Multiple programs in one UNIX process Don Libes National Bureau of Standards Bldg 220, Rm A-127 Gaithersburg, MD 20899 (301) 975-3535 {seismo,mimsy}!nbs-amrf!libes A small operating system (XINU) was ported to 4.2BSD. The entire operating system runs as a single UNIX process. The code is approximately 1000 lines of C (including comments) and 6 lines of assembler. All of the code is user-level, and thus presents a system easy to examine, understand, and experiment with further. The code has been used as a base for an application of several cooperating processes communicating through global variables. Alternatively, the system provides semaphores and messages for interprocess communication. Keywords: XINU, UNIX, Operating Systems, Multitasking, Concurrent C, Common Memory, Interprocess Communication Background - Why did we need this This project fell out of a recent porting effort at NBS. The original desire was to move an application from a non-UNIX computer to a UNIX computer. The non-UNIX computer ran a simple home-brewed operating system the details of which are unimportant except that it provided interprocess communication through global variables. While 4.2 promised shared memory, it failed to deliver on this. (This has since been remedied by [Libes 85].) To quote from the manual page for [Joy 83]:

DESCRIPTION N.B.: This call is not completely implemented in 4.2. Taking the cryptic advice, we decided that it might be possible to port the entire operating system and application as a single UNIX process. This proved to be possible with the help of several recent enhancements of 4.2 UNIX including sub-second interval timers and non-blocking I/O. Our first implementation did not require separate process stacks, and we realized that by adding them, we would have a tool of much more generality. Before proceeding much further, we quickly realized the similarity to XINU as presented by [Comer 84]. (Other approaches are discussed by [Kepecs 85] and [Tevanian 87].) XINU "Operating System Design, The XINU Approach" is a book by Douglas Comer. In this book, Comer presents a layered and modular operating system. In contrast to other operating system texts which compare and contrast a variety of algorithms for typically only the most interesting tasks in an operating system, Comer chooses one technique for each problem, usually the most straightforward one or that leading to the simplest presentation. (Alternatives are often proposed in the exercises at the end of each chapter.) The book is further unique in discussing every last aspect of a single implementation. This implementation is XINU. The entire source of XINU is in the text, including the machine dependent code for running XINU on a Digital Equipment Corporation LSI 11/2 (a microcomputer version of the PDP 11). Based on Comer's unusually well-written text, we felt that it might be possible to bring up XINU on top of UNIX. Such a system would be able to provide the concurrency and shared variables that our original application needed, and at the same time be immediately useful to others, since it was already well documented. In fact, it was not hard. Our task was much easier than Comer's in that we had a complete set of device drivers already. This included a file system and terminal interface. We also did not have to worry (much) about operating system startup and C start up. It took approximately 3 hours to type in the necessary parts of XINU. This included process management and utilities for fifo and priority queues. Our initial version of XINU did not use a real-time clock. Processes had to explicitly give up control (through calls to etc). During that time, it was possible to use to switch between processes. saves/restores the registers including the pc and sp (stack pointer), and also the signal mask (used as an interrupt mask). Process rescheduling - resched() XINU processes call (via etc) to temporarily give up control of the processor to a ready process. Here is the code fragment where an old process gives control to a new process. /* _resched.c - reschedule and context switch processes

_resched() { . . /* old process is running */ if (0 == setjmp(optr->pregs)) longjmp(nptr->pregs,1); /* new process resumes here */ . . } Each process has a structure describing its state whenever the process is not currently executing (analogous to the structure in the UNIX kernel). In the above code fragment, points to the new process to begin executing. points to the old process that is going to be suspended. The field is the register save area. All that is necessary to switch processes is to save the current register values in and restore the old register values in saves most of the registers including the pc and sp. However, it does not save the condition codes. This is because and are never immediately followed by a test of the condition codes. Using the 4.2BSD interval timer, we added a real-time clock. The real-time clock could interrupt computations anywhere, including the case where the conditions code had been set but not tested. At this point, it became necessary to do context switches ourselves. That required a small number of assembler statements. The following amended version of (for an MC68000) can be called from interrupt handlers as well as user processes. /* _resched.c - reschedule and context switch processes */

