1990

Xinu on the Transputer

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Victor Norman

Report Number:
90-1024

XINU ON THE TRANSPUTER

Douglas Comer
Victor Norman

CSD-TR-1024
September 1990

Xinu on the Transputer

Douglas Comer
Victor Norman

Computer Science Department
Purdue University
West Lafayette, IN 47907
CSD-TR-1024

September 21, 1990

Abstract

This paper reports on an effort to port the Xinu operating system to the INMOS transputer. Because the transputer has an unconventional architecture, it requires novel implementation techniques. To take advantage of the transputer's hardware, we introduced new features into the Xinu implementation, while still preserving all of Xinu's original functionality and programming interface. This paper reviews the INMOS transputer and the Xinu operating system. It then describes the problems encountered in this implementation, our solutions, and some implementation details.

1 Introduction

This paper reports on an effort to port the Xinu operating system to the INMOS transputer. The port provided two major challenges. The first was in implementing Xinu's multiple priority scheduling scheme on the transputer's dual priority scheduling hardware. The second was in simulating interrupts and the interrupt handling used in Xinu (and most conventional operating systems), because the transputer does not fully support interrupts in the hardware. This paper includes a discussion of the transputer hardware, an introduction to the Xinu operating system, new ideas used in the Xinu implementation, and implementation details.

2 An Introduction to the Transputer

The transputer is a high-speed microcomputer developed by INMOS, Ltd. that contains an unconventional architecture [INM88]. The design of the transputer allows a process to communicate with a remote process on a connected transputer as easily as with a process on the local transputer.

The T800 model transputer includes, on a single chip, a 32-bit processor, 4K RAM, 4 high-speed serial communication links, an external memory controller, a floating point processor, and two timers. The 4 communication links operate at 20 Mbps and allow a pair of transputers to communicate.

A few key features of the transputer make it an interesting and unconventional machine. First, the transputer uses a stack of three registers instead of the typical array of registers found in most architectures. This stack of registers holds operands for many operations and holds values when a function call returns.

Second, the transputer executes a microcode-embedded, rudimentary operating system. This operating system understands the concepts of processes, ready queues, process priorities, context switching and time-slices, and timers. A stack pointer and an instruction pointer define a transputer process – no registers or status words need to be saved at a context switch. The transputer uses the area below each process'

stack as a process control block, saving the instruction pointer and other important information there at predefined offsets from the stack pointer.

Transputer processes execute at one of only two priorities: high priority or low priority. By convention, most processes run at low priority, with high priority reserved for special tasks. With each priority is associated a ready list that holds ready processes waiting to execute. The differences between high and low priority are important. First, a high-priority process always gets the CPU immediately when it becomes ready (if no other high priority process is executing at that time). Thus, all low-priority processes wait until no high-priority processes are ready. Second, high-priority processes are not time-sliced, as are low-priority processes. A high-priority process that begins execution does not relinquish the CPU until it deschedules itself, explicitly or implicitly. A process may deschedule itself in one of two ways: by either explicitly executing a descheduling action or by blocking on I/O or on a timer. Thus, a high-priority process may monopolize the CPU by avoiding all descheduling actions. On the other hand, a low-priority process cannot monopolize the use of the CPU, but instead executes only for a time-slice given by the operating system. Also, as one would expect, low-priority processes do not execute if a ready, high-priority process exists.

A round-robin scheduling algorithm schedules low-priority processes on the ready queue. Each process executes in a time-slice of approximately 2 milliseconds. When a low-priority process' time-slice expires, the transputer operating system places the process on the end of the low-priority queue and schedules the first process in the queue for execution. This context switch requires only 1 microsecond. A low-priority process is descheduled after expiration of a time-slice only after execution of instructions that clear the register stack. Thus, the register stack does not need to be saved along with the stack pointer and instruction pointer at each context switch.

The transputer also has two timers and timer queues. One timer and timer queue is associated with high-priority processes, the other pair with low-priority processes. The high-priority timer updates a clock register every 1 microsecond, while the low-

priority timer ticks every 64 microseconds. Processes may delay themselves using these timers and be restarted and rescheduled when their delay time expires.

