XV6 PROCESS AND CPU SCHEDULING

李大為、张雷、陈传文、邓凌鹏

TABLE OF CONTENTS

PROCESS

BASIC DATA STRUCTURE

PROCESS
PCB, STATES, MEMORY

- Most Important Pieces of Kernel State
  - pgdir
    (page table)
  - kstack
    (kernel stack)
  - state
    (run state)

```c
/* proc.h */

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// 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 memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap
```

PROCESS LIST (PTABLE) AND CONTEXT

- ptable is a global variable in proc.c
  in param.h: #define NPROC 64 // max processes

```c
/* proc.c */

struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

```c
/* proc.h */

// Saved registers for kernel context switches.
// Don't need to save all the segment registers (%cs, etc),
// because they are constant across kernel contexts.
// Don't need to save %eax, %ecx, %edx, because the
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom of the stack they
// describe; the stack pointer is the address of the context.
// The layout of the context matches the layout of the stack in swtch.S
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,
// but it is on the stack and allocproc() manipulates it.
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

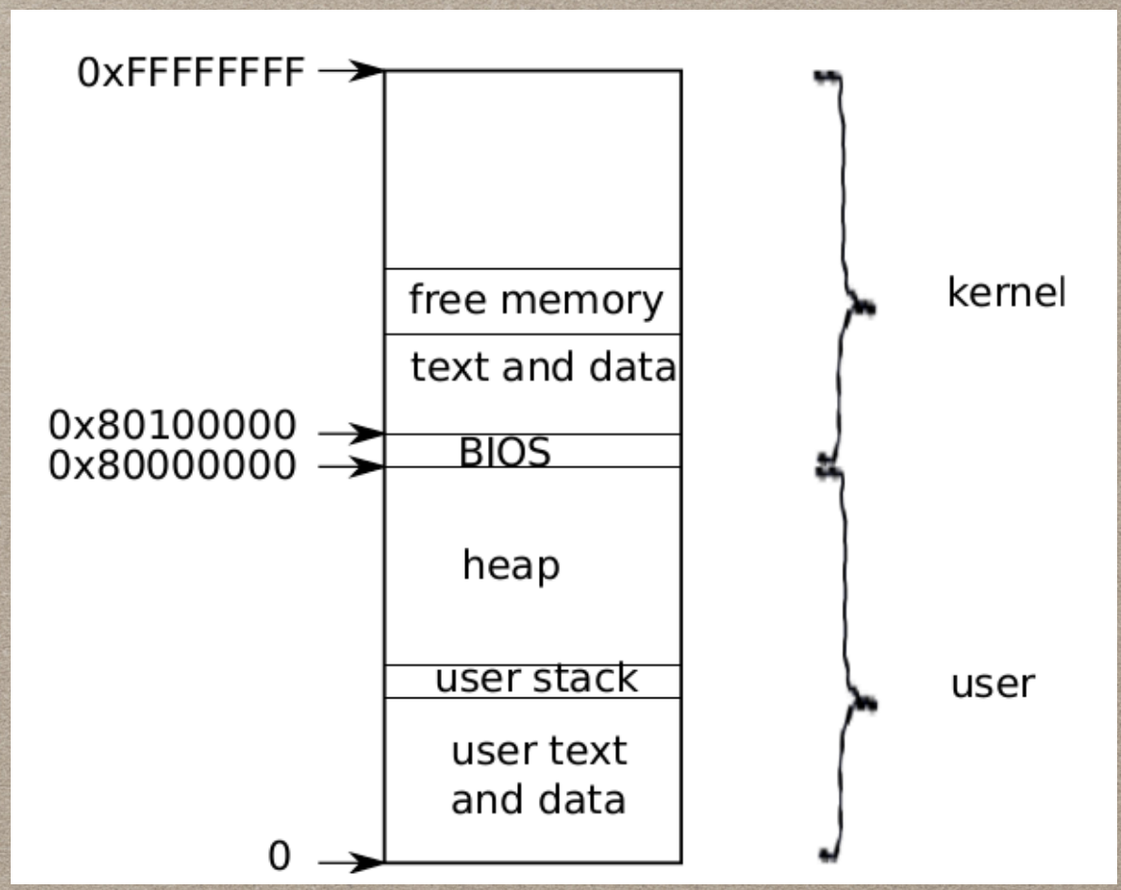
TRAPFRAME

- Stores the user register
  - %esp: stack pointer
  - %cs: segment sector
  - %eflags: hardware interrupt

```
/* x86.h */

// Layout of the trap frame built on the stack by the
// hardware and by trapasm.S, and passed to trap().
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to
kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```

LAYOUT OF XV6'S VIRTUAL ADDRESS SPACE

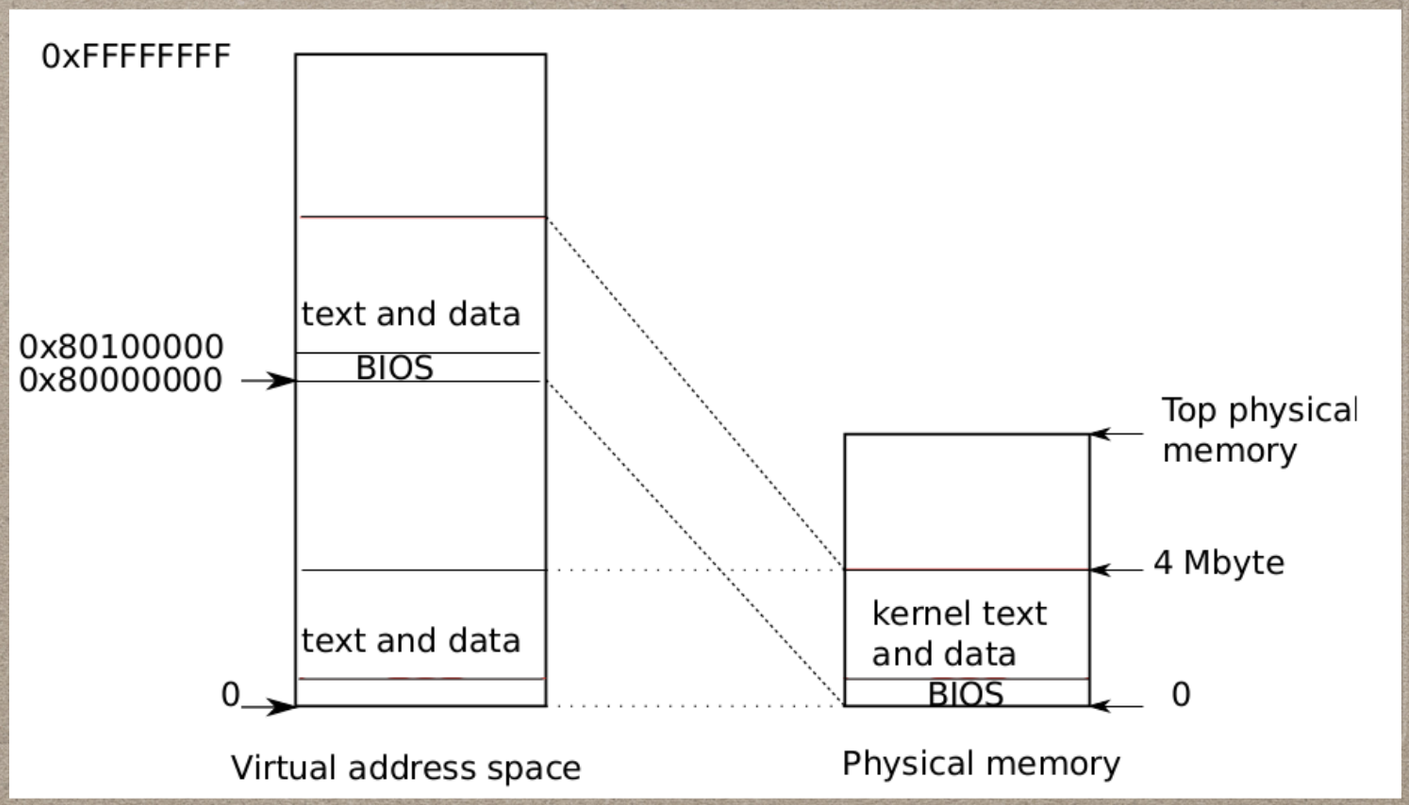
- User Stack (0x00000000)
  - instructions
  - global variables
  - stack
  - heap



KERNEL SPACE MAPPING (STARTS FROM 0X80100000)

- Each process maps the kernel's address space into the user program's memory
  =>System call executes in the kernel mappings of the process's address space

Thus, the kernel's syscall can directly refer to user memory



