took for the execution of the parent task code before it started waiting for the child. Another variable can store the time it took for the fork() exec() of the fork server to start the child. If the fork() exec() was still executing when the parent task started waiting, then the call is an inefficient remote call.

Another time variable that should be monitored is the execution time of the child function excluding all remote call overhead. In other words, the time it would take for a function to execute if it were not remote. This time can be compared to the total time necessary to send and receive a message to start and finish a remote

function. The sending and receiving messages are not handled in parallel so if that time takes longer than the time it would take to execute the function locally, the remote function call does not provide a net gain in execution speed.

6.3.6 Dynamic Linkage

Because TDP generated programs have aligned virtual memory, dynamic linking activities can be much more efficiently achieved. When code is dynamically linked into an application, only the code pointers must be relocated, all data pointers will be valid for all program tasks.

TDP also makes it easy to redefine a code address because remote functions are called using a pointer variable in the data section. This function pointer variable can be reassigned whenever a different dynamically linkable object is to be called.

Ideally, decision logic can be added to the TDPcall() function to allow it to

optionally call a function remotely or dynamically link it into the parent task depending on run time conditions for the system and application.

6.3.7 Full Code Duplication

The time necessary to load a child task can sometimes be significant. If performance is of the utmost priority and the necessary system capabilities are provided, a much more efficient scheme can be employed.

The program can be processed by TDP as usual but instead of linking in dummy declarations for each remote function, all tasks can be linked identically using the same code so that identical load modules are produced for each task. Logic can be added to the TDPcall() function that would allow it to call the specified function remotely (starting another process) or locally (by calling within the process), depending on run time conditions for the system and application. Run time conditions that TDPcall() might ask the operating system:

Are there any more unoccupied processors available on this machine?

Is there a processor that is less loaded than the current one?

For systems that allow it, the text section in every task can be shared with the other tasks because the text section in this case would be identical in every task. If sharing the text section is not possible, many systems allow the text section to remain in memory by setting a bit in the file permissions or by making a call to the operating system the first time the program is executed. In a distributed system where each CPU has its own controlling operating system, a task would only have to be loaded once when it was first executed. Alternatively each CPU could preload the application when the program begins. Upon starting up, a task would only have to "attach" to the data section if shared physical memory is used. The only thing that would have to be loaded would be the stack for the task process and static data necessary for the TDP run time structures.

6.4 Related Thesis Topics for the Future

6.4.1 Automatic Task Generator

The conversion of single process C programs to multiple task C programs can theoretically be done automatically [BOK,EVA,KIL,WAI,WAN]. A program that can detect parallelism in a C program and calculate the potential for efficiency using parallelism would help the software transition to multiprocessor systems.

6.4.2 Portable Multitasking Languages

UNIX runs on most computing hardware, and TDP programs can run on single processor as well as multiprocessor UNIX systems. Because of the portability of TDP, anything that generates TDP compatible code, essentially becomes ''hardware independent.'' Writing multitasking languages or code generators that use TDP as an object language, allows them to produce software for all UNIX systems. The implementation of multitasking languages that use TDP as an object language, will be an important step in improving the popularity and utility of these languages.

7 BIBLIOGRAPHY

[AHU] Ahuja S., Carriero N., and Gelernter D., Linda and friends. Computer Vol. 19 No. 8 (Aug. 1986) 26-34

[AND] Andrews G. R., Olsson R. A., Coffin M., Elshoff I., Nisen K., Purdin T., and Townsend G., An Overview of the SR Language and Implementation., ACM Trans. Prog. Lang. and Sys. Vol. 10 No. 1 (Jan. 1988) p51-86

[BAC] Bach M.J. and Buroff S.J. Multiprocessor UNIX Operating Systems. AT&T Bell Labs Technical Journal Vol. 63 No. 8 Oct. 1984

[BIN] Binding C., Cheap Concurrency in C., ACM SIGPLAN Notice 20, 9 (Sept. 1985) 21-26.

