Chapter 2

Processes

One of an operating system's central roles is to allow multiple programs to share the CPUs and main memory safely, isolating them so that one errant program cannot break others. To that end, xv6 provides the concept of a process, as described in Chapter 0. This chapter examines how xv6 allocates memory to hold process code and data, how it creates a new process, and how it configures the processor's paging hardware to give each process the illusion that it has a private memory address space. The next few chapters will examine how xv6 uses hardware support for interrupts and context switching to create the illusion that each process has its own private CPU.

Address Spaces

xv6 ensures that each process can only read and write the memory that xv6 has allocated to it, and not for example the kernel's memory or the memory of other processes. xv6 also arranges for each process's memory to be contiguous and to start at virtual address zero. The C language definition and the Gnu linker expect process memory to be contiguous. Process memory starts at zero because that is traditional. A process's view of memory is called an address space.

x86 protected mode defines three kinds of addresses. Executing software uses virtual addresses when fetching instructions or reading and writing memory. The segmentation hardware translates virtual to linear addresses. Finally, the paging hardware (when enabled) translates linear to physical addresses. Software cannot directly use a linear or physical address. xv6 sets up the segmentation hardware so that virtual and linear addresses are always the same: the segment descriptors all have a base of zero and the maximum possible limit. xv6 sets up the x86 paging hardware to translate (or "map") linear to physical addresses in a way that implements process address spaces with the properties outlined in the previous paragraph.

The paging hardware uses a page table to translate linear to physical addresses. A page table is logically an array of 2^20 (1,048,576) page table entries (PTEs). Each PTE contains a 20-bit physical page number (PPN) and some flags. The paging hardware translates a linear address by using its top 20 bits to index into the page table to find a PTE, and replacing those bits with the PPN in the PTE. The paging hardware copies the low 12 bits unchanged from the linear to the translated physical address. Thus a page table gives the operating system control over linear-to-physical address translations at the granularity of aligned chunks of 4096 (2^12) bytes.

Each PTE contains flag bits that tell the paging hardware to restrict how the associated linear address is used. `PTE_P` controls whether the PTE is valid: if it is not set,

a reference to the page causes a fault (i.e. is not allowed). PTE_W controls whether instructions are allowed to issue writes to the page; if not set, only reads and instruction fetches are allowed. PTE_U controls whether user programs are allowed to use the page; if clear, only the kernel is allowed to use the page.

A few notes about terms. Physical memory refers to storage cells in DRAM. A byte of physical memory has an address, called a physical address. A program uses virtual addresses, which the segmentation and paging hardware translates to physical addresses, and then sends to the DRAM hardware to read or write storage. At this level of discussion there is no such thing as virtual memory, only virtual addresses. Because xv6 sets up segments to make virtual and linear addresses always identical, from now on we'll stop distinguishing between them and use virtual for both.

xv6 uses page tables to implement process address spaces as follows. Each process has a separate page table, and xv6 tells the page table hardware to switch page tables when xv6 switches between processes. A process's memory starts at virtual address zero and can have size of at most 640 kilobytes (160 pages). xv6 sets up the PTEs for the process's virtual addresses to point to whatever pages of physical memory xv6 has allocated for the process's memory, and sets the PTE_U, PTE_W, and PTE_P flags in these PTEs. If a process has asked xv6 for less than 640 kilobytes, xv6 will leave PTE_P clear in the remainder of the first 160 PTEs.

Different processes' page tables translate the first 160 pages to different pages of physical memory, so that each process has private memory. However, xv6 sets up every process's page table to translate virtual addresses above 640 kilobytes in the same way. To a first approximation, all processes' page tables map virtual addresses above 640 kilobytes directly to physical addresses, which makes it easy to address physical memory. However, xv6 does not set the PTE_U flag in the PTEs above 640 kilobytes, so only the kernel can use them. For example, the kernel can use its own instructions and data (at virtual/physical addresses starting at one megabyte). The kernel can also read and write the physical memory beyond the end of its data segment.