TWO STACKS: KERNEL, USER

- Stacks: User stack and Kernel stack (p->kstack)
- User Space
  - user stack: in use
  - kernel stack: empty
- Kernel Space (by system call or interrupt)
  - user stack: still contains saved data
  - kernel stack: kernel code executes on it

CREATING PROCESS AND RUN

CREATING THE FIRST PROCESS (TRACE CODE)

1. main(): create first process by userinit()
2. userinit(): for the first process
   (p->tf->eip = 0; // beginning of initcode.S)
3. allocproc(): scan and allocate a unused slot (struct proc) in the ptable and initialize process's state
   - for each new process, and also used by fork()
4. Once the process is initialized, userinit() marks it available for scheduling by setting p->state to RUNNABLE

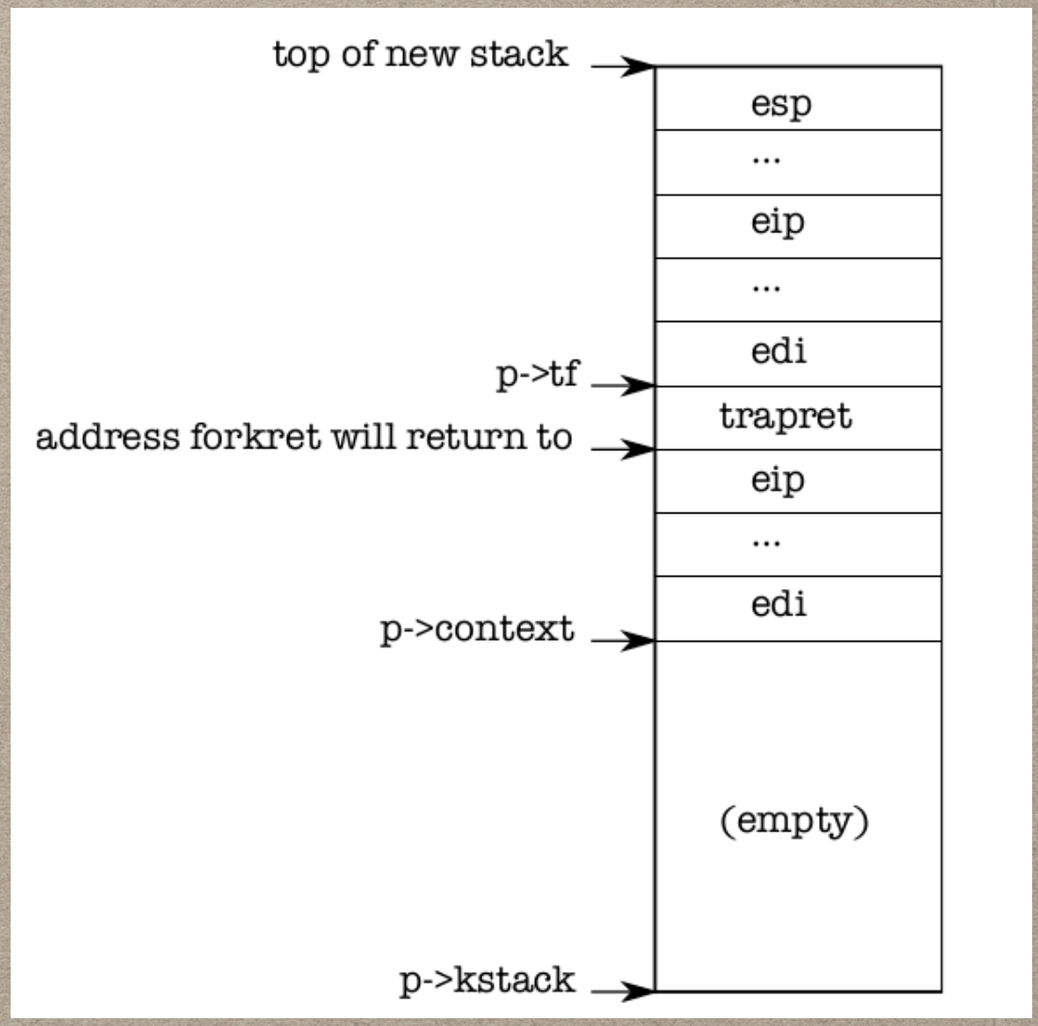
ALLOCPROC() IN DETAIL

- Set state to EMBRYO to mark it as used and give it a unique pid
- Set state back to UNUSED and return 0 when fail
- Set up the new process's kernel stack
- Set up the return function pointer (p->context->eip)