[BIR] Birrel Andrew D., and Nelson Bruce Jay, Implementing Remote Procedure Calls., ACM Trans. Comput. Syst. Vol. 2 No. 1 (Feb. 1984) p39-59

[BOA] Boari M. et al., Multiple - Microprocessor Programming Techniques: MML, a New set of Tools, IEEE Computer (Jan 1984), pp. 47-59

[BOD] Bodenstab D.E., Houghton T.F., Kelleman K.A., Ronkin G., and Schan E.P., UNIX Operating System Porting Experience. AT&T Bell Labs Technical Journal Vol. 63 No. 8 Oct. 1984

[BOK] Bokhari S.H., Partitioning Problems in Parallel Pipelined, and Distributed Computing. IEEE Trans. Comput., vol. 37 No. 1 (Jan. 1988) 48-58

[COR] Cormack G. V., Short Communication - A Micro-Kernel for Concurrency in C, Softw. Pract. Exper. Vol 18(5) (May 1988) 485-491

[DIE1] Dietz H. and Klappholz D., Refining a conventional language for race-free specification of parallel algorithms. Proc. of the 1984 Internat. Conf. on Parallel Processing

[DIE2] Dietz H. and Klappholz D., Refined C: A Sequential Language for Parallel Programming. Proc. of the 1985 Internat. Conf. on Parallel Processing

[EVA] Evans D.J. and Williams S.A., Analysis and detection of parallel processable code, The Computer Journal vol. 23 No. 1 Oct. 1978.

[FAL] Falcone J.R., A Programmable Interface Language for Heterogeneous Distributed Systems ACM 5,4 (Nov. 1987) 330-351

[GEH] Gehani N. and Roome W., Concurrent C., Softw. Pract. Exper. 16, 9, (Sept. 1986)

[GLA1] Glasser N.E., Interface Language for C with Remote Channels. LCS Programming Systems Research Memo #9, MIT (Jan 1987)

[GLA2] Glasser N.E., The Remote Channel System. thesis Master of Science MIT (June 1987)

[HAM] Hamilton M.A., On the Pragmatic Design of a Remote Procedure Call System. thesis Master of Science MIT (June 1985)

[KER] Kernighan Brian W., Ritchie Dennis M., The C Programming Language, Prentice Hall Software Series, 1978.

[KIL] Kilian Michael F., A Conditional Critical Region Pre-processor for C Based on the Owicki and Gries Scheme. ACM SIGPLAN Notice 20, #4 April 1985. p42-56.

[LIS] McCarthy, J., Et Al. LISP 1.5 Programmer's Manual, M.I.T. Press, Cambridge MA, 1962

[LAP] Lapin J.E., Portable C and UNIX System Programming. Rabbit Software, Prentice-Hall Inc. 1987.

[MUN] Mundie D.A. and Fisher D.A., Parallel Processing in Ada, Computer,

Aug. 1986. pp. 20-25

[NEL1] Nelson B.J., Remote Procedure Call Technical Report. CSL 81-89, Xerox Palo Alto Research Center (1981)

[NEL2] Nelson B.J., Remote procedure call. Ph.D. dissertation, Camegie-Mellon University, Pittsburgh, Pa., 1981

[REF] Reference Manual for the Programming Language Ada, U.S. Department of Defense, ANSI/MIL-STD-1815-A, 1983.

[RUS] Russel Channing H., Waterman Pamela J., Variations on UNIX for Parallel-Processing Computers. Comm. of the ACM Vol. 30 No. 12 Dec. 1987.

[SON] Sonnenschein M., An extension of the language C for concurrent programming. Parallel Comput. 3, 1, (March 1986).

[TSU] Tsujino Y., Audo M., Araki T. and Tokura N., Concurrent C: A programming language for distributed multiprocessor systems. Software - Practice and Exper. 14(11) (1984) 1061-1078

[UNI] UNIX System V Release 3.0 INTEL 80286/80386 Computer Version. Programmer's Guide. AT&T. 1987.

