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. Specifically, will understand how traps are handled in xv6, how processes are created, how scheduling and context switching works, and how the xv6 shell starts up at boot time.

0. Background to understand x86 assembly code in xv6

- We will review the basics of x86 assembly language that are required to understand some simple assembly language code in the xv6 OS. While none of the xv6 programming assignments require you to write assembly code, some basic understanding will help you follow the concepts better. It is not required to read and understand all of this background right away; you can keep coming back to this section as and when you encounter assembly language code in xv6. We will begin with a review of the common x86 registers and instructions used in xv6 code.
- One may wonder: **why does an OS need to have assembly code?** Why can't everything be written in a high-level language like C? Below is a small explanation of the relationship between high-level language code, assembly code, and hardware instructions, specifically in the context of operating systems.
  - Every CPU comes with own architecture: a set of instructions which do specific tasks (mov, add, load, store, compare, jump, and so on) and a set of registers which are used during computations. That is, when the CPU executes the instruction "add register1 register2", what happens in the CPU hardware is defined by the CPU manufacturers. For example, this instruction will read the value in register1, and add it to register2. How does the CPU hardware accomplish this task? The CPU designers would have written code to implement this addition functionality in a hardware description language (HDL) like Verilog. You would have learnt about this in a previous course.
  - Now, given that you have CPU hardware capable to executing a bunch of instructions, software running on this hardware will tell the CPU what instructions exactly to execute in what order to accomplish a useful task. For example, you can write an assembly language program to read a few numbers from the terminal input, add them, and print them out to the screen. This software program will comprise of a sequence of instructions that the underlying hardware can understand and execute. That is, assembly software code consists of hardware instructions that tell the hardware to do certain tasks for you. Obviously, if you change the underlying hardware, the set of instructions will also change, so your assembly code must suitably change too.
  - But of course, writing assembly code is not everyone's cup of tea. This is where high level languages like C come in. You can write code in more understandable language and the C compiler will translate this code into assembly code that your hardware will

understand and execute. The C compiler changes the translated assembly code depending on the underlying hardware architecture, because different architectures have different sets of instructions. So, your C program will compile to different assembly code on different architectures, but you as a C programmer do not have to worry about this. The C compiler will use some standard techniques to translate high level code into assembly.

  - For example, if you write a statement like "int c = a + b;" in your code, it will use some standard template to translate this into assembly code, say like this: "load the integer from memory address of a into register R1, load b into register R2, add R1 and R2 and store it into R3, store R3 back into the memory location of c". Thus, your one line of C code will be translated into one or more lines of assembly code by the compiler, and when you run your executable a.out, all of these instructions will be executed by the hardware to accomplish the task you have written in your C program.
  - Sometimes, you may not like the way the compiler translates a given statement into assembly. For example, maybe you wanted to use registers R4, R5, and R6 in the example above (for whatever reason). Sometimes, you may also want to execute certain instructions that are not easily expressible through a high level language. For example, you may want to switch your stack pointer from one value to another, but there is no C language expression that lets you do this. In such cases, you can write your own assembly code, instead of relying on the assembly code output by the C compiler. Though, for most users, compiler generated assembly will be good enough for most of us.
  - Now, operating system is also a large piece of software. For convenience, OS developers write most OS code in a high level language like C. However, for some parts of the OS code, the developers need more control on the underlying assembly code, and cannot rely on C code. Such small parts of an OS are written in assembly language. This part of the OS must be rewritten for every underlying architecture the OS needs to run on, while the C code can work across all architectures (because it is translated suitably by the compiler). Thus, every OS has an architecture-independent part of its code written in a high level language like C, and a small architecture-dependent part written in assembly. The same holds for xv6 as well.

- **x86 registers.** 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 you 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 pointers to the base and top of the current stack frame. That is, they contain the (virtual) memory address of the base/top of the current stack frame of a process. (More on this later.)
  3. The program counter (PC) is also referred to as EIP (instruction pointer). It stores the (virtual) memory address of the instruction that is currently running on the CPU.

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. We will not study segments in detail, since xv6 does not use segmentation much. It is enough for you to understand that segment registers are changed when moving from user mode to kernel mode and vice versa.
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. You can think of the page table as collection of pointers to (physical addresses of) memory locations in RAM where the memory image of a process is stored. This information is used by the CPU to translate virtual addresses to physical addresses.

