Lecture Notes on Operating Systems

Mythili Vutukuru, Department of Computer Science and Engineering, IIT Bombay

Process Management in xv6

We will study xv6 process management by walking through some of the paths in the code: the PCB datastructure (sheet 23), interrupt handling (sheets 32–34), process related system calls like fork (sheets 24–25), process scheduling (sheet 27), and the workings of the first init and shell processes (sheets 24–25, sheets 83–88).

0. Review of x86 architecture

- The architecture of the underlying CPU defines the instructions and registers available during process execution. Details of the CPU architecture are useful in understanding the architecture-dependent parts of an OS. We will review the basics here.
- Several CPU registers are referenced in the discussion of operating system concepts. While the names and number of registers differ based on the CPU architecture, here are some names of x86 registers that we will find useful. (Note: this list of registers is by no means exhaustive.)
    1. EAX, EBX, ECX, EDX, ESI, and EDI are general purpose registers used to store variables during computations.
    2. The general purpose registers EBP and ESP store the base and top of the current stack frame.
    3. The program counter (PC) is also referred to as EIP (instruction pointer).
    4. The segment registers CS, DS, ES, FS, GS, and SS store pointers to various segments (e.g., code segment, data segment, stack segment) of the process memory.
    5. The control registers like CR0 hold control information. For example, the CR3 register holds the address of the page table of the current running process, which is used to translate virtual addresses to physical addresses.
- Of these registers, EAX, ECX, and EDX are called caller-save registers, and the rest (EBX, ESI, EDI, ESP, EBP, EIP) are callee-save registers. The callee-save registers must be preserved across function calls and context switches. Suppose an executing process pushes a new stack frame onto a stack during a function call, or the CPU moves away to another process. When the function call returns, or the CPU is switched back to the process, the callee-save registers must have the same values as before the event. The ESP register should be pointing to the old stack as before, and EIP should contain the return address from where the process resumes execution. The EAX register is used to pass arguments and return values. The caller save registers may have changed, and no assumptions would be made of them. This convention is followed by compilers and operating systems.

1. Handling Interrupts

- Let us begin by looking at the `proc` data structure (line 2353), that corresponds to the PCB. Especially note the fields corresponding to the kernel stack pointer, the trap frame (which stores context before handling a trap), and the context structure (which stores context during a context switch). The trap frame and context structures are stored on the kernel stack, but a separate pointer is also stored in the PCB for easy access.

- xv6 maintains an array of PCBs in a process table or `ptable`, seen at lines 2409–2412. This process table is protected by a lock—any function that accesses or modifies this process table must hold this lock while doing so. We will study the use of locks later.

- We will now study how a process handles interrupts and system calls in xv6. Interrupts are assigned unique numbers (called IRQ) on every machine (all system calls get the same IRQ). The interrupt descriptor table (IDT) stores information on what to do for every interrupt number, and is setup during initialization (line 3317). The IDT entries for all interrupts point to a generic function to handle all traps (line 3254) that we will examine soon. Even system calls are handled like (hardware) interrupts.

- When an interrupt / program fault / system call occurs in xv6, the interrupting hardware causes the processor to execute the `int` instruction, with a specific interrupt number (IRQ) as argument. This special instruction moves the CPU to kernel mode (if it was in user mode previously). The CPU has a *task state segment* for the current running task, that lets the CPU find the kernel stack for the current process and switch to it. That is, the ESP moves from the old stack (user or kernel) to the kernel stack of the process. As part of the execution of the `int` instruction, the CPU also saves some registers on the kernel stack (pointers to the old code segment, stack segment, program counter etc.), which will eventually form a part of the *trap frame*. Next, the CPU fetches the IDT entry corresponding to the interrupt number, which has a pointer to the interrupt handler. Now, the control is transferred to the kernel, and the CPU starts executing the kernel code that is responsible for handling interrupts. (Note that this switch to the kernel code segment and kernel stack need not happen if the process was already in kernel mode at the time the interrupt occurred.)

- It is important to note that the change in EIP (from user code to the kernel code of interrupt handler) and the change in ESP (from whatever was the old user stack to the kernel stack) must necessarily happen as part of the hardware CPU instruction, and cannot happen once the kernel starts to run (because the kernel can start executing only on the kernel stack and once the EIP points to its code). So it is imperative that some part of the trap handling should be implemented as part of the hardware logic of the `int` instruction.