Every process's page table simultaneously contains translations for both all of the process's memory and all of the kernel's memory. This setup allows system calls and interrupts to switch between a running process and the kernel without having to switch page tables. For the most part the kernel does not have its own page table; it is almost always borrowing some process's page table. The price paid for this convenience is that the sum of the size of the kernel and the largest process must be less than four gigabytes on a machine with 32-bit addresses.

To review, xv6 ensures that each process can only use its own memory, and that a process sees its memory as having contiguous virtual addresses. xv6 implements the first by setting the PTE_U bit only on PTEs of virtual addresses that refer to the process's own memory. It implements the second using the ability of page tables to translate a virtual address to a different physical address.

Memory allocation

xv6 needs to allocate physical memory at run-time to store its own data structures and to store processes' memory. There are three main questions to be answered

when allocating memory. First, what physical memory (i.e. DRAM storage cells) are to be used? Second, at what virtual address or addresses is the newly allocated physical memory to be mapped? And third, how does xv6 know what physical memory is free and what memory is already in use?

xv6 maintains a pool of physical memory available for run-time allocation. It uses the physical memory beyond the end of the loaded kernel's data segment. xv6 allocates (and frees) physical memory at page (4096-byte) granularity. It keeps a linked list of free physical pages; xv6 deletes newly allocated pages from the list, and adds freed pages back to the list.

When the kernel allocates physical memory that only it will use, it does not need to make any special arrangement to be able to refer to that memory with a virtual address: the kernel sets up all page tables so that virtual addresses map directly to physical addresses for addresses above 640 KB. Thus if the kernel allocates the physical page at physical address 0x200000 for its internal use, it can use that memory via virtual address 0x200000 without further ado.

What if a process allocates memory with `sbrk`? Suppose that the current size of the process is 12 kilobytes, and that xv6 finds a free page of physical memory at physical address 0x201000. In order to ensure that process memory remains contiguous, that physical page should appear at virtual address 0x3000 when the process is running. This is the time (and the only time) when xv6 uses the paging hardware's ability to translate a virtual address to a different physical address. xv6 modifies the 3rd PTE (which covers virtual addresses 0x3000 through 0x3fff) in the process's page table to refer to physical page number 0x201 (the upper 20 bits of 0x201000), and sets `PTE_U`, `PTE_W`, and `PTE_P` in that PTE. Now the process will be able to use 16 kilobytes of contiguous memory starting at virtual address zero. Two different PTEs now refer to the physical memory at 0x201000: the PTE for virtual address 0x201000 and the PTE for virtual address 0x3000. The kernel can use the memory with either of these addresses; the process can only use the second.

Code: Memory allocator

The xv6 kernel calls `kalloc` and `kfree` to allocate and free physical memory at run-time. The kernel uses run-time allocation for process memory and for these kernel data strucures: kernel stacks, pipe buffers, and page tables. The allocator manages page-sized (4096-byte) blocks of memory.

The allocator maintains a *free list* of addresses of physical memory pages that are available for allocation. Each free page's list element is a `struct run` (2360). Where does the allocator get the memory to hold that data structure? It store each free page's `run` structure in the free page itself, since there's nothing else stored there. The free list is protected by a spin lock (2360-2362). The list and the lock are wrapped in a struct to make clear that the lock protects the fields in the struct. For now, ignore the lock and the calls to `acquire` and `release`; Chapter 4 will examine locking in detail.

`Main` calls `kinit` to initialize the allocator (2371). `kinit` ought to determine how much physical memory is available, but this turns out to be difficult on the x86. Instead it assumes that the machine has 16 megabytes (`PHYSTOP`) of physical memory,

and uses all the memory between the end of the kernel and PHYSTOP as the initial pool of free memory. kinit uses the symbol end, which the linker causes to have an address that is just beyond the end of the kernel's data segment.