[WAI] Wainwright Rojer L., Deriving Parallel Computations from Functional Specifications: A Seismic Example on a Hypercube., International Jour-

nal of Parallel Programming Vol. 16, No. 3 1987

[WAN] Wang L.L. and Tsai W.H., Optimal assignment of task modules with precedence for distributed processing by graph matching and state-space search., BIT 28 1988 54-68

APPENDIX A

Example Program - Input to TDP

SOURCE FILE 1 (main.c)

```
int globnum = 5;


main(argc,argv)
int argc;
char *argv[];
{
        static int statnum = 6;
        int localnum = 7;
        int i;
        unsigned long retval;
        long inst1,inst2,inst3,inst4;
        char *prstring = "I'm BACK! retval = %ld\n";

        for (i=1;i<argc;i++)
        printf(" %s",argv[i]);

        printf("\n");

        inst1 = doit("AAAAAAAAAA",globnum,statnum,localnum);

        retval = TDPwait(inst1);
        if (retval < 0)
        exit(1);

        printf(prstring,retval);

        inst2 = doit("BBBBBBBBBB",1,2,3);
        inst3 = doit("CCCCCCCCCC",4,4,4);
        inst4 = doit("DDDDDDDDDD",9,9,9);

        retval = TDPwait(inst2);
        printf(prstring,retval);

        retval = TDPwait(inst3);
        printf(prstring,retval);

        retval = TDPwait(inst4);
        printf(prstring,retval);
```

```
        doit("EEEEEEEEEE",1,2,3);
        doit("FFFFFFFFFF",4,5,6);

}
```

SOURCE FILE 2 (doit.c)

```
#include <stdio.h>

doit(str,num1,num2,num3)
char *str;
int num1,num2,num3;
{
        int i = 0;

        while(str[i])
        {
        printf("%c",str[i++]);
        fflush(stdout);
        sleep(1);
        }

        printf("  %d, %d, %d\n",num1,num2,num3);
        return(getpid());
}
```

APPENDIX B

Example Program - Output of TDP

TASK FILE 1 (TDPtsk1.c)

```
#include "TDPdefs.h"
#include "TDPtypes.h"
#include "TDPstatic.h"



TDPstart(argc,argv)
int argc;
char *argv[];
{

        extern int TDPstat1;

#define statnum TDPstat1

        int localnum = 7;
        int i;
        unsigned long retval;
        long inst1,inst2,inst3,inst4;
        char *prstring = &TDP.string[107];

        for(i=1;i<argc;i++)
        printf(&TDP.string[131],argv[i]);

        printf(&TDP.string[135]);

        inst1 = TDPcall(1,&TDP.string[137],globnum,statnum,localnum);

        retval = TDPwait(inst1);
        if(retval < 0)
        TDPusrexit(1);

        printf(prstring,retval);

        inst2 = TDPcall(1,&TDP.string[148],1,2,3);
        inst3 = TDPcall(1,&TDP.string[159],4,4,4);
        inst4 = TDPcall(1,&TDP.string[170],9,9,9);

        retval = TDPwait(inst2);
```

```
                printf(prstring,retval);

                retval = TDPwait(inst3);
                printf(prstring,retval);

                retval = TDPwait(inst4);
                printf(prstring,retval);

                TDPcall(1,&TDP.string[181],1,2,3);
                TDPcall(1,&TDP.string[192],4,5,6);


#undef statnum
TDPexit(0);
}

long
doit(){}
TDPdummy()
{
                printf();
                fflush();
                sleep();
                getpid();
}
```

TASK FILE 2 (TDPtsk2.c)

```
#include "TDPdefs.h"
#include "TDPtypes.h"
#include "TDPstatic.h"
typedef struct {

        int_cnt;
        unsigned char*_ptr;

        unsigned char*_base;
        char_flag;
        char_file;
} FILE;

extern FILE _iob[20];

doit(str,num1,num2,num3)
char *str;
int num1,num2,num3;
{
        int i = 0;

        while(str[i])
        {
        printf(&TDP.string[203],str[i++]);
        fflush((&_iob[1]));
        sleep(1);
        }

        printf(&TDP.string[206],num1,num2,num3);
        return(getpid());
}
TDPstart(){}

TDPdummy()
{
        printf();
        fflush();
        sleep();
        getpid();
}
```