- The kernel code pointed at by the IDT entry (sheet 32) pushes the interrupt number onto the stack (e.g., line 3233), and completes saving various CPU registers (lines 3256-3260), with the result that the kernel stack now contains a trap frame (line 0602). Note that the registers that would have changed by the time the kernel code has started to run (EIP, ESP etc.) are saved by the CPU hardware instruction, while the registers that would not have changed since

the user execution can be saved by the kernel. The trap frame serves two purposes: it saves the execution context that existed just before the trap (so that the process can resume where it left off), and it provides arguments to handle the trap (e.g., interrupt number). In addition to creating a trap frame, the kernel sets up various segment registers to point to the correct code (lines 3263-3268). Then the stack pointer (which is also a pointer to the trap frame, since the top of the stack is the trap frame) is pushed onto the stack, and the generic function to handle all traps (line 3350) is executed. The trap frame is an argument to this function. This function looks at the number of the interrupt in the trap frame, and executes the corresponding action. For example, if it is a system call, the corresponding system call function is executed.

- After trap returns at `trapret`, (line 3277), the registers that were pushed onto the stack are popped (by the OS in sheet 32, and by the CPU using the `iret` instruction). Thus the context of the process prior to the interrupt is restored, and the process can resume execution.

- If the interrupt was a timer interrupt, the trap function tries to yield the CPU to another process (line 3424). If the scheduler does decide to context switch, the return from trap does not happen right away. Instead, the kernel context of the process is pushed onto the kernel stack by the code that does a context switch (we will read it later), and another process starts executing. The return from trap happens after this process gets the CPU again.

- Note that if the trap handled by a process was an interrupt, it could have lead to some blocked processes being unblocked (e.g., some process waiting for disk data). However, note that the process that serviced the interrupt to unblock a process does not immediately switch to the unblocked process. Instead, it continues execution and gives up the CPU upon a timer interrupt or when it blocks itself. It is important to note that unblocked processes do not run right away, and will wait in the ready state to be scheduled by the scheduler.

- **The contents of the kernel stack are key to understanding the workings of xv6.** When servicing an interrupt, the kernel stack has the trap frame. When the process is switched out by the scheduler after servicing an interrupt, the kernel stack has the trap frame, followed by the process context that is written during the context switch.

- Interrupt handling in most Unix systems is conceptually similar, though more complicated. For example, in Linux, interrupt handling is split into two parts: every interrupt handler has a *top half* and a *bottom half*. The top half consists of the basic necessities that must be performed on receiving an interrupt (e.g., copy packets from the network card's memory to kernel memory), while the bottom half that is executed at a later time does the not-so-time-critical tasks (e.g., TCP/IP processing). Linux also has a separate interrupt context and associated stack that the kernel uses while servicing interrupts, instead of running on the kernel stack of the interrupted process itself.

2. Process creation via fork

- xv6 supports several system calls which are invoked by the trap function that we just studied. You can see the complete set of system calls on sheets 35–36. The system calls themselves could be implemented in other places thorughout the code; the entry function `syscall` (line 3624) is responsible for locating and invoking the suitable system call from the trap function, by looking at the system call number stored in the EAX register. You can also find the helper functions to parse system call arguments on sheet 35.

- We will now study the system call used for process creation. When a process calls fork, the generic trap handling function calls the specific system call fork function (line 2554). Fork calls the allocproc function (line 2455) to allocate an unused proc data structure.

- The allocproc function creates a new process data structure, and allocates a new kernel stack. On the kernel stack of a new process, it pushes a trap frame and a context structure. The fork function copies the parent's trap frame onto the child's trap frame (line 2572). As a result, both child and the parent return to the same point in the program, right after the fork system call. The only difference is that the fork function zeroes out the EAX register in the child's trap frame (line 2575), which is the return value from fork, causing the child to return with the value 0. On the other hand, the parent returns with the pid of the child.

- In addition to the trap frame, the child's kernel stack also has the context data structure. Normally, processes have this context data structure pushed onto the kernel stack when they are being switched out, so that this data structure can be used to restore context when the process has to resume again. For a new process, this context structure is pushed onto the stack by allocproc, so that the scheduler can start scheduling this new process like any other process. The instruction pointer in this (somewhat artificially created) context data structure is set (line 2491) to point to a piece of code called `forkret`, which is where this process starts executing when the scheduler swaps it in. The function forkret mainly calls `trapret` (and does a few other things we'll see later). So, when the child process resumes, it also starts executing the code corresponding to the return from a trap, much like its parent. This is the reason why the child process returns to exactly same instruction after the fork system call, much like the parent, with only a different return value.

- Memory management functions like copying the user space memory image, setting up separate page tables, and other initializations are also done by fork. We will study these later.

3. Process Scheduling

- xv6 uses the following names for process states (line 2350): UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE. The `proc` structures are all stored in a linked list.