Kinit (2371) calls kfree with the address of each page of memory between end and PHYSTOP. This will cause kfree to add those pages to the allocator's free list. A PTE can only refer to a physical address that is aligned on a 4096-byte boundary (is a multiple of 4096), so kinit uses PGROUNDUP to ensure that it frees only aligned physical addresses. The allocator starts with no memory; these initial calls to kfree gives it some to manage.

Kfree (2405) begins by setting every byte in the memory being freed to the value 1. This will cause code that uses memory after freeing it (uses "dangling references") to read garbage instead of the old valid contents; hopefully that will cause such code to break faster. Then kfree casts v to a pointer to struct run, records the old start of the free list in r->next, and sets the free list equal to r. Kalloc removes and returns the first element in the free list.

Code: Page Table Initialization

mainc (1354) creates a page table for the kernel's use with a call to kvmalloc, and mpmain (1380) causes the x86 paging hardware to start using that page table with a call to vmenable. This page table maps most virtual addresses to the same physical address, so turning on paging with it in place does not disturb execution of the kernel.

kvmalloc (2576) calls setupkvm and stores a pointer to the resulting page table in kpgdir, since it will be used later.

An x86 page table is stored in physical memory, in the form of a 4096-byte "page directory" that contains 1024 PTE-like references to "page table pages." Each page table page is an array of 1024 32-bit PTEs. The paging hardware uses the top 10 bits of a virtual address to select a page directory entry. If the page directory entry is marked PTE_P, the paging hardware uses the next 10 bits of the virtual address to select a PTE from the page table page that the page directory entry refers to. If either of the page directory entry or the PTE has no PTE_P, the paging hardware raises a fault. This two-level structure allows a page table to omit entire page table pages in the common case in which large ranges of virtual addresses have no mappings.

setupkvm allocates a page of memory to hold the page directory. It then calls mappages to install translations for ranges of memory that the kernel will use; these translations all map each virtual address to the same physical address. The translations include the kernel's instructions and data, physical memory up to PHYSTOP, and memory ranges which are actually I/O devices. setupkvm does not install any mappings for the process's memory; this will happen later.

mappages (2531) installs mappings into a page table for a range of virtual addresses to a corresponding range of physical addresses. It does this separately for each virtual address in the range, at page intervals. For each virtual address to be mapped, mappages calls walkpgdir to find the address of the PTE that should the address's translation. It then initializes the PTE to hold the relevant physical page number, the desired permissions (PTE_W and/or PTE_U), and PTE_P to mark the PTE as valid (2542).

`walkpgdir` (2504) mimics the actions of the x86 paging hardware as it looks up the PTE for a virtual address. It uses the upper 10 bits of the virtual address to find the page directory entry (2510). If the page directory entry isn't valid, then the required page table page hasn't yet been created; if the `create` flag is set, `walkpgdir` goes ahead and creates it. Finally it uses the next 10 bits of the virtual address to find the address of the PTE in the page table page (2524). The code uses the physical addresses in the page directory entries as virtual addresses. This works because the kernel allocates page directory pages and page table pages from an area of physical memory (between the end of the kernel and `PHYSTOP`) for which the kernel has direct virtual to physical mappings.

`vmenable` (2602) loads `kpgdir` into the x86 `%cr3` register, which is where the hardware looks for the physical address of the current page directory. It then sets `CR0_PG` in `%cr0` to enable paging.

Code: Process creation

This section describes how xv6 creates the very first process. The xv6 kernel maintains many pieces of state for each process, which it gathers into a `struct proc` (1721). A process's most important pieces of kernel state are its page table and the physical memory it refers to, its kernel stack, and its run state. We'll use the notation `p->xxx` to refer to elements of the `proc` structure.

You should view the kernel state of a process as a thread that executes in the kernel on behalf of a process. For example, when a process makes a system call, the CPU switches from executing the process to executing the process's kernel thread. The process's kernel thread executes the implementation of the system call (e.g., reads a file), and then returns back to the process.