STATIC DATA FILE (TDPstatic.c)

```
#include <stdio.h>
#include "TDPtypes.h"
#include "TDPdefs.h"
char TDPstrdata[] = {
  '\0',
  '\0',
  'T','D','P',' ','r','u','n','t','i','m','e',' ','e','r','r','o','r',' ',
  'a','t',' ','l','o','c','a','t','i','o','n',':',' ','%','d',',',' ',
  'e','r','r','n','o',' ','=',' ','%','d','\n','\0',
  '%','d',' ','%','d',' ','%','d',' ','%','l','d','\0',
  '%','s',' ','%','d',' ','%','d',' ','%','d',' ','%','s','\0',
  '%','d',' ','%','s',' ','%','d',' ','%','l','d',' ','%','s',' ','%','s','\0',
  'T','D','P','r','c','v','a','r','g','s','\0',
  'I','\'','m',' ','B','A','C','K','!',' ','r','e','t','v','a','l',' ','=',' ','%','l','d','\n','\0',
  ' ','%','s','\0',
  '\n','\0',
  'A','A','A','A','A','A','A','A','A','A','\0',
  'B','B','B','B','B','B','B','B','B','B','\0',
  'C','C','C','C','C','C','C','C','C','C','\0',
  'D','D','D','D','D','D','D','D','D','D','\0',
  'E','E','E','E','E','E','E','E','E','E','\0',
  'F','F','F','F','F','F','F','F','F','F','\0',
  '%','c','\0',
  ',',',',',',',','%','d',',',',',' ','%','d','\n','\0',
};
int globnum=5;
int TDPstat1=6;
```

UNSHARED STATIC DATA USED BY TDP (TDPrundata.c)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/signal.h>
#include <sys/msg.h>
#include "TDPtypes.h"
#include "TDPdefs.h"
extern char TDPstrdata[];
extern int TDPstart();
extern int doit();
struct TDPfuncdescr TDPlistdata[TDP_NUMFUNCS] = {
{"TDPtsk1","TDPstart",TDPstart,(sizeof(int ) +
(sizeof(char *) / sizeof(int)) + ((sizeof(char *) % sizeof(int)) > 0)
+ 0)},
{"TDPtsk2","doit",doit,(
(sizeof(char *) / sizeof(int)) + ((sizeof(char *) % sizeof(int)) > 0)
+ sizeof(int ) + sizeof(int ) + sizeof(int ) + 0)}
};
int TDPargsmem[TDP_MAXARG_WORDS];
struct TDPinst TDPinstmem[TDP_NUMFUNCS][TDP_MAXINST] = {0};
struct TDPfuncstruct TDPrundata[TDP_NUMFUNCS] = {
        { TDPinstmem[0] },
        { TDPinstmem[1] },
};
struct TDPrunstruct TDP = {
        TDPQUEUE,0,TDP_MAXMSG,TDP_MAX_PROCESSES,
        TDPDEBUG,TDP_START_TASK,TDPSIGNAL,
        TDP_MAXINST,TDP_MAXARG_WORDS,TDP_NUMFUNCS,0,0,-1,
        TDPrundata,
        TDPlistdata,
        TDPstrdata,
        TDPargsmem
};
```

ROUTINES TO FACILITATE SHARED MEMORY (TDPshmem.c)

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/errno.h>
#include "TDPtypes.h"
#include "TDPdefs.h"
#include "TDPstatic.h"
extern int errno;
extern char TDPstrdata[221];

TDPreceive_shared_data(fname)
char *fname;
{
        int fd;

        errno = 0;
        if ((fd = open(fname,O_RDONLY)) < 0)
        TDPfatal_err(100,errno);
        if (read(fd,&globnum,sizeof(globnum)) != sizeof(globnum))
        TDPfatal_err(101,errno);
        if (read(fd,&TDPstat1,sizeof(TDPstat1)) != sizeof(TDPstat1))
        TDPfatal_err(102,errno);
        if (read(fd,TDPstrdata,sizeof(TDPstrdata)) != sizeof(TDPstrdata))
        TDPfatal_err(103,errno);

        close(fd);

}

TDPsend_shared_data(fname)
char *fname;
{
        int fd;

        errno = 0;
        if ((fd = open(fname,O_RDWR | O_CREAT | O_EXCL, 0666)) < 0)
        TDPfatal_err(104,errno);
        if (write(fd,&globnum,sizeof(globnum)) != sizeof(globnum))
        TDPfatal_err(105,errno);
        if (write(fd,&TDPstat1,sizeof(TDPstat1)) != sizeof(TDPstat1))
        TDPfatal_err(106,errno);
        if (write(fd,TDPstrdata,sizeof(TDPstrdata)) != sizeof(TDPstrdata))
        TDPfatal_err(107,errno);
```

```
        close(fd);

}


TDPaccept_shared_data(oldfname,newfname)
char *oldfname,*newfname;
{
        int fdold,fdnew;

        errno = 0;
        if ((fdold = open(oldfname,O_RDONLY)) < 0)
        TDPfatal_err(108,errno);

        if ((fdnew = open(oldfname,O_RDONLY)) < 0)
        TDPfatal_err(109,errno);
        TDPreplace_mem(TDPDFLT_REPLACE,fdold,fdnew,&globnum,
        sizeof(globnum));
        TDPreplace_mem(TDPDFLT_REPLACE,fdold,fdnew,&TDPstat1,
        sizeof(TDPstat1));
        TDPreplace_mem(TDPDFLT_REPLACE,fdold,fdnew,TDPstrdata,
        sizeof(TDPstrdata));

        close(fdold);

        close(fdnew);

}
```

