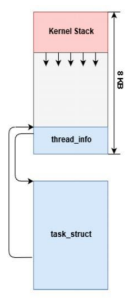
****

**Signal**: Although most signals can be ignored, blocked (masked temporarily), or caught (caused to execute a program-defined handler function), certain signals operate as NMIs, and can never be ignored, blocked, or caught. As with physical interrupt request lines, raising a signal twice does not necessarily cause two invocations of an associated handler; if the two signals are raised closely enough in time, the handler is executed only once. Traditionally, and by default, no information other than the signal number is provided to the handler, and a signal is masked while the handler for that signal executes. The user program’s hardware context, including all registers, is also copied from the kernel stack to the program’s stack, and is copied back during the sigreturn system call. This copying serves two purposes. First, remember that signals are the analogue of interrupts, and a program running on a private machine might want to manipulate registers during an interrupt. A user-level thread package, for example, may want to switch to a different thread when a signal handler executes. To do so, it needs access to the machine state. The other reason for copying the hardware state to user space is to avoid having the kernel stack overflow due to malicious programs

**System Call**

System Calls: ○ INT $0x80 to invoke ○ Place system call number in EAX ○ Return value in EAX. Trick to get EIP (and errno): ○ Make a fake function call, get return address, then use that to get errno address from offset

**Syscall** use callee saved registers because the caller saved registers are used as arguments so we only need the values in the registers once for that specific system call.

**A pipeline** is a set of processes chained by their standard streams, so that the output of each process (stdout) feeds directly as input (stdin) to the next one. program1 | program2 | program3• Similar to file scenario but: • No temporary file created • Processes run concurrently • $ ls | more vs $ ls > temp $ more < temp. Let's resume the previous example. When the command shell interprets the ls|more statement, it essentially performs the following actions: **1. Invokes the pipe( )** system call; let's assume that pipe( ) returns the file descriptors 3 (the pipe's read channel) and 4 (the write channel). **2. Invokes the fork( )** system call twice. **3.** **3. Invokes the close ()** system call twice to release file descriptors 3 and 4.

**Memory Allocation** ● **Types**: ○ a few small items → **kmalloc** ○ a lot of items, repeatedly → **slab cache** ○ a big, physically contiguous region → **free pages** ○ a big area of virtual memory → vmalloc (not necessarily physically contiguous) ● **kmalloc** ○ has gfp (get free pages) flags to customize allocation type (e.g. GFP\_KERNEL). ○ Each allocation is contiguous in physical memory ● **slab cache**: ○ for specific struct, set aside contiguous physical memory broken into struct size pieces (faster allocation/deallocation, less fragmentation) ○ NOTE: Nothing to specifically do with TLB or hardware caches ○ frequent allocations/deallocations ○ one cache per item type ○ physical memory is contiguous ○ Protocol: ■ Creation returns a page handle ■ Use handle to allocate/deallocate objects or to destroy the slab cache when done ● For memory allocation, use Buddy System algorithm ● malloc() is not a system call! malloc() uses sbrk, which is a system call

**Buddy System** ● Free pages are categorized into exponential “bins”, of size - 1, 2, 4 … pages each.

**Scheduling** ● Scheduler must be efficient, fair and responsive

**Types of jobs:** ○ Interactive - Driven by user interaction (shell) ○ Batch - Time to completion is main concern ○ Real-time - Periodic deadlines

**Linux scheduler:** ○ Splits time into **epochs**. Each task gets one **quantum** slice and programs are run until no **runnable** tasks are left. Then a new epoch is started. ○ Real-time tasks get higher priority ○ Schedule new task by calling schedule() Can be called by kernel (preempt) or task (yield)

**Task->state:** ○ **TASK\_RUNNING**: task is executing currently or waiting to execute; task is in a run queue on some processor ○ **TASK\_INTERRUPTIBLE**: task is sleeping on a semaphore/condition/signal; task is in a wait queue; can be made runnable by delivery of signal ○ **TASK\_UNINTERRUPTIBLE**: task is busy with something that can’t be stopped (e.g., a device that will stay in unrecoverable state without further task interaction); cannot be made runnable by delivery of signal ○ **TASK\_STOPPED**: task is stopped; task is not in a queue; must be woken by signal ○ **TASK\_ZOMBIE**: task has terminated; task state retained until parent collects exit status information; task is not in a queue. **Each processor has a run queue**. ○ Each run queue has two priority arrays. ○ Arrays contain lists of tasks of each priority, which corresponds to the index in the array - real time tasks are first 100 indices, standard tasks are other 40 indices. ○ They are double-buffered to implement epochs

**Mutex**: tell scheduler the process is not doing useful work – Try to acquire lock – If lock not available, put self on wait queue – Remove self from runqueue – Process is “sleeping” or “blocked” • Wake up processes during unlock – Check wait queue and add process back to run queue Signal is user-level interrupt .

**The x86 processor supports the notion of a task**; this hardware support is encapsulated in a Task State Segment, or TSS. The x86 requires that you set up one TSS for, among other things, privilege level stack switching. The important fields are SS0 and ESP0. These fields contain the stack segment and stack pointer that the x86 will put into SS and ESP when performing a privilege switch from privilege level 3 to privilege level 0 (for example, when a user-level program makes a system call, or when a hardware interrupt occurs while a user-level program is executing). These fields must be set to point to the kernel's stack segment and the processes kernel-mode stack, respectively. Note that when you start a new process, just before you switch to that process and start executing its user-level code, you must alter the TSS entry to obtain its new kernel-mode stack pointer. This way, when a privilege switch is needed, the correct stack will be set up by the x86 processor.