- **x86 instructions.** The x86 ISA (instruction set architecture) has several instructions. Some common ones you will encounter when reading xv6 code are described below at a high level.

  1. `mov src dst` instruction moves contents of `src` to `dst`. [1] The source/destination can be a constant value, content of a register (e.g., `%eax`), data present at a memory location whose address is stored in a register (e.g., `(%eax)`), or even data present at a memory address that is located at an offset from the address stored in a register (e.g., `4(%eax)` refers to data at a memory location that is 4 bytes after the memory address stored in `eax`).
  2. `pop` pops the value at the top of the stack (pointed at by ESP) and stores it into the register provided as an argument to `pop`. Similarly, `push` pushes the argument provided onto the top of the stack. Both these instructions automatically update ESP. `pusha` and `popa` push/pop all general purpose registers in one go.
  3. `jmp` jumps or changes PC to the instruction address provided.
  4. `call` makes a function call and jumps to the address of the function provided as an argument, after storing the return address on the stack. The `ret` instruction returns from a function to the return address stored at the top of the stack.
  5. Finally, some of the above instruction names have a letter appended at the end (e.g., `movw`, `pushl`) to indicate the different sizes of registers to be used in different architectures.

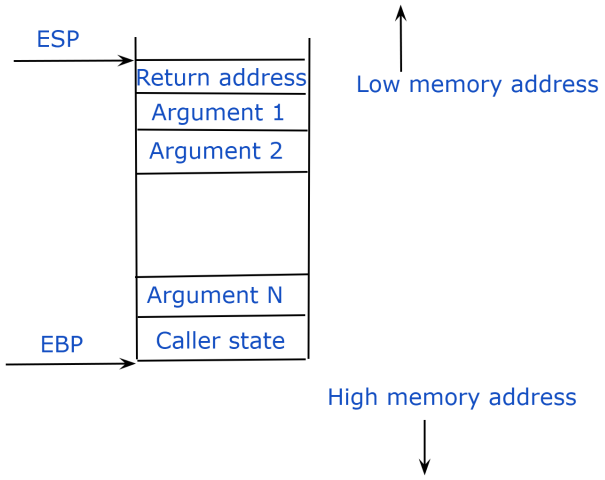
- **What happens on a function call.** We will now understand some basic C language calling conventions in x86, i.e., what happens on the stack when you make a function call. The C compiler does several things on the stack for you, the details of which are beyond the scope of this course, but below is a rough outline.

  1. Some basics first. Processes usually allocate an empty area for the stack, and make ESP/EBP point to the bottom of this memory region, i.e., to a high memory address. As things are pushed onto the stack, the stack grows "upwards", i.e., it starts at a high memory address and moves towards lower (empty) memory addresses. In other words, pushing something onto the stack decreases the value of the memory address stored in ESP. The EBP stays at the bottom of the stack unless explicitly changed.

[1] This is called the AT&T syntax, and this is followed in xv6. A different syntax called the Intel syntax places the destination before the source.

2. You will also need to understand some terminology. Of the various x86 registers, EAX, ECX, and EDX are called caller-save registers, and EBX, ESI, EDI, EBP are called callee-save registers. This classification denotes an agreement between caller and callee during function calls in languages like C, on how to preserve context across function calls. The caller-save registers are saved by the caller, or the entity that makes the function call. The caller does not make any assumptions about the callee preserving the values of these registers, and saves them itself. In contrast, the callee-save registers must be saved by the callee, and restored to their previous values when returning to the caller.

3. To make a function call, the caller (the code making the function call) first saves the caller-save registers onto the stack. Then the caller pushes all the arguments passed to the function onto the stack in the reverse order (from right to left), which also updates the ESP. Now, when you read the contents of the stack from ESP downwards, you will find all the arguments from the top of the stack in left-to-right order. Finally, the `call` instruction is invoked. This instruction pushes the return address onto the stack, and the PC jumps to the function's instructions. Your function call has begun. The stack of the process looks like this now.