TDP DEFINED TYPES (TDPtypes.h)

```
/* THE VALUES IN THIS FILE SHOULD NEVER BE CHANGED UNLESS THE
  TDP LIBRARY CAN BE RECOMPILED.
*/
#define TDP_FILLALL 0
#define TDP_FILLTOUCHED_VAR 1
#define TDP_FILLTOUCHED_BYTES 2
#define TDP_FILLTOUCHED_BITS 3
#define STR1OFF 0
#define STR2OFF 1
#define STR3OFF 2
#define STR4OFF 49
#define STR5OFF 62
#define STR6OFF 77
#define STR7OFF 96
struct TDPtermmsg {
                long mtype;
                int pid;
                int exit_code;
                };
struct TDPfuncdescr {
                char *tskname;
                char *funcname;
                int (*funcptr)();
                int numarg_words;
};
struct TDPinst {
                unsigned longretval;
                intrunning;
};
struct TDPfuncstruct {
                struct TDPinst *instance;
                intnumarg_words;
                intlastinst_returned;
                unsigned longlast_retval;
                intnum_running;
};
struct TDPrunstruct {
                int queue_key;
                int queue;
                int max_msg;
                int max_processes;
```

```
        int debug_flag;
        char *start_name;
        int signal_num;
        int max_inst;
        int max_args;
        int num_functions;
        int holdsig;
        int numclds_running;
        int lastfunc_retumed;
        struct TDPfuncstruct *runlist;
        struct TDPfuncdescr *funclist;
        char **string;
        int **funcargs;
};
```

PROGRAM SPECIFIC DEFINITIONS (TDPdefs.h)

```
#define TDP_MAXARG_WORDS 8
#define TDP_MAXMSG 256 + (TDP_MAXARG_WORDS * sizeof(int))
#define TDPQUEUE 1
#define TDPSIGNAL SIGUSR1
#define TDP_NUMFUNCS 2
#define TDP_MAXINST 15
#define TDP_MAX_PROCESSES 15
#define TDPDFLT_REPLACE TDP_FILLTOUCHED_BYTES
#define TDPDFLT_TMPDIR 0
#define TDPDFLT_PREFIX 1
#define TDPDEBUG 0
/* The following #defines should only be used by the fork server */
#define TDP_START_TASK "TDPtsk1"
#define TDP_NUMARGS 5
```