**Executing User-level Code:** ○ The convention to use for the privilege switch is the IRET instruction. Set up the correct values for the user-level EIP, CS, EFLAGS, ESP and SS registers on the kernel stack, and then execute IRET. The values for the CS and SS registers must point to the correct entries in the GDT that correspond to the user code and stack segments, respectively. The EIP you need to jump to is the entry point from bytes 24-27 of the executable that you have just loaded. Finally, you need to set up a user-level stack for the process. The DS register must be set to point to the correct entry in the GDT for the user mode data segment (USER DS) before you execute IRET. Conversely, when an entry into the kernel happens, for example, through a system call, exception, or interrupt, you should set DS to point to the KERNEL DS segment. Finally, you will need to modify the TSS.

ksoftirqd - Periodically runs tasklets

kswapd - Determines when to swap memory to disk

**Create a new program:** ○ **fork**: creates a copy of curr process ○ **exec**: replaces curr program w new program ○ Usually fork is called, and exec right after ○ Copy on write - “lazy” - instead of duplicating data, duplicate page tables, turn off write permission, and give each process a private copy of a page when it tries to write to it ● If we want a task to only run in kernel space, we use the kernel\_thread function. It inherits the address space of the previous user task.

**Task Structure**

**User View**: 1. Each task must be uniquely identifiable - using a process descriptor (which includes pid) 2. tgid (thread group id) is a pid for multithreaded applications (common id for all threads in the process). Most processes belong to a thread group with a single member.

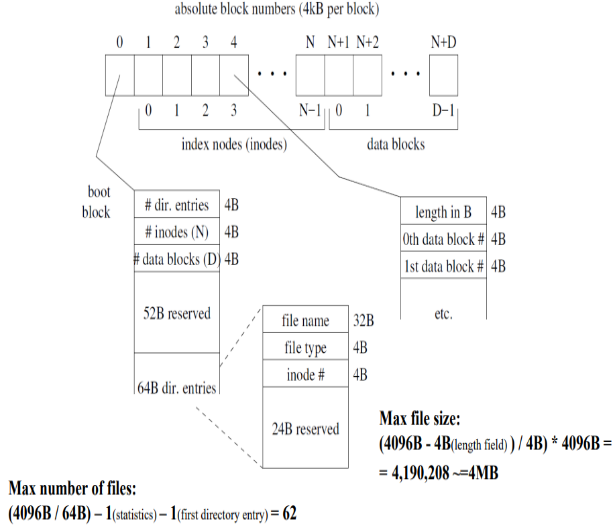
**Kernel View:** 1. Two dynamically structures that grow towards each other in an 8kB area: **thread\_info:** Keeps pointer to task\_struct/process descriptor **Kernel stack** Only 8kB per process in kernel space - don’t put too much on the kernel stack.

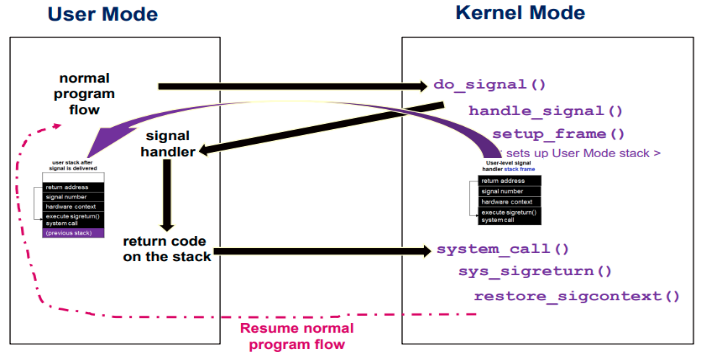
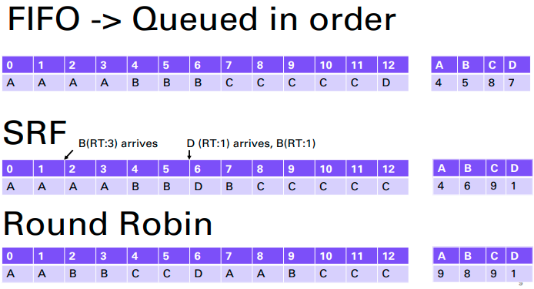
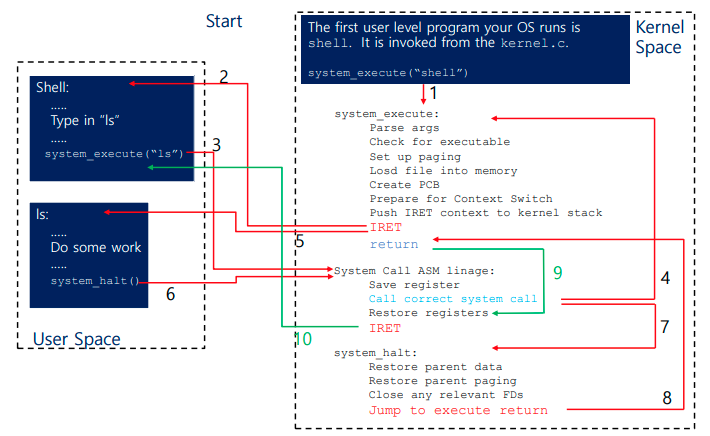
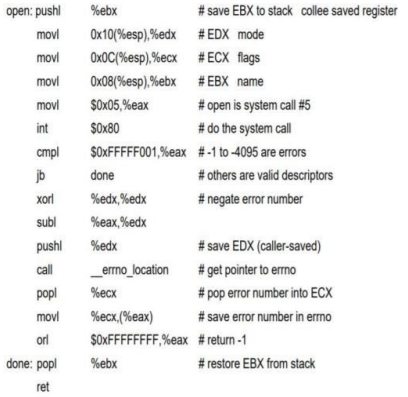
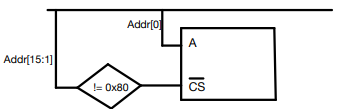
**Tasks**: stored in a cyclic, doubly-linked list, starting w/ init\_task at bootup ○ Quick access is achieved using a hash table based on using the PID for a key.