4. At this point, the control has been transferred to the callee, or the function that was called. The callee can now choose to push a new "stack frame" onto the stack, to hold all its data that it wishes to store on the stack during the function call. Before pushing the new stack frame, the EBP register points to the base of the previous caller's stack frame, and the ESP register points to the top of the stack. A new stack frame is pushed onto the stack by the callee in the following manner: the old base of stack is saved on the stack (push EBP onto stack), and the new base of the stack is initialized to the old top of the stack (EBP = ESP). At this time, both EBP and ESP point to the same address, which is the top of the old stack frame / base of a new stack frame. Now, anything that is pushed onto the stack changes the value of ESP, while EBP remains at the bottom of this new stack frame.

5. Next, the callee stores local variables, callee-save registers, and whatever else it wishes to store on its stack frame, and begins its execution. The function can access the arguments passed to it by looking below the base of its new stack frame.

6. When the function completes, all the above actions are reversed. The callee pops the things it put on the stack, and invokes the `ret` instruction. This instruction uses the return address at the top of the stack to return to the caller. The caller then pops the things it put on the stack as well before resuming execution of code after the function call.
7. One important thing to note is that the callee places the return value of the function in the EAX register (by convention) before returning to the caller. The caller reads the function return value from this register.
8. All of these actions are done under the hood for you by the C compiler when translating your C code to assembly. However, if you directly write assembly code of a function, you will have to do these things yourself. A high level understanding of this process will be useful when we read assembly code in xv6, as you will see various things getting pushed and popped off the stack.

1. Handling Interrupts

- Let us begin by looking at the `proc` data structure (line 2353), that corresponds to the PCB structure studied in class. Especially note the fields corresponding to the kernel stack pointer, the trapframe, and the context structure. Every process in xv6 has a separate area of memory called the kernel stack that is allocated to it, to be used instead of the userspace stack when running in kernel mode. This memory region is in the kernel part of the RAM, is not accessible when in usermode, and is hence safe to use in kernel mode. When a process moves from usermode to kernel mode, the fields of the trapframe data structure are pushed onto the kernel stack to save user context. On the other hand, when the kernel is switching context from one process in kernel mode to another during a context switch, the fields of the context structure are pushed onto the kernel stack. These two structures are different, because one needs to store different sets of registers in these two situations. For example, the kernel may want to store many more registers in the trapframe to fully understand why the trap occurred, and may choose to only save a small subset of registers that really need to be saved in the context structure. While the trapframe and context structure are already on the kernel stack, the `struct proc` has explicit pointers to the trapframe and context structures on the stack, in addition to the pointer to the whole kernel stack itself, in order to easily locate these structures when needed.

- 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. For example, an interrupt from a keyboard will get an IRQ number that is different from the interrupt from a network card. All system calls are considered as software interrupts and 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). You need not understand the exact structure of the IDT, or how it is constructed, but it is enough to know that in a simple OS like xv6, the IDT entries for all interrupts point to a generic function to handle all traps (line 3254) that we will examine soon. That is, no matter which interrupt occurs, the kernel will start executing at this common function.

- Now, how does control shift from the user code to this all traps function in the kernel? When an interrupt / program fault / system call (henceforth collectively called a **trap**) occurs in xv6, the CPU stops whatever else it is doing and executes the `int n` instruction, with a specific interrupt number (IRQ) as argument. For example, a signal from an external I/O device can cause the CPU to execute this instruction. In the case of system calls, the user program explicitly invokes this `int n`—you may never have realized this because users almost never call system calls directly, and only call C library functions. The C library function internally invokes this `int n` instruction for you to make a system call. So, someone, either an external hardware device, or the user herself, must invoke the `int n` instruction to begin the processing of a trap event.

- What happens to the CPU when it runs this trap instruction? This special trap 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, old stack segment, old program counter etc.), which will eventually form a part of the trapframe. Next, the CPU fetches the IDT entry corresponding to the interrupt number, which has a pointer to the kernel trap handling code. Now, the control is transferred to the kernel, and the CPU starts executing the kernel code that is responsible for handling traps. [2] You will not find the code for what was described in the previous few sentences in xv6. Why? Because all of this is done by the CPU hardware as part of the `int n` instruction, and not by software. That is, this logic is built into your CPU processor hardware, to be run when the special trap instruction is executed.

- 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 be done by the kernel software, because the kernel can start executing only on the kernel stack and once the EIP points to its code. Therefore, it is imperative that some part of the trap handling should be implemented as part of the hardware logic of the `int` instruction. Someone has to switch control to kernel before it can run, and that "someone" is the CPU here.