TDP GENERATED MAKE FILE FOR PROGRAM (TDPprogname.mk)

```
TYPE=I386
CFLAGS=

progname: TDPalign TDPtsk1 TDPtsk2 TDPforkserv
        rm -f progname
        ln TDPforkserv progname
        TDPalign TDPtsk1 TDPtsk2
TDPtsk1: TDPstatic.h TDPstatic.o TDPrundata.o TDPshmem.o TDPtsk1.o
        cc -r -o TDPtsk1 TDPstatic.o TDPrundata.o TDPshmem.o TDPtsk1.o libTDP.
TDPtsk1.o: TDPtsk1.c TDPstatic.h TDPdefs.h TDPtypes.h
        cc ${CFLAGS} -c TDPtsk1.c
TDPtsk2: TDPstatic.h TDPstatic.o TDPrundata.o TDPshmem.o TDPtsk2.o
        cc -r -o TDPtsk2 TDPstatic.o TDPrundata.o TDPshmem.o TDPtsk2.o libTDP.
TDPtsk2.o: TDPtsk2.c TDPstatic.h TDPdefs.h TDPtypes.h
        cc ${CFLAGS} -c TDPtsk2.c
TDPforkserv: TDPservdata.o TDPforkserv.o
        cc -o TDPforkserv TDPforkserv.o TDPservdata.o
TDPalign: TDPalign.o
        cc -o TDPalign TDPalign.o -lld
TDPservdata.o: TDPservdata.c TDPdefs.h TDPtypes.h
        cc ${CFLAGS} -c TDPservdata.c
TDPstatic.o: TDPstatic.c TDPdefs.h TDPtypes.h
        cc ${CFLAGS} -c TDPstatic.c
TDPrundata.o: TDPrundata.c TDPdefs.h TDPtypes.h
        cc ${CFLAGS} -c TDPrundata.c
TDPshmem.o: TDPshmem.c TDPstatic.h TDPdefs.h TDPtypes.h
        cc ${CFLAGS} -c TDPshmem.c
TDPforkserv.o:
        ar x libTDP.a TDPforkserv.o
TDPalign.o:
        ar x libTDP.a TDPalign.o
```

APPENDIX C

Running the Example Program

The example program was built with and without TDP processing. The program was then run with the following invocation:

progname these are the args >outfile

This was done for the version of ''progname'' that was built without TDP and for the version built using TDP.

SAMPLE PROGRAM OUTPUT WITHOUT TDP

```
 these are the args
AAAAAAAAAA  5, 6, 7
I'm BACK! retval = 13569
BBBBBBBBBB  1, 2, 3
CCCCCCCCCC  4, 4, 4
DDDDDDDDDD  9, 9, 9
I'm BACK! retval = 13569
I'm BACK! retval = 13569
I'm BACK! retval = 13569
EEEEEEEEEE  1, 2, 3
FFFFFFFFFF  4, 5, 6
```