**Switch b/w tasks:** 1. Context switch from user to kernel 2.Switch task struct 3.Context switch to new task.

Your friend suggests adding a routine that replaces the driver’s internal data structures with a new set passed in from user-space (after discarding the previous version of the data structures). Explain why such a routine is useful and what drawbacks it presents. Avoids having to perform system calls and relying on interrupts/kernel to alter this data structure. Users have direct access and avoid the overhead of context switching. User uses the data structure incorrectly. Describe the system call calling convention for Linux in general terms and explain why (and where) indirection is used, why parameters are passed in the way that they are passed, why return values are defined as they are defined, and why specific registers are callee- or caller-saved. System call is a request generated by a software interrupt. Since its a kernel level interaction, indirection is added for security purposes. Indirection is also used in order to allow applications to operate on different OSes, so that the code of the program doesn’t have to change, only the table that contains the appropriate handlers and procedures. All registers are pushed onto the stack. System calls are referred to with numbers and use registers to pass operands into the OS rather than pushing the operands on to the stack. Different modes have different stacks. When a system call occurs, the program is calling a kernel level function from user level. The arguments would be present in the user stack, but then not be present once the stack switched over to the kernel. The solution to preserve the parameters through the mode switch is to pass them through specific registers of which will not be changed. Return values are put into EAX. EAX also holds the syscall number. Stack has EBX, ECX, EDX, etc. Syscalls with one parameter will use EBX, two will use EBX and ECX etc. They are defined in a way that gives information about the result of the system call. Bytes read, pointer to file, index, or just a success or failure. They are defined this way to provide feedback and error handling to the program that called them. This way, the program can error handle itself, or make decisions on what to do next based on the returned value. Why is assembly linkage necessary between the address specified for a system call in the IDT and the C function that implements the system call? In other words, why don’t operating systems simply use the C calling convention for system calls? Iret. System calls are triggered in User Mode, are executed in kernel mode, and then need to return the privilege back to user mode. The only way to go from Kernel Mode (Privilege Level 0) to User Mode (Privilege Level 3) is iret. Arguments are passed by register not stack. Describe four of the default behaviors taken in response to signal delivery. 4A. ULK has: Terminate(kill), Dump, Ignore, Stop, Continue. More generally: Ignore, Execute (do 1 of those things^^), or catch the signal by invoking a corresponding signal-handler function. (maybe block is the fourth one and these 4 are what they’re looking for) What is the difference between masking a signal and ignoring it? Masking a signal is just postponing it until a later point. When you mask a signal, it does not get lost, whereas a signal gets lost if you ignore it Masked signals will be delivered when they are unmasked. However, ignored signal will be discarded. Describe two similarities and two differences between signals and interrupts. Similarities: Asynchronous with respect to program execution. Can be ignored, blocked, or caught Differences: Generated by software (either the kernel or a program via a sys call). No devices associated with a signal, only software with permission can send a signal. What information is provided to a signal handler with the traditional model? What additional information is available under the Posix standard? Use an example to explain how this additional information can be useful. Traditional: Only signal #. Posix: has info to differentiate senders. Why does Linux copy the saved machine state from the kernel stack to the user stack when delivering a signal? Reason for copying the hardware state to user space is a program running on a private machine might want to manipulate registers during an interrupt and to avoid having the kernel stack overflow due to malicious programs. The big reason is that any switch from user mode to kernel mode starts at the bottom of a kernel stack. Thus, any IRET context from the initial program exception is overwritten and lost once the signal handler returns. Describe two ways in which a signal handler stack frame can be torn down (i.e., removed) from the stack. 1.Copy the hardware context from the source of the signal call to the kernel stack. 2.Remove the frame from the user stack by restoring the old esp of user stack. In what cases is it useful to force a program to receive a particular signal, even though the program has asked that the signal be masked out? When delivering signals generated by exceptions. A program that generates an exception cannot make forward progress, since the processor does not know how to proceed. The operating system can either terminate the program or make a last-ditch effort to get its attention, which is the purpose of the signal forcing functions. List all of the OS and architectural data structures that need to be managed as part of the state of a task. (Hint, every data structure for virtualizing anything about the system gets included here.) PCB and its associated file descriptors, page table, RTC value, stack frame values, video mapping, signal handler packages. What events could cause an OS scheduler to switch the system to executing a new task? If the process used up all its quanta in this epoch. The process calls schedule () to yield the CPU. The current process goes to sleep or gets killed. What should happen when the interrupt handler for IRQ0 causes a context switch from task A to task B and returns? What if task A was already interrupted by IRQ6 when IRQ0 arrives and causes a context switch? Task B should be running after IRQ0 returns. If A was already interrupted by IRQ6 then Task B will be running from wherever it was after the IRQ0 executes. Explain the advantages and disadvantages of round-robin scheduling and FIFO scheduling for tasks. Round Robin: Advantages: Every task gets an equal amount of time to execute, no chance of starvation. Good response time. Slow Disadvantages: higher average time to completion. Extensive overhead FIFO: Advantage: No chance of starvation Disadvantage: Throughput can be low, since long processes can hold the CPU. Turnaround time, waiting time and response time can be high for the same reasons above. Explain the performance penalties for context switching. How can system designers reduce or avoid those penalties in some cases? So, there is lot of time needed for OS kernel to save execution state (all registers; and many special control structures) of current running process to memory, and then load execution state of other process (read in from memory). TLB flush, if needed, will add some time to the switch, but it is only small part of total overhead. Use a more efficient scheduling algorithm that requires less switches to compute processes. Scheduling Goals • Throughput – Spend more Time doing useful work • Time to completion – Finish jobs quickly • Responsiveness – Maintain illusion of parallel execution • Fairness • Some of these are in conflict – More context switches: higher interactivity, lower throughput. How to pass in lots of data: The last common way to pass arguments is to store them in memory. Before making the System Call the caller must store a pointer to the argument's location in a register for the System Call handler. What is meant when saying that a scheduling algorithm could result in starvation? That it is possible for a process to be runnable but never scheduled. The layout of executable files in the file system the entire file stored in the file system is the image of the program to be executed. In this file, a header that occupies the first 40 bytes gives information for loading and starting the program. The first 4 bytes of the file represent a “magic number” that identifies the file as an executable. These bytes are, respectively, 0: 0x7f; 1: 0x45; 2: 0x4c; 3: 0x46. If the magic number is not present, the execute system call should fail. The other important bit of information that you need to execute programs is the entry point into the program, i.e., the virtual address of the first instruction that should be executed. This information is stored as a 4-byte unsigned integer in bytes 24-27 of the executable, and the value of it falls somewhere near 0x08048000 for all programs we have provided to you. When processing the execute system call, your code should make a note of the entry point, and then copy the entire file to memory starting at virtual address 0x08048000. It then must jump to the entry point of the program to begin execution using IRET. #define USER\_STACK\_ADDR 0x08400000 - 0x4 = 132MB – 4 Bytes User Stack Address. Getting to ring 3 can be done using iret because the way it works has been documented. When you receive an interrupt, the processor pushes:1. The stack segment and pointer (ss:esp), as 4 words 2.EFLAGS 3.The return code segment and instruction pointer (cs:eip), as 4 words 4. An error code, if required. iret works by undoing steps 1-3 (The ISR is responsible for undoing step 4 if necessary). We can use this fact to get to ring 3 by pushing the required information to the stack and issuing an iret instruction. Make sure you have the proper CPL in your code and stack segments (the low two bits should be set in each) TSS: the important fields are SS0 and ESP0. These fields contain the stack segment and stack pointer that the x86 will put into SS and ESP when performing a privilege switch from privilege level 3 to privilege level 0 (for example, when a user-level program makes a system call, or when a hardware interrupt occurs while a user-level program is executing). Scheduling Philosophy • Several types of jobs exist – Interactive• examples: editors, GUIs • driven by human interaction (e.g., keystrokes, mouse clicks) • little or no work to do after each event • but important to respond quickly – Batch • examples: compilation, simulation • usually only Time to completion matters • want a fair share of CPU computation. Do not need to be responsive and are often penalized by the scheduler – Real-time • examples: music, video, teleconferencing • periodic deadlines (e.g., 30 frames per second of video) • work often only useful if finished on time • General strategy – break time into slices – allow interactive jobs to preempt the current job based on interrupts • typically, a job becomes runnable when an interrupt occurs (e.g., a key is pressed) • Linux checks for rescheduling after each interrupt, system call, and exception. Linux processes are preemptable. When a process enters the TASK\_RUNNING state, the kernel checks whether its dynamic priority is greater than the priority of the currently running process. If it is, the execution of current is interrupted and the scheduler is invoked to select another process to run (usually the process that just became runnable). Of course, a process also may be preempted when its time quantum expires. When this occurs, the TIF\_NEED\_RESCHED flag in the thread info structure of the current process is set, so the scheduler is invoked when the timer interrupt handler terminates. After a keyboard int the scheduler selects the editor and performs a process switch; as a result, the execution of the editor is resumed very quickly and the character typed by the user is echoed to the screen. When the character has been processed, the text editor process suspends itself waiting for another keypress and the compiler process can resume its execution. Quantum 2 short: lots of context switches. Quantum 2 long: processes don’t look like they’re running concurrently. The system call handler performs the following operations: Saves the contents of most registers in the Kernel Mode stack (this operation is common to all system calls and is coded in assembly language). • Handles the system call by invoking a corresponding C function called the system call service routine. Exits from the handler: the registers are loaded with the values saved in the Kernel Mode stack, and the CPU is switched back from Kernel Mode to User Mode (this operation is common to all system calls and is coded in assembly language). Synchronization not needed: Interrupt handlers and tasklets need not to be coded as reentrant functions. • Per-CPU variables accessed by softirqs and tasklets only do not require synchronization. • A data structure accessed by only one kind of tasklet does not require synchronization. CPU Bound means the rate at which process progresses is limited by the speed of the CPU. A task that performs calculations on a small set of numbers, for example multiplying small matrices, is likely to be CPU bound. I/O Bound means the rate at which a process progresses is limited by the speed of the I/O subsystem. A task that processes data from disk, for example, counting the number of lines in a file is likely to be I/O bound. Signal Delivery: 1. Mask all other signals. 