Third, the transputer provides little support for interrupts or interrupt-driven events. Most conventional operating systems rely heavily on the existence of hardware support for interrupts. For example, on conventional architectures, the system starts data transfer by by placing an address in the device's control register. Then, to signal completion of the data transfer, the device interrupts the CPU. This method of data communication is essentially asynchronous because the CPU continues to execute while the I/O device transfers data. However, all communications on the transputer occur using the synchronous I/O instructions, IN and OUT. These instructions deschedule the currently executing process, perform the data transfer, and then reschedule the process upon completion of the transfer. Thus, the transputer scheme essentially produces a blocking, synchronous communications interface. Thus, processes must be used to handle in-coming and out-going communication.

Fourth, the transputer has 4 full-duplex, 20 Mbps communication links. These links allow groups of transputers to be easily interconnected. Each link sends data and acknowledges data upon receipt. Special instructions make communication on a link simple and straightforward.

Fifth, the instruction set of the transputer is unconventional. It consists of 16 *direct* functions and approximately 100 *indirect* functions. The direct functions are the the most commonly used instructions. Each direct function is specified in 4 bits, with its operand filling the other 4 bits, allowing many of the operations with their operands to be stored in 1 byte.

Sixth, the transputer contains a built-in notion of inter-process communication, based on the IN and OUT instructions mentioned above. These instructions initiate inter-process communication over a *channel*, which is a communications port through which two processes transfer data. The communication may be between processes either on the same machine or on different machines. If the processes are on different machines, the channel is the address of one of the transputer's high-speed links (a

hard link); otherwise, the channel is the address of a predetermined word in memory (a *soft* link). The IN and OUT instructions are both synchronous instructions, so that a process does not return from these instructions until all communication over the channel completes.

The transputer supports both on-chip memory and off-chip memory. Access time to on-chip (cache) memory is substantially less than access time to off-chip (external) memory. Thus, efficient use of this cache memory can significantly reduce execution time of programs. Most transputers are equipped with 2 to 4KB of cache memory, and 2 MB of external memory, although the address space can accommodate up to approximately 4GB of external memory.

The transputer hardware offers no support for virtual memory, memory protection, or operating system modes. This lack of memory management hardware could lead to significant implementation problems for some operating systems. Fortunately, Xinu Version 7 does not require these hardware memory management facilities.

3 An Introduction to Xinu

The Xinu operating system [Com84], developed by Douglas Comer in 1980, is a small, elegant operating system, originally for use on "small" machines, such as the Digital Equipment Corporation's LSI-11, with 64 KB of memory. Xinu is a conventional, process-based operating system, that includes all of the basic parts of an operating system such as memory management, process management, process coordination and synchronization, interprocess communication, real-time clock management, and device drivers. Xinu Version 7 [Com87], which we implemented on the transputer, also includes inter-machine communication, and a file system.

Xinu differs from other operating systems in three important aspects. First, the Xinu kernel and all user processes execute in the same contiguous address space. Thus, user code may easily access kernel variables and will incur low overhead for system calls. Second, Xinu does not offer memory protection for user or kernel routines. No distinction exists between kernel calls and user procedure calls. Third, Xinu Version

7 does not distinguish between "kernel" mode or "user" mode, as do many operating systems.

The process scheduling policy of Xinu is a priority, round-robin scheme, in which the highest priority ready process always obtains the CPU. When more than one process with highest priority becomes ready, the time-slicing algorithm allows fair access to the CPU. The single process that has the CPU at any one time is called the *current* process. Processes may have any positive integer priority, although usually 0 to 100 is used, with 0 being the lowest priority.

4 New Concepts Added to Xinu

The strengths of the transputer lie in its ability to context switch processes very quickly and in its ability to communicate rapidly over its links. We wanted to take advantage of these strengths, while still preserving the original functionality, policies, and interaction of the components of Xinu. We found that, while most of Xinu fits well on the transputer architecture, some features of the transputer architecture force us to add new concepts to Xinu. The problems we faced in the implementation arise because the transputer's notion of a process differs from Xinu's notion of a process, especially with regard to a process' priority, and, the transputer does not support the notion of interrupts.