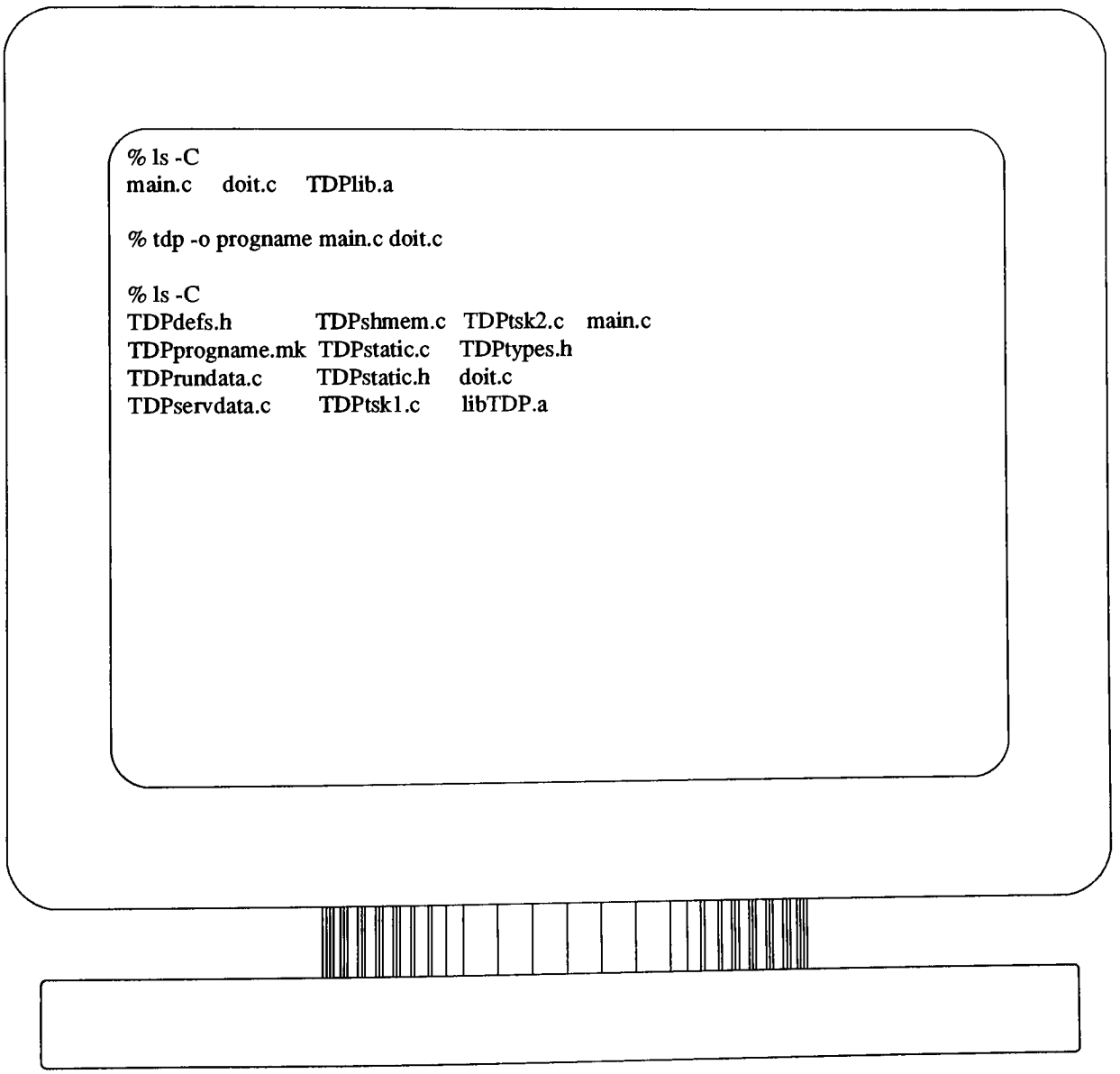
SAMPLE PROGRAM OUTPUT FOR TDP PROCESSED VERSION

```
AAAAAAAAAA  5, 6, 7
BCBDCBCDBCDBCDBCDBCDDBCBCDBCDD  4, 4, 4
  1, 2, 3
  9, 9, 9
 these are the args
I'm BACK! retval = 13547
I'm BACK! retval = 13548
I'm BACK! retval = 13549
I'm BACK! retval = 13550
EFFEFEFEFEFEFEFEFEFE  4, 5, 6
  1, 2, 3
```