2. Set up the signal handler’s stack frame. You’ll need the current value of the user-level ESP register to find the user’s current stack location. The signal handler stack frame goes directly above this on the stack. Setting up the signal handler stack frame involves: copying a return address and a signal number parameter to the user-level stack, copying the process’s hardware context (see Figure 2) from the point when the program was interrupted for the signal, and copying a small amount of assembly linkage to the user-level stack that calls sigreturn when the signal handler is finished. 3. Finally, execute (in user space) the handler specified in the signal descriptor. No other information needs to be passed to the signal handler (no siginfo t structure like the modern Linux signals). • On call to sigreturn • dump stack frame • check if signal was delivered during system call • if so, checks whether system call should be restarted • if not, returns –EINTR • if so, sets EAX to previous value (sys call #) and changes PC to re-execute INT x80 instruction • if not, simply continue the process • Restarting system calls avoids having to process EINTR in user space • What happens if sigreturn not called (possible security hole?) • no state left on kernel stack, so has no impact on kernel • can only affect user-level stack for that process. When the user-level signal handler returns, it will use the return address you have copied on its stack, which will jump to the assembly linkage (also on the stack). This assembly linkage should make the sigreturn system call (using the standard int $0x80 user-level system call calling convention). The sigreturn system call should copy the hardware context that was on the user-level stack back onto the processor. To find the hardware context, you will need to know the user-level value of ESP (will be saved by your system call handler) as well as the exact setup of the user-level stack frame. To copy the hardware context back onto the processor, you will actually overwrite the kernel’s copy of the process’s hardware context that was saved on the kernel stack when it handled the sigreturn system call. In this way, when the sigreturn system call handler returns to user space, the hardware context will automatically be copied back onto the processor by your return-from-kernel code that you have already written. One thing to be careful of: you’ll probably have system calls set up to return a value (in EAX) to user space. Be sure you don’t clobber the user’s EAX value from its hardware context with a bogus “return value” from sigreturn – have sigreturn return the hardware context’s EAX value so that you won’t have to special-case the return from sigreturn. What would happen if user program blocked signal from an exception? • Program can’t execute next instruction (it causes an exception) • kernel can’t deliver signal (signal blocked) • deadlock! Notice that blocking a signal is different from ignoring it. A signal is not delivered as long as it is blocked; it is delivered only after it has been unblocked. An ignored signal is always delivered, and there is no further action. It is important to note that signals are delivered only to the program currently running; a program with pending signals is not given priority over another program for use of the processor just because of the pending signals. Signals may thus remain pending for some time while other programs execute. The execute system call attempts to load and execute a new program, handing off the processor to the new program until it terminates. The command is a space-separated sequence of words. The first word is the file name of the program to be executed, and the rest of the command—stripped of leading spaces—should be provided to the new program on request via the getargs system call. The execute call returns -1 if the command cannot be executed, for example, if the program does not exist or the filename specified is not an executable, 256 if the program dies by an exception, or a value in the range 0 to 255 if the program executes a halt system call, in which case the value returned is that given by the program’s call to halt. The halt system call terminates a process, returning the specified value to its parent process. The system call handler itself is responsible for expanding the 8-bit argument from BL into the 32-bit return value to the parent program’s execute system call. Be careful not to return all 32 bits from EBX. This call should never return to the caller. The read system call reads data from the keyboard, a file, device (RTC), or directory. This call returns the number of bytes read. If the initial file position is at or beyond the end of file, 0 shall be returned (for normal files and the directory). In the case of the keyboard, read should return data from one line that has been terminated by pressing Enter, or as much as fits in the buffer from one such line. The line returned should include the line feed character. In the case of a file, data should be read to the end of the file or the end of the buffer provided, whichever occurs sooner. In the case of reads to the directory, only the filename should be provided (as much as fits, or all 32 bytes), and subsequent reads should read from successive directory entries until the last is reached, at which point read should repeatedly return 0. For the real-time clock (RTC), this call should always return 0, but only after an interrupt has occurred (set a flag and wait until the interrupt handler clears it, then return 0). You should use a jump table referenced by the task’s file array to call from a generic handler for this call into a file-type-specific function. This jump table should be inserted into the file array on the open system call (see below). The write system call writes data to the terminal or to a device (RTC). In the case of the terminal, all data should be displayed to the screen immediately. In the case of the RTC, the system call should always accept only a 4-byte integer specifying the interrupt rate in Hz, and should set the rate of periodic interrupts accordingly. Writes to regular files should always return -1 to indicate failure since the file system is read-only. The call returns the number of bytes written, or -1 on failure. The RTC device itself can only generate interrupts at a rate that is a power of 2 (do a parameter check), and only up to 8192 Hz. Your kernel should limit this further to 1024 Hz — an operating system shouldn’t allow user space programs to generate more than 1024 interrupts per second by default. Note that you should be using the RTC’s Periodic Interrupt function to generate interrupts at a programmable rate. The RTC interrupt rate should be set to a default value of 2 Hz (2 interrupts per second) when the RTC device is opened. For simplicity, RTC interrupts should remain on at all times. The open system call provides access to the file system. The call should find the directory entry corresponding to the named file, allocate an unused file descriptor, and set up any data necessary to handle the given type of file (directory, RTC device, or regular file). If the named file does not exist or no descriptors are free, the call returns -1. The close system call closes the specified file descriptor and makes it available for return from later calls to open. You should not allow the user to close the default descriptors (0 for input and 1 for output). Trying to close an invalid descriptor should result in a return value of -1; successful closes should return 0. The getargs call reads the program’s command line arguments into a user-level buffer. Obviously, these arguments must be stored as part of the task data when a new program is loaded. Here they are merely copied into user space. If the arguments and a terminal NULL (0-byte) do not fit in the buffer, simply return -1. The shell does not request arguments, but you should probably still initialize the shell task’s argument data to the empty string. The vidmap call maps the text-mode video memory into user space at a pre-set virtual address. Although the address returned is always the same (see the memory map section later in this handout), it should be written into the memory location provided by the caller (which must be checked for validity). If the location is invalid, the call should return -1. To avoid adding kernel-side exception handling for this sort of check, you can simply check whether the address falls within the address range covered by the single user-level page. Note that the video memory will require you to add another page mapping for the program, in this case a 4 kB page. It is not ok to simply change the permissions of the video page located < 4MB and pass that address. Note that some system calls need to synchronize with interrupt handlers. For example, the read system call made on the RTC device should wait until the next RTC interrupt has occurred before it returns. Use simple volatile flag variables to do this synchronization (e.g., something like int rtc interrupt occurred;) when possible (try something more complicated only after everything works!), and small critical sections with cli/sti. For example, writing to the RTC should block interrupts to interact with the device. Writing to the terminal also probably needs to block interrupts, if only briefly, to update screen data when printing (keyboard input is also printed to the screen from the interrupt handler) When are signals delivered? (see entry.S) • check sigpending (in task structure) when returning from • any interrupt • any exception • any system call • only deliver to currently executing process • E.g.: • Process A executes kill(pid of B, signal) • B’s sigpending is set to 1 • B woken up if sleeping (added to run queue) • A keeps executing • Later, scheduler performs a context switch A -> B • B returns from system call or interrupt after context switch • Signal is delivered at this point A signal is fatal for a given process if delivering the signal causes the kernel to kill the process. The SIGKILL signal is always fatal; moreover, each signal whose default action is "Terminate" and which is not caught by a process is also fatal for that process. Notice, however, that a signal caught by a process and whose corresponding signal-handler function terminates the process is not fatal, because the process chose to terminate itself rather than being killed by the kernel. Why bother copying h/w context (registers, etc.) from kernel stack to user stack? • switch threads in signal handler by swapping context with that on stack • avoid kernel stack overflow due to malicious programs • allow user to modify context. 0x004B8000 / 4KB (4096) = 4B8

