COP4610: Operating Systems

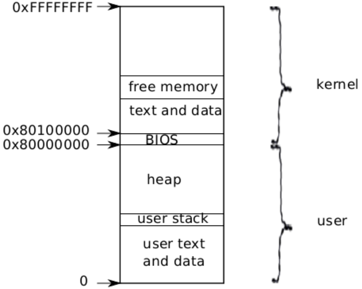
Xv6 Processes

Zhi Wang

Florida State University

Spring 2015

Process Address Space



- Each process has a separate page table that defines its address space
- The (same) kernel is mapped in all the processes
  ➠ the kernel can safely switch user page tables without disruption
- The process can run either in the kernel (syscall) or in the user-space

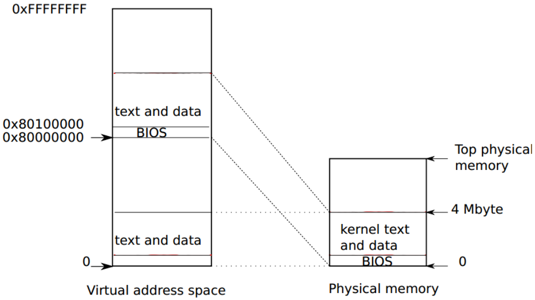
Process Control Block (proc.h)

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

Process Control Block

- pgdir: the process's page table
  - data structure used by x86 to map virtual address to physical ones
- kstack: the bottom of the kernel stack for this process
  - each process has a user stack and a kernel stack
  - kernel stack is empty when the process is running in the user space
- tf: saved user-space state when entering the kernel (e.g., via syscall)
- context: saved kernel state for context switch (swtch)

Initial Kernel Address Space



- Boot loader loads the kernel (xv6.img) at physical address 0x100000
- The kernel starts execution at entry (entry.S)
  - entry loads initial page table and jumps to high addresses (main)
  - main runs in the proper kernel address space (main.c)
- There is no process yet...

The First Process

- First process must be manually crafted
  ➠ no process to call fork and exec
- main calls userinit to create the first process, which becomes init
  ➠ init is the first user process (init.c)
- userinit allocates a PCB, and initializes the kernel stack as if the process has just made a fork syscall (proc.c)
  ➠ it returns to the user space just like a forked child process

userinit

1. call allocproc to allocate a new PCB

userinit

1. call allocproc to allocate a new PCB
2. call setupkvm to set up the kernel address space
   ➠ the kernel is mapped in every process' address space, remember?

userinit

1. call allocproc to allocate a new PCB
2. call setupkvm to set up the kernel address space
   - the kernel is mapped in every process' address space, remember?
3. call inituvm to copy initcode (initcode.S) to user-space
   - initcode simply calls exec("init", 0) (init.c)

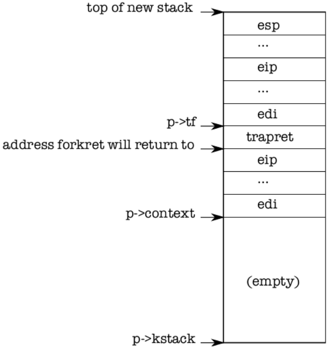
userinit

1. call allocproc to allocate a new PCB
2. call setupkvm to set up the kernel address space
   - ➠ the kernel is mapped in every process' address space, remember?
3. call inituvm to copy initcode (initcode.S) to user-space
   - ➠ initcode simply calls exec("init", 0) (init.c)
4. set up trapframe to "return" to initcode
   - ➠ `p->tf->eip = 0; // beginning of initcode.S`
   - ➠ the kernel returns to `p->tf->eip` after syscall, remember?

allocproc

1. loop through ptable for a free PCB
2. allocate the kernel stack
3. set up the kernel stack for the new process
   - the “only” way to create a new process is through fork
   - the stack is set up for the forked child to return to user space
   - now, a quick trip to fork (proc.h)

allocproc



- context switch (swtch) pops p->context off the kernel stack, and returns to trapret
- trapret restores user registers (p->tf)and returns to p->tf->eip
  - p->tf->eip points to initcode