4.1 Xinu Processes vs. Transputer Processes

To make effective use of the transputer's notions of processes, ready queues, context switches, and time-slices, we had to map Xinu's concept of a process onto the transputer concept. Our implementation maps the Xinu multi-level priority queue onto the transputer low-priority queue by manually manipulating the transputer ready queues. At each point in execution Xinu provides the machine with only the set of processes that have highest priority. This solution works in the Xinu environment because all events that cause a new process to become ready also evaluate the state of the system, causing a context switch to occur if and only if the new ready process has higher

priority than the currently executing processes. Thus, our system preserves the Xinu policy that states that the highest priority ready process always has the CPU. This context switch may require a substantial amount of hardware queue manipulation: the current process and the ready processes on the transputer are removed and inserted by priority into the Xinu ready queue. Then, the highest priority processes in this queue are put back onto the transputer ready queue and allowed to execute. We refer to this type of context switch as a *heavy* context switch.

To reduce the number of occurrences of these expensive, heavy context switches, the transputer implementation of Xinu supports a new concept of the current process. In previous implementations of Xinu, only one current process existed at any one time. This notion changed, so that now, at each context switch, a *set* of processes becomes current. This current set of processes is then scheduled on the transputer low-priority queue. The current set always consists of all ready processes with equal and highest priority, along with all processes that have the special permanent priority (see below). In doing so, the transputer queue contains multiple processes ready to execute, thus reducing the number of heavy context switches required by the system. We found that this solution works well, and does not change the functionality, policies, or behavior of Xinu, but still makes use of the transputer's ability to perform *light* context switches, automatically time-slicing between ready processes waiting on the low priority queue.

4.2 Interrupt-driven Events

In many conventional operating systems as well as in Xinu, all low-level I/O operations are handled by interrupt-driven events. Additionally, a clock interrupt handler normally manages all timing events. However, the transputer does not directly support the notions of interrupts or interrupt-driven events.

Our implementation uses processes to handle low-level I/O events and clock events. These processes operate like all other processes, except that they execute in the transputer's high-priority mode. Being in high-priority mode allows them to service events immediately without having to wait for the CPU. To guarantee that these processes

remain scheduled and ready to handle interrupt events, we designate these processes as *permanent* processes. Permanent processes are processes that have a special priority such that they always remain in the set of current processes. This special priority may be thought of as a floating priority, that always matches the priority of the highest, ready, normal process. In other words, these processes remain among the processes put onto the transputer to execute. Without permanent processes, the system would have the problem of either never allowing normal processes to become current or having one of the interrupt-handling processes descheduled and left ready on the Xinu ready queue. Neither of these conditions is acceptable and our implementation prevents both circumstances from occurring.

5 Implementation Details

Here we present more details of interesting aspects of the implementation of Xinu on the transputer.

5.1 Disabling Interrupts

Other implementations of Xinu rely heavily on the ability to disable interrupts in order to enforce atomic execution of critical sections of code. Because the transputer has no interrupts to disable, a process simulates the disabling of interrupts by moving itself from the transputer's low to high priority queue. Conversely, a process restores interrupts by moving itself back down to the low priority queue. Correct coding of these actions is quite tricky, as moving from the low to high priority queue leaves a reference to the interrupted low priority process in special registers that must be cleared.

5.2 Context Switching

As previously discussed, context switches occur at two different levels in the transputer implementation of Xinu. First, the transputer performs *light* context switches between processes that it has on its ready queues, thus time-slicing the CPU between

equal priority processes. Second, the Xinu operating system performs *heavy* context switches whenever the set of current processes has potentially changed. We explain in detail here the implementation of this *heavy* context switch.

When a process running on the transputer initiates an action that could cause the set of highest priority processes to change, a Xinu context switch occurs. Examples of these initiating actions are a call to *kill* a process, *resume* a process, or *suspend* a process. The Xinu context switch begins by determining which processes still wait on the transputer ready queues (these are the processes that are current). The context switch removes these processes from the transputer ready queues and places them in order on the Xinu ready queue. (Placing them in order preserves the fairness rule for processes of equal priority.) The system places permanent processes at the head of the queue. As processes are removed from the transputer queues and placed in the Xinu ready queue, the system keeps a pointer to the last, highest-priority process in the Xinu ready queue. This pointer partitions the ready queue into two sets of processes: those that will be part of the next set of current processes, and the rest. By keeping a pointer into the list, we remove the need to conduct a linear search of the list to find the new current set.