`p->pgdir` holds the process's page table, an array of PTEs. xv6 causes the paging hardware to use a process's `p->pgdir` when executing that process. A process's page table also serves as the record of the addresses of the physical pages allocated to store the process's memory.

`p->kstack` points to the process's kernel stack. When a process's kernel thread is executing, for example in a system call, it must have a stack on which to save variables and function call return addresses. xv6 allocates one kernel stack for each process. The kernel stack is separate from the user stack, since the user stack may not be valid. Each process has its own kernel stack (rather than all sharing a single stack) so that a system call may wait (or "block") in the kernel to wait for I/O, and resume where it left off when the I/O has finished; the process's kernel stack saves much of the state required for such a resumption.

`p->state` indicates whether the process is allocated, ready to run, running, waiting for I/O, or exiting.

The story of the creation of the first process starts when `mainc` (1363) calls `userinit` (1902), whose first action is to call `allocproc`. The job of `allocproc` (1854) is to allocate a slot (a `struct proc`) in the process table and to initialize the parts of the process's state required for its kernel thread to execute. `Allocproc` is called for all new processes, while `userinit` is only called for the very first process. `Allocproc`

scans the table for a process with state UNUSED (1819-1862). When it finds an unused process, allocproc sets the state to EMBRYO to mark it as used and gives the process a unique pid (1808-1868). Next, it tries to allocate a kernel stack for the process's kernel thread. If the memory allocation fails, allocproc changes the state back to UNUSED and returns zero to signal failure.

Now allocproc must set up the new process's kernel stack. Ordinarily processes are only created by fork, so a new process starts life copied from its parent. The result of fork is a child process that has identical memory contents to its parent. allocproc sets up the child to start life running its kernel thread, with a specially prepared kernel stack and set of kernel registers that cause it to "return" to user space at the same place (the return from the fork system call) as the parent. allocproc does part of this work by setting up return program counter values that will cause the new process's kernel thread to first execute in forkret and then in trapret (1885-1890). The kernel thread will start executing with register contents copied from p->context. Thus setting p->context->eip to forkret will cause the kernel thread to execute at the start of forkret (2183). This function will return to whatever address is at the bottom of the stack. The context switch code (2308) sets the stack pointer to point just beyond the end of p->context. allocproc places p->context on the stack, and puts a pointer to trapret just above it; that is where forkret will return. trapret restores user registers from values stored at the top of the kernel stack and jumps into the process (2929). This setup is the same for ordinary fork and for creating the first process, though in the latter case the process will start executing at location zero rather than at a return from fork.

As we will see in Chapter 3, the way that control transfers from user software to the kernel is via an interrupt mechanism, which is used by system calls, interrupts, and exceptions. Whenever control transfers into the kernel while a process is running, the hardware and xv6 trap entry code save user registers on the top of the process's kernel stack. userinit writes values at the top of the new stack that look just like those that would be there if the process had entered the kernel via an interrupt (1914-1920), so that the ordinary code for returning from the kernel back to the process's user code will work. These values are a struct trapframe which stores the user registers.

Here is the state of the new process's kernel stack:

```
 ----------  <-- top of new process's kernel stack
| esp      |
| ...      |
| eip      |
| ...      |
| edi      | <-- p->tf (new proc's user registers)
| trapret  | <-- address forkret will return to
| eip      |
| ...      |
| edi      | <-- p->context (new proc's kernel registers)
|          |
| (empty)  |
|          |
 ----------  <-- p->kstack
```

The first process is going to execute a small program (initcode.S; (7200)). The

process needs physical memory in which to store this program, the program needs to be copied to that memory, and the process needs a page table that refers to that memory.

`userinit` calls `setupkvm` (2583) to create a page table for the process with (at first) mappings only for memory that the kernel uses.