/* Since we can't pass parameters to rte from resched, we use these */ /* variables that are global to both routines. */ static int *new_sp; /* new stack pointer register to be loaded */ static int (*new_pc)(); /* new program counter register to be loaded */ static int new_signal_mask; /* new signal mask to be loaded */

void _resched() { register struct pentry *optr; /* pointer to old process entry */ register struct pentry *nptr; /* pointer to new process entry */

/* no switch needed if current process priority higher than next */ if (((optr = &_proctab[_currpid])->pstate == PRCURR) && (lastkey(_rdytail)<optr->pprio)) return;

/* if the old process was still runnable, mark READY */ if (optr->pstate == PRCURR) {

optr->pstate = PRREADY; _insert(_currpid,_rdyhead,optr->pprio); }

/* remove highest priority process at end of ready list */ nptr = &_proctab[(_currpid = _getlast(_rdytail))]; nptr->pstate = PRCURR; /* mark it currently running */

#ifdef RTCLOCK /* schedule an interrupt for the end of a quantum or the next event */ /* in the sleep queue, whichever is sooner */ _start_itimer((_slnempty && (*_sltop < QUANTUM))?*_sltop:QUANTUM); #endif

/* ctxsw(optr->pregs,nptr->pregs);*/ /* at this point, optr->pregs == a5@, nptr->pregs = a4@ */

/* save all registers in optr->pregs */ asm("moveml #0xffff,a5@"); /* save all the registers */ asm("movl #OLDPROC,a5@(64)"); /* change pc and save it */ optr->signal_mask = sigblock(0); /* save old interrupt reg */

/* we have completed putting the old process to bed */ /* now restart the new process */

/* prepare pc, sp and interrupt mask for rte() to use */ new_sp = nptr->sp; /* movl a4@(60),_new_sp */ new_pc = nptr->pc; /* movl a4@(64),_new_pc */ new_signal_mask = nptr->signal_mask; /* load rest of registers directly except for a7 (sp) */
asm("moveml a4@,#0xfff"); /* restore d0-d7,a0-a3 */
asm("moveml a4@(52),#0x6000"); /* restore a5-a6 */
asm("movl a4@(48),a4"); /* restore a4 */

kill(getpid(),RTE); fprintf(stderr,"resched: kill(,RTE) returned?0);

/* old process returns here */ asm("OLDPROC:"); } Notice that the scheduler here is very simplistic. Highest priority processes are selected round-robin. (More complex schedulers might use per-process quantums as well as reassigning priorities.) /* if the old process was still runnable, mark READY */ if (optr->pstate == PRCURR) { optr->pstate = PRREADY; _insert(_currpid,_rdyhead,optr->pprio); }

/* remove highest priority process at end of ready list */ nptr = &_proctab[(_currpid = _getlast(_rdytail))]; An interval timer is then scheduled to occur at the end of the next quantum or for the first scheduled sleeping process, whichever is sooner. /* schedule an interrupt for the end of a quantum or the next event */ /* in the sleep queue, whichever is sooner */ _start_itimer((_slnempty && (*_sltop < QUANTUM))?*_sltop:QUANTUM); The real-time clock is simulated using the 4.2 interval timer. Rather than generating constant interval clock ticks, the interval timer is only set for known events (i.e. quanta and sleeping processes). This reduces the number of clock interrupts significantly. Context switching - rte() Comer describes (p. 59) the LSI instruction (return from trap) which reloads the pc and ps (processor status register) at the same time. We have a similar problem, although rather than reloading the ps, we want to reload the signal mask, The solution is to artificially provoke a signal (via which at termination executes a (return from exception). This is the 68000's analog to the 11/2's /* rte() - indirectly execute 68K rte (return from exception) instruction */ /* Always called from resched(). This routine is necessary to load the */ /* signal mask at the same time as we load the new pc and sp. */ /* setjmp/longjmp is unusable as it doesn't save/restore all the registers. */