```
/* proc.c */

// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  release(&ptable.lock);

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;

  return p;
}
```

A NEW KERNEL STACK

- allocproc() sets up the new process with a “prepared” kernel stack
- also set of kernel registers that cause it to “return” to user space when it first run (the forkret and trapret)



RUNNING THE FIRST PROCESS (TRACE CODE)

1. mpmain(): calls scheduler() to start running processes

2. scheduler(): looks for a process with p->state is RUNNABLE

3. initcode.S: the first process program to run (in user space)

SCHEDULER

- set the per-cpu variable to the process it found
- switchuvm()
- swtch() switch to process => now the processor is running on the p's kernel stack

```
/* proc.c */

// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);

  }
}
```

SWITCHUVM() IN DETAIL

```
/* vm.c */

// Switch TSS and h/w page table to correspond to process p.
void
switchuvm(struct proc *p)
{
  if(p == 0)
    panic("switchuvm: no process");
  if(p->kstack == 0)
    panic("switchuvm: no kstack");
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");

  pushcli();
  mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
                                sizeof(mycpu()->ts)-1, 0);
  mycpu()->gdt[SEG_TSS].s = 0;
  mycpu()->ts.ss0 = SEG_KDATA << 3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
  // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
  // forbids I/O instructions (e.g., inb and outb) from user space
  mycpu()->ts.iomb = (ushort) 0xFFFF;
  ltr(SEG_TSS << 3);
  lcr3(V2P(p->pgdir));  // switch to process's address space
  popcli();
}
```

- Tell the hardware
  to start using the target process's page table

- Task State Segment (SEG_TSS): instruct the hardware to execute system calls and interrupts on the process's kernel stack

CONTEXT SWITCH SWTCH()

- Save the current hardware register in per-cpu storage (cpu->scheduler)
- Loads the saved registers of the target kernel thread (p->context)
- ret: the function pointer to call when “return”

2. nvim swtch.S (nvim)

swtch.S

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
~
<WD: /Users/daviddwlee84/Documents/Program/Github/XV6/xv6-public   Line: 1  Column: 1
```

FORKRET()
TRAPRET()

```c
/* proc.c */

// A fork child's very first scheduling by scheduler()
// will swtch here.  "Return" to user space.
void
forkret(void)
{
  static int first = 1;
  // Still holding ptable.lock from scheduler.
  release(&ptable.lock);

  if (first) {
    // Some initialization functions must be run in the
context
    // of a regular process (e.g., they call sleep), and thus
cannot
    // be run from main().
    first = 0;
    iinit(ROOTDEV);
    initlog(ROOTDEV);
  }

  // Return to "caller", actually trapret (see allocproc).
}
```

2. nvim trapasm.S (nvim)

```
trapasm.S
#include "mmu.h"

  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
<WD: /Users/daviddwlee84/Documents/Program/Github/XV6/xv6-public   Line: 2  Column: 0
```

