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. xv6 implements a process as a set of data structures, but a process is quite special: it comes alive with help from the hardware. 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 segmentation hardware to give each process the illusion that it hash its own 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.

Code: Memory allocation

xv6 allocates most of its data structures statically, by declaring C global variables and arrays. The linker and the boot loader cooperate to decide exactly what memory locations will hold these variables, so that the C code doesn't have to explicitly allocate memory. However, xv6 does explicitly and dynamically allocate physical memory for user process memory, for the kernel stacks of user processes, and for pipe buffers. When xv6 needs memory for one of these purposes, it calls `kalloc`; when it no longer needs them memory, it calls `kfree` to release the memory back to the allocator. Xv6's memory allocator manages blocks of memory that are a multiple of 4096 bytes, because the allocator is used mainly to allocate process address spaces, and the x86 segmentation hardware manages those address spaces in multiples of 4 kilobytes. The xv6 allocator calls one of these 4096-byte units a page, though it has nothing to do with paging.

`Main` calls `kinit` to initialize the allocator (1226). `Kinit` ought to begin by determining how much physical memory is available, but this turns out to be difficult on the x86. Xv6 doesn't need much memory, so it assumes that there is at least one megabyte available past the end of the loaded kernel and uses that megabyte. The kernel is around 50 kilobytes and is loaded one megabyte into the address space, so xv6 is assuming that the machine has at least a little more than two megabytes of memory, a very safe assumption on modern hardware.

`Kinit` (2277) uses the special linker-defined symbol `end` to find the end of the kernel's static data and rounds that address up to a multiple of 4096 bytes (2284). When `n` is a power of two, the expression `(a+n-1) & ~(n-1)` is a common C idiom to round `a` up to the next multiple of `n`. `Kinit` then does a surprising thing: it calls `kfree` to free a megabyte of memory starting at that address (2287). The discussion of kalloc and kfree above said that kfree was for returning memory allocated with kalloc, but that was a client-centric perspective. From the allocator's point of view, calls to kfree give it memory to hand out, and then calls to kalloc ask for the memory back. The allocator starts with no memory; this initial call to kfree gives it a megabyte to manage.

The allocator maintains a *free list* of memory regions that are available for allocation. It keeps the list sorted in increasing order of address in order to ease the task of merging contiguous blocks of freed memory. Each contiguous region of available memory is represented by a struct run. But where does the allocator get the memory to hold that data structure? The allocator does another surprising thing: it uses the memory being tracked as the place to store the run structure tracking it. Each run *r represents the memory from address (uint)r to (uint)r + r->len. The free list is protected by a spin lock (2262-2265). 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.

Kfree (2305) begins by setting every byte in the memory being freed to the value 1. This step is unnecessary for correct operation, but it helps break incorrect code that continues to refer to memory after freeing it. This kind of bug is called a dangling reference. By setting the memory to a bad value, kfree increases the chance of making such code use an integer or pointer that is out of range (0x01010101 is around 16 million).

Kfree's first real work is to store a run in the memory at v. It uses a cast in order to make p, which is a pointer to a run, refer to the same memory as v. It also sets pend to the run for the block following v (2316-2317). If that block is free, pend will appear in the free list. Now kfree walks the free list, considering each run r. The list is sorted in increasing address order, so the new run p belongs before the first run r in the list such that r> pend. The walk stops when either such an r is found or the list ends, and then kfree inserts p in the list before r (2337-2340). The odd-looking for loop is explained by the assignment *rp = p: in order to be able to insert p *before* r, the code had to keep track of where it found the pointer r, so that it could replace that pointer with p. The value rp points at where r came from.

There are two other cases besides simply adding p to the list. If the new run p abuts an existing run, those runs need to be coalesced into one large run, so that allocating and freeing small blocks now does not preclude allocating large blocks later. The body of the for loop checks for these conditions. First, if rend == p (kalloc.c/rend.==.p/), then the run r ends where the new run p begins. In this case, p can be absorbed into r by increasing r's length. If growing r makes it abut the next block in the list, that block can be absorbed too (kalloc.c/r->next && r->next == pend/,/}/). Second, if pend == r (kalloc.c/pend.==.r/), then the run p ends where the new run r begins. In this case, r can be absorbed into p by increasing p's length and then replacing r in the list with p (2330-2335).

Kalloc has a simpler job than kfree: it walks the free list looking for a run that is large enough to accommodate the allocation. When it finds one, kalloc takes the memory from the end of the run (2364-2365). If the run has no memory left, kalloc deletes the run from the list (2367-2368) before returning.

Code: Process creation

This section describes how xv6 creates the very first process. Xv6 represents each process by a `struct proc` (1529) entry in the statically-sized `ptable.proc` process table. The most important fields of a `struct proc` are `mem`, which points to the physical memory containing the process's instructions, data, and stack; `kstack`, which points to the process's kernel stack for use in interrupts and system calls; and and `state`, which indicates whether the process is allocated, ready to run, running, etc.