/* ARGSUSED */ static void rte(sig,code,scp) int sig; int code; struct sigcontext *scp; { scp->sc_sp = (int)new_sp; scp->sc_pc = (int)new_pc; scp->sc_mask = new_signal_mask;

/* No need to reload ps, as no one looks at it anyway, upon return. */

A simple example We want two XINU processes to execute simultaneously, one continuously printing "1", and the other continuously printing "2". To do it, we create two subroutines as follows: prog1() { for (;;) printf("1"); }

prog2() { for (;;) printf("2"); } The following subroutine is all that is necessary to run them. user_main() { xresume(xcreate(prog1,2000,20,"prog1",0));
xresume(xcreate(prog2,2000,20,"prog2",0)); } Compiling this together with the XINU support routines and running the executable produces the following output: 11111222221111122222111112222211111222222 . . . . takes a subroutine and creates a runnable (XINU) process, returning a process id. Passing the process id to allows the process to run. The remaining parameters to are the stack size, a process priority, a tag for debugging, and a number and list of arguments passed to the process when started. Further information can be found in Comer's book. Miscellaneous but important implementation notes The entire project took approximately two person-weeks. This included typing in the source, learning the necessary amount of both LSI 11/2 assembler (Comer's original) and a mongrel 68K/UNIX assembler provided by the vendor of our 4.2 system. Lastly, we