The initial contents of the first process's memory are the compiled form of `initcode.S`; as part of the kernel build process, the linker embeds that binary in the kernel and defines two special symbols `_binary_initcode_start` and `_binary_initcode_size` telling the location and size of the binary (XXX sidebar about why it is extern char[]). `Userinit` copies that binary into the new process's memory by calling `inituvm`, which allocates one page of physical memory, maps virtual address zero to that memory, and copies the binary to that page (2666). Then `userinit` sets up the trap frame with the initial user mode state: the `cs` register contains a segment selector for the `SEG_UCODE` segment running at privilege level `DPL_USER` (i.e., user mode not kernel mode), and similarly `ds`, `es`, and `ss` use `SEG_UDATA` with privilege `DPL_USER`. The `eflags FL_IF` is set to allow hardware interrupts; we will reexamine this in Chapter 3. The stack pointer `esp` is the process's largest valid virtual address, `p->sz`. The instruction pointer is the entry point for the initcode, address 0. Note that `initcode` is not an ELF binary and has no ELF header. It is just a small headerless binary that expects to run at address 0, just as the boot sector is a small headerless binary that expects to run at address `0x7c00`. `Userinit` sets `p->name` to `initcode` mainly for debugging. Setting `p->cwd` sets the process's current working directory; we will examine `namei` in detail in Chapter 7.

Once the process is initialized, `userinit` marks it available for scheduling by setting `p->state` to `RUNNABLE`.

Code: Running a process

Now that the first process's state is prepared, it is time to run it. After `main` calls `userinit`, `mpmain` calls `scheduler` to start running processes (1384). `Scheduler` (2108) looks for a process with `p->state` set to `RUNNABLE`, and there's only one it can find: `initproc`. It sets the per-cpu variable `proc` to the process it found and calls `switchuvm` to tell the hardware to start using the target process's page table (2636). Changing page tables while executing in the kernel works because `setupkvm` causes all processes' page tables to have identical mappings for kernel code and data. `switchuvm` also creates a new task state segment `SEG_TSS` that instructs the hardware to handle an interrupt by returning to kernel mode with `ss` and `esp` set to `SEG_KDATA<<3` and `(uint)proc->kstack+KSTACKSIZE`, the top of this process's kernel stack. We will reexamine the task state segment in Chapter 3.

`scheduler` now sets `p->state` to `RUNNING` and calls `swtch` (2308) to perform a context switch to the target process's kernel thread. `swtch` saves the current registers and loads the saved registers of the target kernel thread (`proc->context`) into the x86 hardware registers, including the stack pointer and instruction pointer. The current context is not a process but rather a special per-cpu scheduler context, so `scheduler` tells `swtch` to save the current hardware registers in per-cpu storage (`cpu->sched-`

uler) rather than in any process's kernel thread context. We'll examine switch in more detail in Chapter 5. The final ret instruction (2327) pops a new eip from the stack, finishing the context switch. Now the processor is running the kernel thread of process p.

Allocproc set initproc's p->context->eip to forkret, so the ret starts executing forkret. Forkret (2183) releases the ptable.lock (see Chapter 4) and then returns. Allocproc arranged that the top word on the stack after p->context is popped off would be trapret, so now trapret begins executing, with %esp set to p->tf. Trapret (2929) uses pop instructions to walk up the trap frame just as swtch did with the kernel context: popal restores the general registers, then the popl instructions restore %gs, %fs, %es, and %ds. The addl skips over the two fields trapno and errcode. Finally, the iret instructions pops %cs, %eip, and %eflags off the stack. The contents of the trap frame have been transferred to the CPU state, so the processor continues at the %eip specified in the trap frame. For initproc, that means virtual address zero, the first instruction of initcode.S.

At this point, %eip holds zero and %esp holds 4096. These are virtual addresses in the process's address space. The processor's paging hardware translates them into physical addresses (we'll ignore segments since xv6 sets them up with the identity mapping (2465)). allocuvm set up the PTE for the page at virtual address zero to point to the physical memory allocated for this process, and marked that PTE with PTE_U so that the process can use it. No other PTEs in the process's page table have the PTE_U bit set. The fact that userinit (1914) set up the low bits of %cs to run the process's user code at CPL=3 means that the user code can only use PTE entries with PTE_U set, and cannot modify sensitive hardware registers such as %cr3. So the process is constrained to using only its own memory.