- 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 trapframe (line 0602). Note that the trapframe has been built collaboratively by the CPU hardware and the kernel. Some parts of the trapframe (e.g., EIP) would have been pushed onto the stack even before the kernel code started to run, and the rest of the fields are pushed by this alltraps function in the kernel. Look at the trapframe structure on sheet 6, and understand how the bottom parts of the trapframe would have been pushed by the CPU and the top parts by the kernel. The trapframe 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 information to the kernel to handle the trap. In addition to creating a trapframe on the stack, the kernel does a few other things like setting up various segment registers to execute correctly in kernel mode (lines 3263-3268). Then the main C function to handle traps (defined at line 3350) is invoked at line 3272. But before this function is called, the stack pointer (which is also a pointer to the trapframe, since the trapframe is at the top of the stack) is pushed onto the stack, as an argument to this trap handling function.

- The main logic to handle all traps is in the C function `trap` starting at line 3350. This function takes the trapframe as an argument. This function looks at the IRQ number of the interrupt in the trapframe, and executes the corresponding action. For example, if it is a system call, the corresponding system call function is executed. If the trap is from the disk, the disk device driver is called, and so on. We will keep coming back to this function often in the course.

[2] 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.

After handling the trap suitably, this trap function returns back to line 3272 from where it was invoked.

- After trap returns, it executes the logic at `trapret`, (line 3277). The registers that were pushed onto the stack are popped by the OS. Then the kernel invokes the `iret` instruction which is basically the inverse of the `int` instruction, and pops the registers that were pushed by the `int` instruction. Thus the context of the process prior to the interrupt is restored, and the userspace process can resume execution. Trap handling is complete!

- It is important to understand one particular aspect of the main trap handling function. In particular, pay attention to what happens if the interrupt was a timer interrupt. In this case, the trap function tries to yield the CPU to another process (line 3424). If the scheduler does decide to context switch away from this process, the return from trap to the userspace of the process 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. We will understand context switches in detail later on.

- 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 trapframe, as we have just seen. When the process is switched out by the scheduler after servicing an interrupt, we will study later that the kernel stack has the trapframe followed by the context structure that stores context 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 the fork system call

- xv6 supports several system calls, which are handled by the trap function we just studied. You can see the complete set of system calls on sheets 35–36. What happens during a system call? The system calls that user programs in xv6 can use are defined in the file `user.h`, which is the header file for the userspace C library of xv6 (xv6 doesn't have the standard glibc). You can find this file in the xv6 code tarball (it is not part of the kernel code PDF document because it is part of the user library of xv6, and not the kernel). User programs can invoke these library functions defined in `user.h` to make system calls. Next, you will find that the code in `usys.S` of the tarball converts the user library function calls to system calls by invoking the `int` instruction with a suitable argument indicating that the trap is due to a system call. Now, getting back to the kernel code, the main trap handling function at line 3350 looks at this argument passed to `int`, realizes that this trap is a system call (line 3353), and calls the common system call processing function `syscall` (line 3624) in the kernel. How does the kernel know which specific system call was invoked? If you would have carefully noticed the user library code in `usys.S`, you would have seen that the system call number is pushed into the EAX register. The common syscall function looks at this syscall number and invokes the specific function corresponding to that particular system call. Note that the system calls themselves could be implemented in other places thorughout the code; the entry function syscall is responsible for locating the suitable system call functions by storing the function pointers in an array.

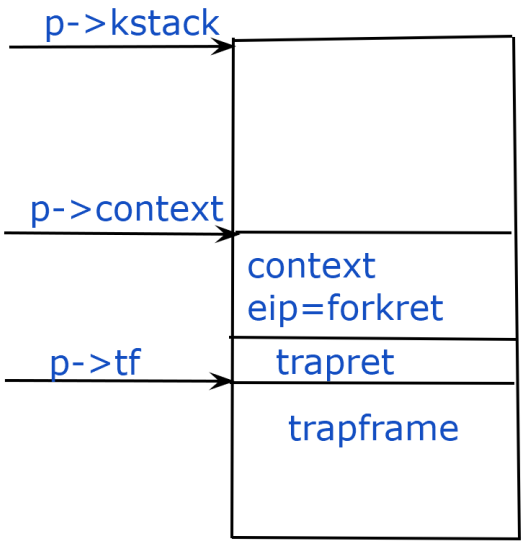
- How are arguments passed to system calls? The user library stores arguments on the userspace stack before invoking the `int` instruction. The system call functions locate the userspace stack (from the ESP stored in the trapframe), and fetch the system call arguments from there. You can also find the various 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. These two functions will now be discussed in detail.