syscall use callee saved registers because the caller saved registers are used as arguments so we only need the values in the registers once for that specific system call. Advantage of ext2 over linked list approach: more space/memory. Advantage of ext2 vs tree (triply indirect blocks): less accesses needed to access data. Inodes also make it easier to recover data and allows multiple files to point to the same data block for sharing. Spinlocks eat up CPU– OK for short-term contention between processors – Wasteful while waiting for external event Mutex: tell scheduler the process is not doing useful work – Try to acquire lock – If lock not available, put self on wait queue – Remove self from runqueue – Process is “sleeping” or “blocked” • Wake up processes during unlock – Check wait queue and add process back to run queue Signal is user-level interrupt .Although most signals can be ignored, blocked (masked temporarily), or caught (caused to execute a program-defined handler function), certain signals operate as NMIs, and can never be ignored, blocked, or caught. As with physical interrupt request lines, raising a signal twice does not necessarily cause two invocations of an associated handler; if the two signals are raised closely enough in time, the handler is executed only once. Traditionally, and by default, no information other than the signal number is provided to the handler, and a signal is masked while the handler for that signal executes. The user program’s hardware context, including all registers, is also copied from the kernel stack to the program’s stack, and is copied back during the sigreturn system call. This copying serves two purposes. First, remember that signals are the analogue of interrupts, and a program running on a private machine might want to manipulate registers during an interrupt. A user-level thread package, for example, may want to switch to a different thread when a signal handler executes. To do so, it needs access to the machine state. The other reason for copying the hardware state to user space is to avoid having the kernel stack overflow due to malicious programs. Advantage of ext2 over linked list approach: more space/memory. Advantage of ext2 vs tree (triply indirect blocks): less accesses needed to access data. Inodes also make it easier to recover data and allows multiple files to point to the same data block for sharing. Spinlocks eat up CPU– OK for short-term contention between processors – Wasteful while waiting for external event