Initcode.S (7207) begins by pushing three values on the stack—$argv, $init, and $0—and then sets %eax to $SYS_exec and executes int $T_SYSCALL: it is asking the kernel to run the exec system call. If all goes well, exec never returns: it starts running the program named by $init, which is a pointer to the NUL-terminated string /init (7220-7222). If the exec fails and does return, initcode loops calling the exit system call, which definitely should not return (7214-7218).

The arguments to the exec system call are $init and $argv. The final zero makes this hand-written system call look like the ordinary system calls, as we will see in Chapter 3. As before, this setup avoids special-casing the first process (in this case, its first system call), and instead reuses code that xv6 must provide for standard operation.

The next chapter examines how xv6 configures the x86 hardware to handle the system call interrupt caused by int $T_SYSCALL. The rest of the book builds up enough of the process management and file system implementation to finally implement exec in Chapter 9.

Real world

Most operating systems have adopted the process concept, and most processes look similar to xv6's. A real operating system would find free proc structures with an

explicit free list in constant time instead of the linear-time search in `allocproc`; xv6 uses the linear scan (the first of many) for simplicity.

Like most operating systems, xv6 uses the paging hardware for memory protection and mapping and mostly ignores segmentation. Most operating systems make far more sophisticated use of paging than xv6; for example, xv6 lacks demand paging from disk, copy-on-write fork, shared memory, and automatically extending stacks. xv6 does use segments for the common trick of implementing per-cpu variables such as `proc` that are at a fixed address but have different values on different CPUs. Implementations of per-CPU (or per-thread) storage on non-segment architectures would dedicate a register to holding a pointer to the per-CPU data area, but the x86 has so few general registers that the extra effort required to use segmentation is worthwhile.

xv6's address space layout is awkward. The user stack is at a relatively low address and grows down, which means it cannot grow very much. User memory cannot grow beyond 640 kilobytes. Most operating systems avoid both of these problems by locating the kernel instructions and data at high virtual addresses (e.g. starting at 0x80000000) and putting the top of the user stack just beneath the kernel. Then the user stack can grow down from high addresses, user data (via `sbrk`) can grow up from low addresses, and there is hundreds of megabytes of growth potential between them. It is also potentially awkward for the kernel to map all of physical memory into the virtual address space; for example that would leave zero virtual address space for user mappings on a 32-bit machine with 4 gigabytes of DRAM.

In the earliest days of operating systems, each operating system was tailored to a specific hardware configuration, so the amount of memory could be a hard-wired constant. As operating systems and machines became commonplace, most developed a way to determine the amount of memory in a system at boot time. On the x86, there are at least three common algorithms: the first is to probe the physical address space looking for regions that behave like memory, preserving the values written to them; the second is to read the number of kilobytes of memory out of a known 16-bit location in the PC's non-volatile RAM; and the third is to look in BIOS memory for a memory layout table left as part of the multiprocessor tables. None of these is guaranteed to be reliable, so modern x86 operating systems typically augment one or more of them with complex sanity checks and heuristics. In the interest of simplicity, xv6 assumes that the machine it runs on has at least 16 megabytes of memory. A real operating system would have to do a better job.

Memory allocation was a hot topic a long time ago, the basic problems being efficient use of very limited memory and preparing for unknown future requests. See Knuth. Today people care more about speed than space-efficiency. In addition, a more elaborate kernel would likely allocate many different sizes of small blocks, rather than (as in xv6) just 4096-byte blocks; a real kernel allocator would need to handle small allocations as well as large ones.

Exercises

1. Set a breakpoint at swtch. Single step with gdb's `stepi` through the ret to `forkret`, then use gdb's `finish` to proceed to `trapret`, then `stepi` until you get to `initcode`

at virtual address zero.

2. Look at real operating systems to see how they size memory.