- The allocproc function allocates a new unused `struct proc` data structure, and initializes it (lines 2460-2465). It also allocates a new kernel stack for the process (line 2473). On the kernel stack of a new process, it pushes various things, beginning with a trapframe. We will first see what goes into the trapframe of the new process. Once allocproc returns, the fork function copies the parent's trapframe onto the child's trapframe (line 2572). As a result, both child and the parent have the same user context stored at the top of the stack during trapret, and hence return to the same point in the user program, right after the fork system call. The only difference is that the fork function zeroes out the EAX register in the child's trapframe (line 2575), which is the return value from fork, causing the child to return with the value 0. On the other hand, the system call in the parent returns with the pid of the child (line 2591).

- In addition to the trapframe, the allocproc function pushes the address of trapret on the stack (line 2486), followed by the context structure (line 2488). That is, the function starts at the bottom of the kernel stack in the beginning (line 2477), and gradually moves upwards to make

space for a trapframe, a return address of trapret, and a context structure on the stack. So, when you start popping the kernel stack from top, you will first find the context structure, followed by a return address of trapret, followed by the trapframe. Below is how the kernel stack of the child will look like after allocproc.



- Let us first understand the purpose of the context structure on the kernel stack of the child. We will study more on this structure later, but for now, you should know that processes have this context data structure pushed onto the kernel stack when they are being switched out by the CPU scheduler, so that the contents in this data structure can be used to restore context when the process has to resume again. For example, this context structure saves the EIP at which a process stopped execution just before the context switch, so that it can resume execution again from the same EIP where it stopped. For a new-born process that was never context switched out, a context structure does not make sense. However, allocproc creates an artificial context structure on the kernel stack to make this process appear like it was context switched out in the past, so that when the CPU scheduler sees this process, it can start scheduling this new process like any other process in the system that it had context switched out in the past. This neat little trick means that once a new process is created and added to the list of active processes in the system, the CPU scheduler can treat it like any other process.

- Now, what should the value of the EIP in this context structure be set to? Where do we want our new process to begin execution when the CPU scheduler restores this context and resumes its execution? The instruction pointer in this (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 (line 2783) does a few things which we will study later, and returns. When forkret returns, the process continues to execute at `trapret` because the return address of trapret is now present at the top of the stack (after the context structure has been popped off the stack when restoring context by the CPU scheduler),

and we know that the address at the top of the stack is used as the return address when returning from a function call. That is, in effect, a newly created process, when it is scheduled on the CPU for the first time, simply behaves as if it is returning from a trap.

- We have seen earlier that the trapret function simply pops the trapframe from the kernel stack, restores userspace context and returns the process to usermode again. The child's trapframe is an exact copy of the parent's trapframe, and the parent's trapframe stores the userspace context of the process that stopped execution at the fork system call. So, when the child process resumes, it returns to the exact same instruction after the fork system call, much like the parent, with only a different return value. Now you should be able to understand why the parent and child processes resume execution at the same line after a fork system call, and the effort that has gone in to make this appear so easy.
- The fork system call does many other things like copying the user space memory image from parent to child and other initializations. We will study these later.