**The first child process**, which must execute the ls program, performs the following operations:**1. Invokes dup2(4,1)** to copy file descriptor 4 to file descriptor 1. From now on, file descriptor 1 refers to the pipe's write channel. **2. Invokes the close( )** system call twice to release file descriptors 3 and 4. **3. Invokes the execute ( )** system call to execute the ls program. The program writes its output to the file that has file descriptor 1 (the standard output); i.e., it writes into the pipe. **The second child process must execute the more program**; therefore, it performs the following operations: **1. Invokes dup2(3,0)** to copy file descriptor 3 to file descriptor 0. From now on, file descriptor 0 refers to the pipe's read channel. **2. Invokes the close( )** system call twice to release file descriptors 3 and 4. **3. Invokes the exec( )** system call to execute more. By default, that program reads its input from the file that has file descriptor. The fork() function is used to create a new process from an existing process. The new process is called the child process, and the existing process is called the parent. You can tell which is which by checking the return value from fork(). The parent gets the child's pid returned to him, but the child gets 0 returned to him.

**\*Translate virtual address to physical address:** The first 10 bits (offset ) indicate the page directory entry. Added to the PD stored in CR3 to give the specific entry. **Memory access 1:** access the data at that entry to get the correct page table entry**.** The next 10 bits are then used to indicate the index into the page table. **Memory access 2:** Access the page table to find the correct page frame Final 12 bits are used to offset into the page(byte offset so you need 12 bits). **Memory access 3**: data at the correct page frame. So there are 3 memory accesses involved \*(\*(CR3 + 4 \* (10 high bits)) + 4 \* (10 mid bits)) + page\_offset ****

**Caller-saved**

EAX, ECX, EDX

**Callee-Saved**

EBX, ESI, EDI

Word=2 bytes.

Long=4 bytes.

EAX=4bytes. Char=1 byte.

Int = 4 bytes.

1B = 8bit

1kB= 2^10B

4kB=2^12B

1MB=2^20B

4MB=2^22B

1GB= 2^30B

**int32\_t sys\_sigreturn (void)**{

pageDir[1].US = 0; // Enable sys protection

flushTLB();

uint32\_t ebp = pcb->sig\_ebp; // Restore context

uint32\_t esp = pcb->sig\_esp;

asm volatile ("movl %0, %%ebp\n" "movl %1, %%esp \n"

: : "r"(ebp), "r"(esp) );

// Restore kernel stack

memcpy((void\*) (pcb->sig\_ebp), (void\*) &(pcb->sig\_stackshot), pcb->sig\_stacksize);

pcb->sig\_mask = 0; // Unmask the interrupt line

return pcb->sig\_eax; // Return backup EAX}

**uint32\_t sig\_dispatch(uint32\_t eax){**

*// Called from kernel-user linkage, handle the signal*

if (pcb->sig\_mask){ return eax; } *// If it is handling, exit*

uint32\_t pending = pcb->sig\_pending; *//Get the pending signal*

*// Check if the program ever went to user space*

if (pcb->user\_esp == NULL){return eax;}

*// Determine if handler is custom*

if ((pending < 5)&& (pcb->sig\_handlers[pending] != NULL)){

uint32\_t user\_esp = pcb->user\_esp; *// Load the last user esp*

if (user\_esp <= Kernel\_addr){ *// No user esp available*

return eax;}

*// Call custom handler*

*// Clear the pending signal and mask the line*

pcb->sig\_pending = NULLSIG;

pcb->sig\_mask = 1;

pcb->sig\_eax = eax; *// Save eax*

uint32\_t ebp, esp; *// Save current context*

asm volatile ("movl %%ebp, %0 \n"

"movl %%esp, %1 \n"

: "=r"(ebp), "=r"(esp) );

pcb->sig\_ebp = ebp;

pcb->sig\_esp = esp;

*// Build the stack frame and back to user space*

pcb->sig\_stacksize = pcb->tss\_esp - ebp; *// Save kernel stack*

memcpy((void\*) &(pcb->sig\_stackshot), (void\*) ebp, pcb->sig\_stacksize);

*// Push signal number and return address*

asm volatile ( "movl %1, %%esp \n"

"pushl %2 \n"

"pushl %3 \n"

"movl %%esp, %0 \n"

"movl %4, %%esp \n"

: "=r"(user\_esp)

:"r"(user\_esp), "r"(pending), "r"(&sig\_linkage),

"r"(pcb->sig\_esp)

: "memory", "cc");

uint32\_t prog\_ss = USER\_DS; *// Push argument;call IRET*

uint32\_t prog\_cs = USER\_CS;