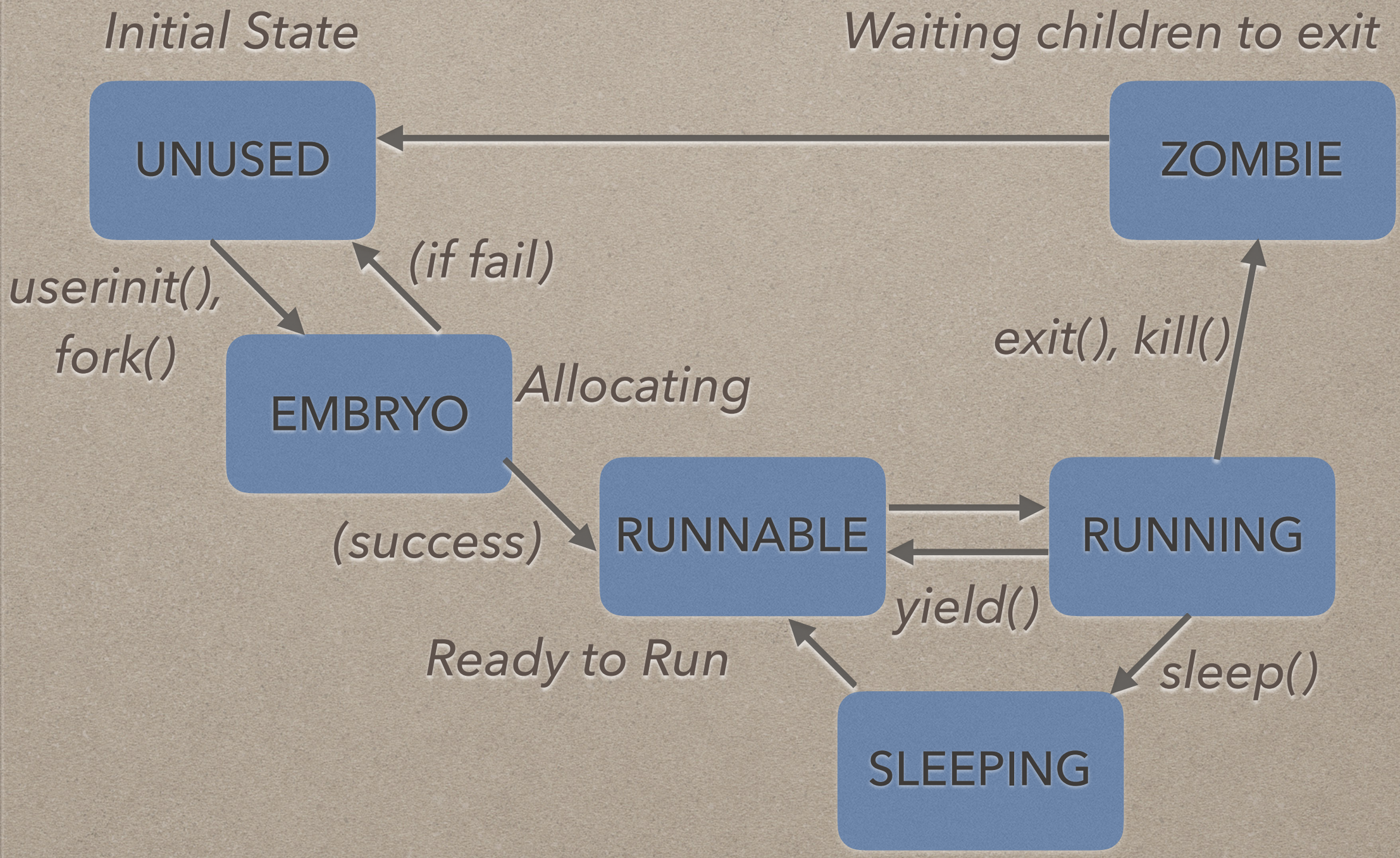
PROCESS STATE SWITCHING

STATES OF PROCESS

p->state

- UNUSED: initial state
- EMBRYO: allocated / used
- RUNNABLE: ready to run
- RUNNING: running
- SLEEPING: blocking e.g. waiting for I/O
- ZOMBIE: exiting

STATUS MODEL



PROCESS OPERATIONS

- fork(): Create a new process copying p as the parent.
  - Sets up stack to return as if from system call.
  - Caller must set state of returned proc to RUNNABLE.
- exit(): Exit the current process. Does not return.
  - An exited process remains in the zombie state until its parent calls wait() to find out it exited.
- wait(): Wait for a child process to exit and return its pid.
  - Return -1 if this process has no children.
- yield(): Give up the CPU for one scheduling round.
- sleep(): Atomically release lock and sleep on chan.
  - Reacquires lock when awakened.