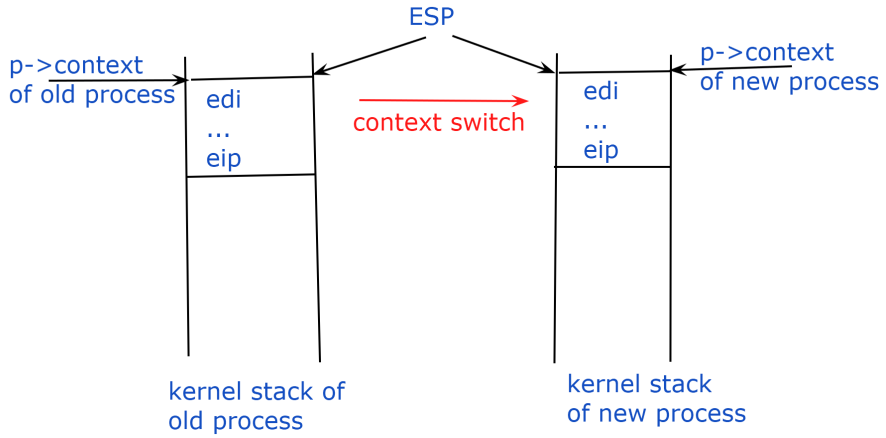
3. Process Scheduling

- xv6 uses the following names for process states (line 2350): UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE. The `proc` structures are all stored in the `ptable` array.
- We will now study how the CPU scheduler works in xv6 (pages 27–29). Every CPU in the machine starts a scheduler thread (think of it as a special process) that finishes some basic bootup activities and calls the `scheduler` function (line 2708) when the machine comes up. This scheduler thread loops in this function forever, finding active processes to run and context switching to them. 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. We will understand this process in detail.
- First, note that 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 this lock held. Always remember: one must acquire the lock, change the process table, and release the lock.
- During a context switch, 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), 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 structure on its stack, and the registers of P2 are reloaded from its context structure (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 this job of switching between two contexts, and old one and a new one. Let us carefully study this process of context switching. Understanding this function needs some basic knowledge of x86 assembly.
  - For example, consider the scenario when the scheduler finds a process to run and decides to context switch to it (line 2728). At this point, it calls the swtch function, providing to it two arguments: a pointer to the context structure of the scheduler thread itself (where context must be saved), and a pointer to the context structure of the new process to run (from where context should be restored), which is available on the kernel stack of the new process.
  - Next, look at the code of `swtch` starting at line 2950. This code is in asembly language, so we need to understand the C calling conventions to understand what is on the stack when this function starts. The top of the stack will have the return address from where swtch was invoked, and below this return address would be the two arguments: the old context pointer and the new context pointer. The function first saves the old context pointer and

the new context pointer into two registers (lines 2959, 2960). That is, EAX has the old context pointer and EDX has the new context pointer.

- Right now, we are still operating on the stack of the old process from which we wish to move away. Next, the function pushes some registers onto this old stack (lines 2963-2966). Note that because this function is the callee, we are only saving the callee-save registers (the other caller-save registers would have been saved by the compiled C code of the caller). The registers pushed by this code, along with the EIP (i.e., the return address pushed on the stack when the funtion call to swtch was made), together will form a context structure on the stack.
- After pushing the context onto the old stack, the stack pointer still points to the top of the old stack, and this top of the stack contains the context structure. The pointer to the old context (which was saved in EAX) is now updated to point to this saved context at the top of the stack (line 2969). Notice the subtlety around the indirect addressing mode in line 2969 (there are parantheses around EAX indicating that you don't store anything directly into EAX but rather at the address pointed by EAX). That is, you are not writing anything into EAX, rather, you are going to the memory location pointed at by EAX (the old context pointer), and overwriting it with the updated context pointer at the top of the stack.
- Having saved the old context, the stack pointer now shifts to point to the top of the stack of the new process, which has a new context structure to restore. Recall that we saved this new context structure pointer in EDX. So, this value of EDX is moved to ESP (line 2970). The stack pointer has now switched from the stack of the old process to that of the new process. A context switch has happened!
- Next, we pop the registers of the context structure from the top of the new stack (lines 2973-2976), and return from swtch into the kernel mode of a new process.

The process of this context switch is illustrated in this figure below.



- Note that, in the swtch function, 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 we switch to a context structure that was not pushed by swtch is when we run a newly created process. 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 (return address) is automatically pushed 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) and switch to the scheduler thread. 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 (line 2753). 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 a system call. We will study these cases in more detail later on.

4. Boot procedure

- We will now understand how xv6 boots up. 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), and we will study this in detail later. After the kernel starts running, it starts setting up user processes, starting with the first `init` process in the `userinit` function (line 2502). We will understand the boot procedure from the creation of init process onwards.

- To create the first process, the allocproc function is used to allocate a new proc structure, much like in fork (line 2507). The memory image of the new process is initialized with the compiled code of the init program in line 2511 (we will study this later). We have seen earlier that allocproc builds the kernel stack of this new process in such a way that it runs forkret, and then begins returning from trap. What will happen when this new process returns from trap? The trapframe of this init 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 trapframe is address 0). As a result, when this process returns from trap, its context is restored from the trapframe, and it starts executing at the start of the userspace init program (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).