After determining the new set of current processes, the system places the processes on the transputer queues in order. If the system does not find the currently running process (the process that initiated the context switch in the first place) among the new set of current processes, then the currently running process stops itself, and the first process in the new set of current processes begins running.

In order to speed up the execution of the Xinu context switch, the code for the context switch resides in the transputer's low, on-chip memory. This helps to minimize the overall execution time spent in context switching.

5.3 Determining the Current Process Id

In other Xinu implementations, the global variable "currpid" is available for processes to determine the process id of the process that is running. Because the current process

concept has now been expanded to a set of current processes, the system now requires a new method to determine the process id of the process that is actually executing. The system determines the executing process' id by using the stack pointer of the process. To assist in this procedure, memory is divided into two kilobyte pages. The memory page of the value of the stack pointer acts as an index into an array that maps memory locations to process ids. The array is initialized at system boot time and updated at process creation during stack memory allocation. This method could become problematic if a stack overrun occurs. However, because a stack pointer bounds check occurs at each context switch, this method is reliable, safe, and fast.

5.4 Interrupts

As mentioned above, there are no provisions in the transputer hardware for the implementation of interrupts or interrupt-driven events. Thus, to simulate the input, output, and clock interrupts that occur in conventional implementations of Xinu, processes serve to handle such events. At system boot time, three permanent processes are created. One is the input interrupt handler. It executes an infinite loop that continuously waits for characters to arrive on a communication link, and then transfers the characters to an input buffer to be read by user processes.

The second process, the output interrupt handler, is created but not started at boot time. Instead, the handler remains suspended until a process wishes to do output (as signified by the insertion of characters into the output buffer), and it remains active while there are characters waiting to go out.

Like the input interrupt handle, the clock interrupt handler also starts executing at system boot time. It executes an infinite loop; first, putting itself to sleep for 1/10th second on the high priority timer queue, then waking up processes and incrementing a real-time clock counter. The clock interrupt handler no longer tests for the occurrence of a preemption event, because the hardware handles all time-slicing.

5.5 The Kernel Debugging Routine *Kprintf*

Conventional architecture implementations of Xinu use the *kprintf* routine to print messages to the console device. *Kprintf* is a blocking, device-polling implementation of the UNIX *printf* routine. It is an indispensible tool during the porting of the operating system because it allows debugging messages to be printed from with system routines, without calls to output interrupt routines (which might not yet be initialized) and without buffering of characters.

Our implementation of *kprintf* differs from other implementations considerably. Other implementations put out a character, poll the device to find out when the character has been transmitted, and then repeat the steps. No other processes execute while the I/O device transmits the character. Because the transputer communication links transmit characters using only synchronous instructions, it is impossible to poll the device. Also, the transputer deschedules the process doing the communication and allows other processes to execute. Thus, our implementation manipulates the transputer queues directly, by removing the process lists from the transputer's low and high priority queues before transmitting characters, and restoring the lists upon completion. This solution provides the same functionality as previous implementations in that no other processes run while a *kprintf* executes. This is a reliable implementation, although it is not efficient. Thus, the use of *kprintf* should be restricted to kernel debugging, as it forces the transputer to be idle for relatively long periods of time while the link communication completes.

6 Summary

The Xinu operating system ports well to the transputer architecture. The most interesting and difficult task in the project was to successfully integrate the transputer's low-level operating system into the Xinu model. Xinu fits well on the transputer's unconventional architecture, using the fast link communication and fast context switching of the low-level machine, and providing a clean and usable interface for application

programs.

References

[Com84] Douglas Comer. *Operating System Design the XINU Approach.* Prentice-Hall, 1984.

[Com87] Douglas Comer. *Operating System Design, Volume II Internetworking with XINU.* Prentice-Hall, 1987.

[INM88] INMOS Ltd. *Transputer Instruction Set A Compiler Writer's Guide*, 1988.