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)
```