*// Switch to user-defined handler*

asm volatile ("pushl %0\n" "pushl %1\n" "pushfl\n" "pushl %2\n" "pushl %3\n"

::"r"(prog\_ss), "r"(user\_esp), "r"(prog\_cs), "r"(pcb->sig\_handlers[pending])

: "memory", "cc");

pageDir[1].US = 1; *// Disable sys protection*

flushTLB();

asm volatile ("iret\n");}

else{pcb->sig\_pending = NULLSIG; *// Clear pending signal*

pcb->sig\_mask = 1;

sig\_handle(pending); *// Call the default handler* }

pcb->sig\_mask = 0; *// Unmask the interrupt line*

*// eax is used to preserve last eax from syscall, not used at here just passthrough,This is a dummy data for linkage other than syscall, it is garbage so whatever*

return eax;}

**int32\_t sys\_set\_handler (int32\_t signum, void\* handler\_address){**

if (signum > 4) { return -1; }

pcb->sig\_handlers[signum] = handler\_address;

return 0; **}**

**int32\_t sys\_close(int32\_t fd){**

if ((fd < 2) || (fd > 7)) /\* initial fd sanity check\*/{ error\_sound(); return -1; }

if ((pcb->file\_descriptor)[fd].flags == FD\_FLAG\_EMPTY){return -1; *// Already closed*}

(pcb->file\_descriptor)[fd].file\_op\_table\_ptr = 0;

(pcb->file\_descriptor)[fd].file\_position = 0;

(pcb->file\_descriptor)[fd].inode = 0;

(pcb->file\_descriptor)[fd].flags =FD\_FLAG\_EMPTY;

return 0;}

**int32\_t sys\_getargs (uint8\_t\* buf, int32\_t nbytes){**

if (pcb->arg\_len == 0) *// Check the buffer and length*{

error\_sound(); return -1; }

else if ((pcb->arg\_len + 1) > nbytes) {error\_sound();return -1; }

memcpy(buf, pcb->arg\_buffer, pcb->arg\_len); *// Copy the argument to buffer*

buf[pcb->arg\_len] = '\0'; return 0; **}**

**void sig\_set(pcb\_t\* pcb, uint8\_t sig\_num){**

*// Check if the signal is masked.Unlike the sepcification, I treat sig\_num with an order.*

if ((pcb->sig\_pending <= sig\_num) || (pcb->sig\_mask)){

if (verbose\_mode){

printf("\n<!> Unable to deliver the signal.\n");}return;}

*// Set the signal*

if (verbose\_mode){ printf("\n<i> Program %s on terminal %u received sig\_num %u\n", pcb->command, pcb->terminal\_id, sig\_num); }

pcb->sig\_pending = sig\_num;}

**void sig\_collect\_esp(uint32\_t esp)**{

// Record last user ESP

if ((esp > KERNEL\_ ADDR) && (!(pcb->sig\_mask))) {pcb->user\_esp = esp;}}

**void sig\_linkage(){**

// Make sys\_sigreturn syscall

asm volatile ("movl $10, %eax \n" "int $0x80 \n" );}

**void sig\_handle(uint8\_t sig\_num){**

// Handle accroding to sig\_num

switch (sig\_num) {case DIV\_ZERO: case SEGFAULT: case INTERRUPT: case SYSKILL:

// Kill the process sys\_halt(0); break; default: break; }}

**void sig\_set(pcb\_t\* pcb, uint8\_t sig\_num){**

*// Check if the signal is masked.Unlike the sepcification, I treat sig\_num with an order.*

if ((pcb->sig\_pending <= sig\_num) || (pcb->sig\_mask)){

if (verbose\_mode){

printf("\n<!> Unable to deliver the signal.\n");}return;}

*// Set the signal*

if (verbose\_mode){ printf("\n<i> Program %s on terminal %u received sig\_num %u\n", pcb->command, pcb->terminal\_id, sig\_num); }

pcb->sig\_pending = sig\_num;}

**void sig\_collect\_esp(uint32\_t esp)**{

// Record last user ESP

if ((esp > KERNEL\_ ADDR) && (!(pcb->sig\_mask))) {pcb->user\_esp = esp;}}

**void sig\_linkage(){**

// Make sys\_sigreturn syscall

asm volatile ("movl $10, %eax \n" "int $0x80 \n" );}

void sig\_handle(uint8\_t sig\_num){

// Handle accroding to sig\_num

switch (sig\_num) {case DIV\_ZERO: case SEGFAULT: case INTERRUPT: case SYSKILL:

// Kill the process sys\_halt(0); break; default: break; }}

**C calling convention**

int binary\_search (int key, int\* array, int size);

return address //ebp+4 int key // ebp+8 int\* array // ebp+12 int size. // ebp+16

pushl %ebp

movl %ebp, %esp

pushl 3,2,1

call callee

addl $12, %esp //tear popl 1,2,3 leave ret

int32\_t sys\_execute(const uint8\_t\* command){

progress = 1; *// Set progress flag*

int available\_pid = find\_next\_pid(); *// Try to find next available PID*

*// Check if can have more programs*

if (available\_pid == -1) { progress = 0; return -1; }

*// Sanity check for multiterminal base terminal init*

if (command == NULL) *// Parameter Check{*

progress = 0; return -1; }

*// Get program name*

unsigned int prog\_name\_start, prog\_name\_len;

for (prog\_name\_start = 0; prog\_name\_start < strlen((int8\_t\*) command); prog\_name\_start++){

if (command[prog\_name\_start] != ' ') { break; } }

command += prog\_name\_start;

for (prog\_name\_len = 0; prog\_name\_len < strlen((int8\_t\*) command); prog\_name\_len++) { if (command[prog\_name\_len] == ' ') { break; }}

if (prog\_name\_len == 0) *// Program name check*{ progress = 0; return -1; }

uint8\_t prog\_name[prog\_name\_len + 1]; *// Extract the program names*

memcpy(prog\_name, command, prog\_name\_len);

prog\_name[prog\_name\_len] = '\0';