- Every CPU has a scheduler thread that calls the `scheduler` function (line 2708) at the start, and loops in it forever. The job of the scheduler function is to look through the list of processes, find a runnable process, set its state to RUNNING, and switch to the process.

- All actions on the list of processes are protected by `ptable.lock`. Any process that wishes to change this data structure must hold the lock, to avoid race conditions (which we will study later). What happens if the lock is not held? It is possible that two CPUs find a process to be runnable, and switch it to running state simultaneously, causing one process to run at two places. So, all actions of the `scheduler` function are always done with the lock held.

- Every process has a `context` structure on its stack (line 2343), that stores the values of the CPU registers that must be saved during a context switch. The registers that are saved as part of the context include the EIP (so that execution can resume at the instruction where it stopped), ESP (so that execution can continue on the stack where it left off), and a few other general purpose registers. To switch the CPU from the current running process P1 to another process P2, the registers of P1 are saved in its context on its stack, and the registers of P2 are reloaded from its context (that would have been saved in the past when P2 was switched out). Now, when P1 must be resumed again, its registers are reloaded from its context, and it can resume execution where it left off.

- The `swtch` function (line 2950) does the job of switching between two contexts, and old one and a new one. The step of pushing registers into the old stack is exactly similar to the step of restoring registers from the new stack, because the new stack was also created by `swtch` at an earlier time. The only time when the `context` structure is not pushed by `swtch` is when a process is created for the first time. For a new process, `allocproc` writes the context onto the new kernel stack (lines 2488-2491), which will be loaded into the CPU registers by `swtch` when the process executes for the first time. All new processes start at `forkret`, so `allocproc` writes this address into the EIP of the context it is creating. Except in this case of a new process, the EIP stored in context is always the address at which the running process invoked `swtch`, and it resumes at the same point at a later time. Note that `swtch` does not explicitly store the EIP to point to the address of the `swtch` statement, but the EIP is automatically stored on the stack as part of making the function call to `swtch`.

- In xv6, the scheduler runs as a separate thread with its own stack. Therefore, context switches happen from the kernel mode of a running process to the scheduler thread, and from the scheduler thread to the new process it has identified. (Other operating systems can switch directly between kernel modes of two processes as well.) As a result, the `swtch` function is called at two places: by the scheduler (line 2728) to switch from the scheduler thread to a new process, and by a running process that wishes to give up the CPU in the function `sched` (line 2766). A running process always gives up the CPU at this call to `swtch` in line 2766, and always

resumes execution at this point at a later time. The only exception is a process running for the first time, that never would have called `swtch` at line 2766, and hence never resumes from there.

- A process that wishes to relinquish the CPU calls the function `sched`. This function triggers a context switch, and when the process is switched back in at a later time, it resumes execution again in `sched` itself. Thus a call to `sched` freezes the execution of a process temporarily. When does a process relinquish its CPU in this fashion? For example, when a timer interrupt occurs and it is deemed that the process has run for too long, the trap function calls `yield`, which in turn calls `sched` (line 2776). The other cases are when a process exits or blocks on an event. We will study these cases in more detail later on.

4. Boot procedure

- The first part of the boot process concerns itself with setting up the kernel code in memory, setting up suitable page tables to translate the kernel's memory addresses, and initializing devices (sheet 12). After the kernel starts running, it starts setting up user processes, starting with the first `init` process in the `userinit` function (line 2502).

- To create the first process, the allocproc function is used to allocate a proc structure, much like in fork. Next, the trap frame of this child process is hand-created (lines 2514-2520) to look like the process encountered a trap right on the first instruction in its memory (i.e., the EIP saved in the trap frame is address 0). As a result, when this process executes, its context is restored from the trap frame, and it starts executing at logical address 0. The userinit function would have set up the page table of the new process, and loaded the "/init" binary at logical address 0. So the CPU starts executing the main function of "/init" (sheet 83).

- The init process sets up the first three file descriptors (stdin, stdout, stderr), and all point to the console. All subsequent processes that are spawned inherit these descriptors. Next, it forks a child, and exec's the shell executable in the child. While the child runs the shell (and forks various other processes), the parent waits and reaps zombies.

- The shell program (sheets 83–88, main function at line 8501) gets the user's input, parses it into a command (`parsecmd` at line 8718), and runs it (`runcmd` at line 8406). For commands that involve multiple sub-commands (e.g., list of commands or pipes), new children are forked and the function `runcmd` is called recursively for each subcommand. The simplest subcommands will eventually call exec when the recursion terminates. Also note the complex manipulations on file descriptors in the pipe command (line 8450).