had to figure out the undocumented C calling conventions for the 4.2 C compiler (very similar to what Comer discusses) as well as experiment with the undocumented statement in our C compiler. Although we have no references on it, is a keyword in (apparently) many C compilers which allows the user to drop assembler statements into the the C compiler's assembler output. For example, . foo(); asm("jsr bar"); /* bar(); */ . . calls after calling The next logical step doesn't work, sprintf(asm_buffer,"jsr %s","bar"); asm(asm_buffer); . . evokes the error syntax error at or near "asm_buffer" from the 4.2 C compiler. You should try this on your particular C compiler. 4.2 XINU system calls The supported system calls are: xsend() - send a message to another process xreceive() - wait for a message and return it xrecvclr() - clear messages, returning waiting message (if any) xresume() - unsuspend a process, making it ready xsuspend() - suspend a process, placing it in hibernation xkill() - kill a process and remove it from the system xcreate() - create a process to start running a new procedure xgetpid() - get the process id of currently executing process xgetprio() - return the scheduling priority of a given process xchprio() - change the scheduling priority of a process xwait() - make current process wait on a semaphore xsignal() - signal a semaphore, releasing one waiting process xscreate() - create and initialize a semaphore, returning its id xsdelete() - delete a semaphore by releasing its table entry xsleep() - put a process to sleep for this many seconds. xmsleep() - put a process to sleep for this many milliseconds. For complete documentation on the system calls, see Comer's text. Most of the supported system calls function exactly as described in the book. The only changes were to provide a millisecond timer rather than a tenth of a second timer, and all XINU system calls are prefaced with "x" (for XINU) to avoid clashes with UNIX calls. All internal procedures and variables that are global have been prefaced with an underscore to avoid conflicting with application names. For example "_resched". The system is configurable and can be recompiled without any combination of the following optional services: realtime clock semaphores messages These and other typical configuration changes are isolated in _conf.h. The smallest 4.2 XINU system comes with only 5 system calls: xcreate() xresume() xsuspend() xkill() xgetpid() and is actually quite useful. Other minor differences between Comer's XINU and 4.2 XINU Comer's XINU is based on the LSI 11/2. 4.2 XINU is based on 4.2 UNIX. The source is almost entirely in C, and makes few assumptions about the underlying machine. Much of the code ports without changes. Besides the differences mentioned elsewhere in this paper, the other primary differences are the size of the registers and interrupt handling. The LSI is 16-bit while 4.2 is Occasional assumptions are necessarily made, unfortunately. Interrupts in the UNIX appear as software signals. Thus, disabling interrupts is done with the 4.2 signal support routines. If you intend to write your own system calls, you must allocate an int to store the old interrupt mask rather than a char. For example, is used to disable interrupts while storing the old processor status in Comer's definition of disable is: /* disable interrupts - LSI 11/2 */ #define disable(ps) asm("mfps ~ps"); asm("mtps $0340") while for 4.2 XINU, it is /* disable interrupts - 4.2BSD software interval timer */ #define disable(oldmask) oldmask = sigblock(1<<(SIGALRM-1)) Here, only the quantum timer interrupt is blocked. You may find that other signals should be blocked, however not all should (e.g. should probably not be trapped). Using UNIX system calls from XINU All UNIX system calls and many library calls should be made with some thought as to their consequences. for example, will completely overlay the entire XINU application and system. Where functionality is duplicated by UNIX, it is generally better to use XINU's calls. For example, if a process wants to go to sleep, calling the UNIX will stall the entire XINU system until the next interval timer occurs. If the process calls (the XINU equivalent) the current process is put on a queue waiting for the clock, and another XINU process is given control of the cpu. We have not reimplemented I/O, since we were able to use UNIX I/O without change, however the default behavior of UNIX I/O is to block, leading to a similar problem as (i.e. block until operation complete or until quantum expires). Note that 4.2 system calls restart automatically upon interrupts. This allows programs to run without having to explicitly handle the quantum interrupt. If this "blocking until quantum" behavior is undesirable, it is possible to use nonblocking I/O, either directly, or through a generalized interface leading to a second set of I/O system calls. Future work in this direction would be very useful. Using UNIX library calls from XINU Many UNIX library calls are nonreentrant, and do not protect themselves against this. This means that they use static variables which are common from one call to the next. If two processes make the same nonreentrant library call at the same time, it is likely that the routines will misbehave. Using reentrant versions of libraries is the best solution. Alternatively, one can embed (or surround, if you don't have the source) semaphores in the library calls (provided by XINU), one per common data structure (such as which is shared with all the routines that are part of the standard I/O library). XINU system calls are protected against reentrancy problems by disabling timer interrupts. Since and are used for XINU memory management, you should disable timer interrupts or set up a semaphore for access to the malloc data structures when doing memory management. Conclusion We have ported an operating system to the UNIX environment by emulating the environment of a microprocessor in a single UNIX process. We now have a tool that is capable of simulating any set of cooperating real-time processes. The applications have the ability to access all the power of UNIX, simply because the emulator runs as normal user code on a 4.2BSD system. Because all the XINU processes run in one UNIX process, it is especially easy to debug multiple programs with one debugger. Perhaps the nicest benefit of this work, has been the ability to write processes that efficiently share data structures, at the expense of using distinct global names. This has long been a missing feature of UNIX. References Comer, Douglas. Operating system design, the XINU approach. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1984. Joy, W., Cooper, E., Fabry, R., Lefler, S., McKusick, K., Mosher, D., "4.2BSD System Interface Overview", Computer Systems Research Group, U.C. Berkeley, July, 1983. Libes, Don. User-Level Shared Variables, Tenth USENIX Conference Proceedings, Summer 1985. Kepecs, Jonathan. Lightweight Processes for UNIX Implementation and Applications. Tenth

USENIX Conference Proceedings, Summer 1985. Tevanian, Jr., A., Rashid, R., Golub, D., Black, D., Cooper, E., Young, M., "Mach Threads and the UNIX Kernel: The Battle for Control", Summer 1987 USENIX Conference Proceedings.