if (check\_executable((uint8\_t \*) prog\_name) == -1) // Executable check

{progress = 0; return -1; }

/\*\*\*\*\*\* Passed all checks, try to start the program \*\*\*\*\*\*/

command += prog\_name\_len; *// Get Arguments*

uint8\_t arg\_buffer\_local[128]; *// Create temp arg buffer*

uint32\_t arg\_len\_local = 0;

if (command[0] != '\0') {

unsigned int arg\_start; *// Trim the start of command*

for (arg\_start = 0; arg\_start < strlen((int8\_t\*) command); arg\_start++){

if (command[arg\_start] != ' ') { break; }}

command += arg\_start;

if (command[0] !='\0') {

unsigned int arg\_end; *// ^Trim the end of command*

for (arg\_end = strlen((int8\_t\*) command) - 1; arg\_end >= 0; arg\_end--) {

if (command[arg\_end] != ' ') { break; }}

arg\_len\_local = ++arg\_end; *// Copy the argument to buffer*

memcpy(arg\_buffer\_local, command, arg\_len\_local);

arg\_buffer\_local[arg\_len\_local] = '\0'; }

else{ arg\_len\_local = 0; *// Rest of commands are empty*}}

else{arg\_len\_local = 0; *// String after program name is empty*}

reMap4MBPage(available\_pid); *// Create page for new program*

*// Load code into memory*

dentry\_t prog\_dentry;

uint8\_t\* prog\_page\_addr = (uint8\_t\*) PROGRAM\_PAGE\_ADDR;

int32\_t prog\_size = return\_file\_size((uint8\_t \*) prog\_name);

if (read\_dentry\_by\_name((uint8\_t \*) prog\_name, &prog\_dentry) == -1 || prog\_size == -1) {progress = 0; return -1; }

read\_data(prog\_dentry.inode\_number, 0, prog\_page\_addr, prog\_size);

uint32\_t prog\_eip = (prog\_page\_addr[27] << 24) | (prog\_page\_addr[26] << 16) | (prog\_page\_addr[25] << 8) | prog\_page\_addr[24];

*// Create PCB*

memset(pcb\_pointer->sig\_stackshot, 0, 80); *// uint32\_t is 4 bytes \* 20 spaces*

memset(pcb\_pointer->sig\_handlers, 0, 20); *// void\* is 4 bytes \* 5 handlers*

memcpy(&(pcb\_pointer->arg\_buffer), &arg\_buffer\_local, arg\_len\_local);

pcb\_pointer->arg\_len = arg\_len\_local;

if (prog\_name\_len > MAX\_CMD\_LEN) *// Parse command*{

prog\_name\_len = MAX\_CMD\_LEN; }

memset(&(pcb\_pointer->command), '\0', (MAX\_CMD\_LEN + 1));

memcpy(&(pcb\_pointer->command), &prog\_name, prog\_name\_len);

(pcb\_pointer->command)[MAX\_CMD\_LEN] = '\0';

int fd\_i; // Initialize file desc array

for (fd\_i = 0; fd\_i < 8; fd\_i++){

(pcb\_pointer->file\_descriptor)[fd\_i].flags = FD\_FLAG\_EMPTY; }

**int32\_t sys\_getargs (uint8\_t\* buf, int32\_t nbytes){**

if (pcb->arg\_len == 0) // Check the buffer and length{return -1; }

else if ((pcb->arg\_len + 1) > nbytes) {return -1; }

memcpy(buf, pcb->arg\_buffer, pcb->arg\_len); // Copy argument to buffer

buf[pcb->arg\_len] = '\0'; return 0; }

**void switchContext(unsigned int terminal\_id)**

terminals[current\_terminal].pcb = pcb// Save current PCB

// Save current process context

uint32\_t ebp, esp;

asm volatile ("movl %%ebp, %0 \n"

"movl %%esp, %1 \n"

: "=r"(ebp), "=r"(esp) );

terminals[current\_terminal].ebp = ebp;

terminals[current\_terminal].esp = esp;

terminals[current\_terminal].tss\_esp = tss.esp0;

switchTerminalInit(terminal\_id); *// Initialize the new terminal}*

else{ // Initialized, then we do the same thing in halt()...

ebp = terminals[terminal\_id].ebp; *// Give up the current stack frame*

esp = terminals[terminal\_id].esp;

pcb = terminals[terminal\_id].pcb; *// Reset PCB pointer*

reMap4MBPage(pcb->process\_id); *// Remap Program Page*

lushTLB();

// Relocate kernel stack

tss.esp0 = terminals[terminal\_id].tss\_esp;

// Do context switch

asm volatile (

"movl %0, %%ebp \n"

"movl %1, %%esp \n"

:: "r"(ebp), "r"(esp) ); }

*// Save old EBP and return address for halt*

uint32\_t ebp, esp;

asm volatile ("movl %%ebp, %0 \n" "movl %%esp, %1 \n"

: "=r"(ebp), "=r"(esp));

pcb\_pointer->ebp = ebp;

pcb\_pointer->esp = esp;

*// Save and relocate kernel stack*

tss.esp0 = KERNEL\_STACK\_ADDR - available\_pid \* KERNEL\_STACK\_OFFSET - 4;

pcb\_pointer->tss\_esp = tss.esp0;

pcb = pcb\_pointer; *// Switch current PCB*

terminals[pcb->terminal\_id].pcb = pcb; *// Modify TI*

vrtc\_alarm[pcb->terminal\_id] = 0; *// Reset VRTC alarm counter*

// Mark the PCB pool position as occupied

pcb\_pool[available\_pid] = pcb;

// Push arguments and call IRET

uint32\_t prog\_ss = USER\_DS;

uint32\_t prog\_esp = PROGRAM\_STACK\_ADDR - 4;

uint32\_t prog\_cs = USER\_CS;

// Clear progress flag

progress = 0;

asm volatile ("pushl %0 \n" "pushl %1 \n" "pushfl\n" "pushl %2\n" "pushl %3 \n" "iret \n"

:: "r"(prog\_ss), "r"(prog\_esp), "r"(prog\_cs), "r"(prog\_eip)

: "memory", "cc" **);**

// Should never return at here

return -1;**}**