APPENDIX D

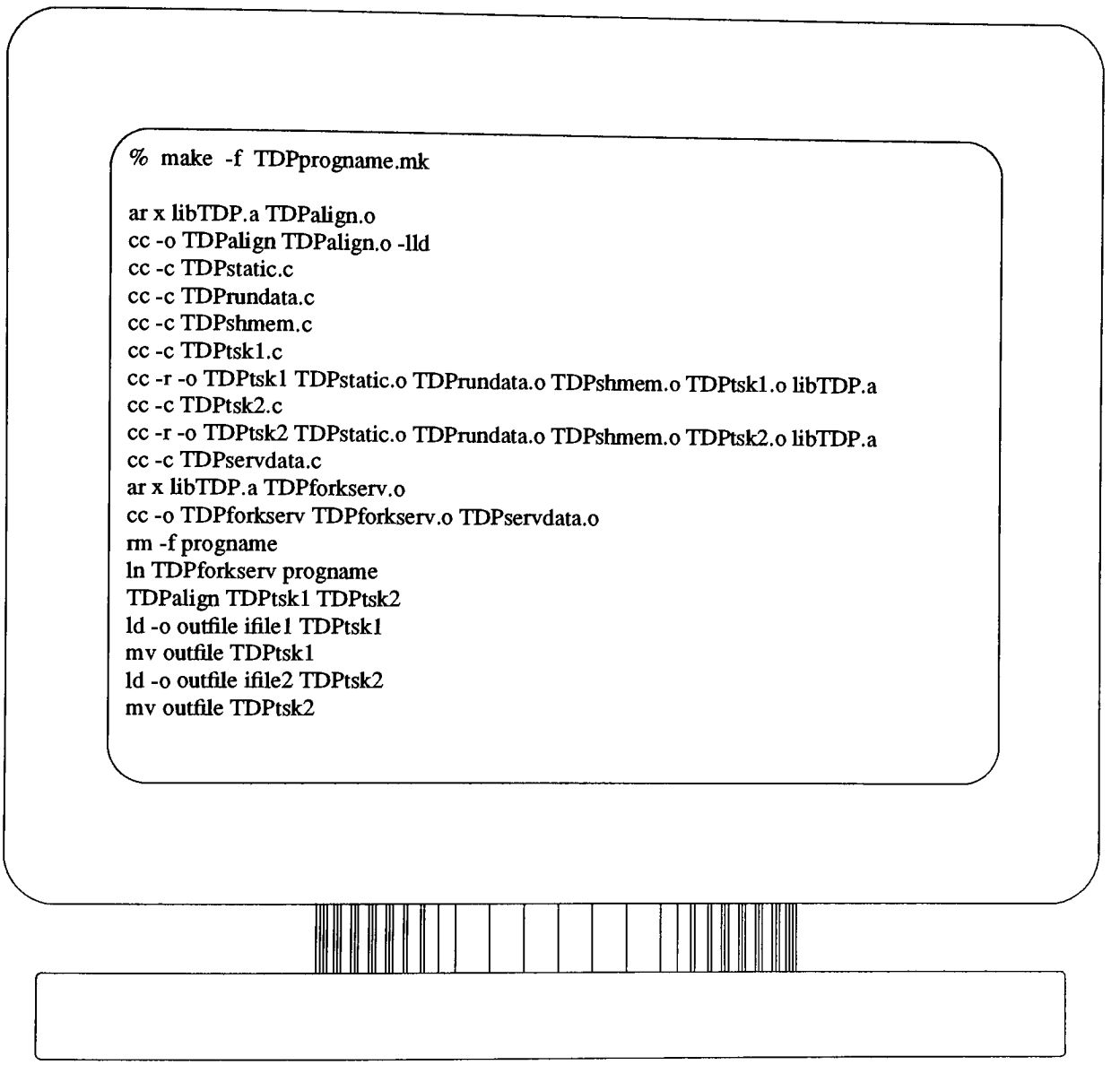
Walk-Through for Example Program
RUNNING THE PREPROCESSOR

The following view shows the appearance of the current working directory containing the example program. A directory listing is made before and after invocation of TDP showing what files are created by TDP. The contents of the files main.c and doit.c are listed in Appendix A. The contents of the files that have names beginning with "TDP" are listed in Appendix B. The TDP run time library is in the archive file: libTDP.a.



BUILDING THE LOAD MODULES

The following view shows the output to the screen when the TDP produced *make* file is invoked.



```
% make -f TDPprogname.mk

ar x libTDP.a TDPalign.o
cc -o TDPalign TDPalign.o -lld
cc -c TDPstatic.c
cc -c TDPrundata.c
cc -c TDPshmem.c
cc -c TDPtsk1.c
cc -r -o TDPtsk1 TDPstatic.o TDPrundata.o TDPshmem.o TDPtsk1.o libTDP.a
cc -c TDPtsk2.c
cc -r -o TDPtsk2 TDPstatic.o TDPrundata.o TDPshmem.o TDPtsk2.o libTDP.a
cc -c TDPservdata.c
ar x libTDP.a TDPforkserv.o
cc -o TDPforkserv TDPforkserv.o TDPservdata.o
rm -f progname
ln TDPforkserv progname
TDPalign TDPtsk1 TDPtsk2
ld -o outfile ifile1 TDPtsk1
mv outfile TDPtsk1
ld -o outfile ifile2 TDPtsk2
mv outfile TDPtsk2
```

RUNNING THE RESULTANT MULTI-TASK PROGRAM

The following view shows the TDP produced multi-task program in action. An explanation for the output can be derived by looking at the source code listings in Appendix A and B. Appendix C gives a comparison of output for the program as a multi-task and single task program.