The story of the creation of the first process starts when `main` (1235) calls `userinit` (1802), whose first action is to call `allocproc`. The job of `allocproc` (1754) is to allocate a slot in the process table and to initialize the parts of the process's state required for it to execute in the kernel. `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` (1669-1762). When it finds an unused process, `allocproc` sets the state to `EMBRYO` to mark it as used and gives the processes a unique `pid` (1658-1768). Next, it tries to allocate a kernel stack for the process. 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. As we will see in Chapter 3, the usual way that a process enters the kernel is via an interrupt mechanism, which is used by system calls, interrupts, and exceptions. The process's kernel stack is the one it uses when executing in the kernel during the handling of that interrupt. `Allocproc` 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, so that the ordinary code for returning from the kernel back to the user part of a process will work. These values are a `struct trapframe` which stores the user registers, the address of the kernel code that returns from an interrupt (`trapret`) for use as a function call return address, and a `struct context` which holds the process's kernel registers. When the kernel switches contexts to this new process, the context switch will restore its kernel registers; it will then execute kernel code to return from an interrupt and thus restore the user registers, and then execute user instructions. `Allocproc` sets `p->context->eip` to `forkret`, so that the process will start executing in the kernel at the start of `forkret`. The context switching code will start executing the new process with the stack pointer set to `p->context+1`, which points to the stack slot holding the address of the `trapret` function, just as if `forkret` had been called by `trapret.`

```
 ----------  <-- 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
```

Main calls userinit to create the first user process (1235). Userinit (1802) calls allocproc, saves a pointer to the process as initproc, ad then configures the new process's user state. First, the process needs memory. This first process is going to execute a very tiny program (initcode.S; (6700)), so the memory need only be a single page (1811-1812). The initial contents of that 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 and zeros the rest (1813-1814). Then it 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

Rather than use special code to start the first process running and guide it to user space, xv6 has chosen to set up the initial data structure state as if that process was already running. But it wasn't running and still isn't: so far, this has been just an elaborate construction exercise, like lining up dominoes. Now it is time to knock over the first domino, set the operating system and the hardware in motion and watch what happens.

Main calls ksegment to initialize the kernel's segment descriptor table (1219). Ksegment initializes a per-CPU global descriptor table c->gdt with the same segments that the boot sector configured (and one more, SEG_KCPU, which we will revisit in Chapter 4). After calling userinit, which we examined above, main calls scheduler to start running user processes (1263). Scheduler (1908) looks for a process with p->state set to RUNNABLE, and there's only one it can find: initproc. It sets the global variable cp to the process it found (cp stands for current process) and calls usegment to create segments on this CPU for the user-space execution of the process (1846). Usegment (1722) creates code and data segments SEG_UCODE and SEG_UDATA mapping addresses 0 through cp->sz-1 to the memory at cp->mem. It 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)cp->kstack+KSTACKSIZE, the top of this process's kernel stack. We will reexamine the task state segment in Chapter 3.

Now that usegment has created the user code and data segments, the scheduler can start running the process. It sets p->state to RUNNING and calls swtch (2208), to perform a context switch from one kernel process to another; in this invocation, from a scheduler process to p. Swtch, which we will reexamine in Chapter 5, saves the scheduler's registers that must be saved; i.e., the context (1518) that a process needs to later resume correctly. Then, Swtch loads p->context into the hardware registers. The final ret instruction (2227) pops a new eip from the stack, finishing the context switch. Now the processor is running process p.

Allocproc set initproc's p->context->eip to forkret, so the ret starts executing forkret. Forkret (1984) 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 (2529) 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 %cs:%eip specified in the trap frame. For initproc, that means SEG_UCODE:0, the first instruction of initcode.S.

At this point, %eip holds zero and %esp holds 4096. These are virtual addresses in the process's user address space. The processor's segmentation machinery translates them into physical addresses. The relevant segmentation registers (cs, ds, and ss) and segment descriptors were set up by userinit and usegment to translate virtual address zero to physical address p->mem, with a maximum virtual address of p->sz. The fact that the process is running with CPL=3 (in the low bits of cs) means that it cannot use the segment descriptors SEG_KCODE and SEG_KDATA, which would give it access to all of physical memory. So the process is constrained to using only its own memory.

Initcode.S (6707) 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 (6720-6722). If the exec fails and does return, initcode loops calling the exit system call, which definitely should not return (6714-6718).

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

Xv6 departs from modern operating systems in its use of segmentation registers for process isolation and address translation. Most operating systems for the x86 uses the paging hardware for address translation and protection; they treat the segmentation hardware mostly as a nuisance to be disabled by creating no-op segments like the boot sector did. However, a simple paging scheme is somewhat more complex to implement than a simple segmentation scheme. Since xv6 does not aspire to any of the advanced features which would require paging, it uses segmentation instead.

The one common use of segmentation is to implement variables like xv6's `cp` that are at a fixed address but have different values in different threads. Implementations of per-CPU (or per-thread) storage on other 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 use of segmentation instead of paging is awkward in a couple of ways, even given its low ambitions. First, it causes user-space address zero to be a valid address, so that programs do not fault when they dereference null pointers; a paging system could force faults by marking the first page invalid, which turns out to be invaluable for catching bugs in C code. Second, xv6's segment scheme places the stack at a relatively low address which prevents automatic stack extension. Finally, all of a process's memory must be contiguous in physical memory, leading to fragmentation and/or copying.

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 one megabyte of memory past the end of the kernel. Since the kernel is around 50 kilobytes and is loaded one megabyte into the address space, xv6 is assuming that the machine has at least a little more than 2 MB of memory. A real operating system would have to do a better job.

Memory allocation was a hot topic a long time ago. Basic problem was how to make the most efficient use of the available memory and how best to prepare for future requests without knowing what the future requests were going to be. See Knuth. Today, more effort is spent on making memory allocators fast rather than on making them space-efficient. The runtimes of today's modern programming languages allocate mostly many small blocks. Xv6 avoids smaller than a page allocations by using fixed-size data structures. A real kernel allocator would need to handle small allocations as well as large ones, although the paging hardware might keep it from needing to handle objects larger than a page.

Exercises

1. Set a breakpoint at swtch. Single step through to forkret. Set another breakpoint at forkret's ret. Continue past the release. Single step into trapret and then all the way to the iret. Set a breakpoint at 0x1b:0 and continue. Sure enough you end up at initcode.

2. Do the same thing except single step past the iret. You don't end up at 0x1b:0. What happened? Explain it. Peek ahead to the next chapter if necessary.

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