GIVE UP CPU

- A process that wants to give up the CPU:
    1. Acquire the process table lock ptable.lock
    2. Release any other locks it is holding
    3. update its own state (proc->state)
    4. call sched()
- yield(), sleep(), exit() follows this convention

SCHED()

- Switch from process to scheduler

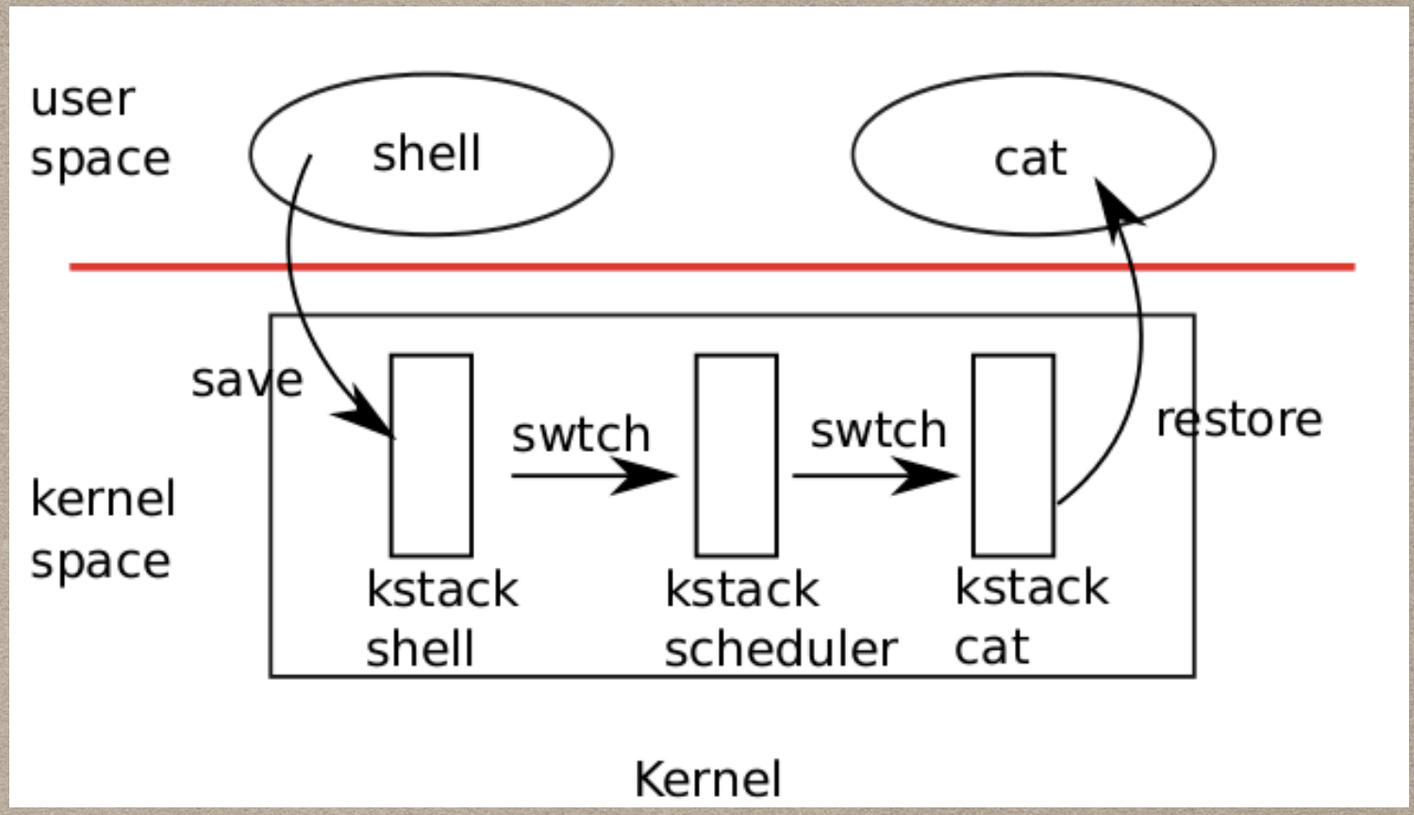
```
/* proc.c */

// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state. Saves and restores
// intena because intena is a property of this
// kernel thread, not this CPU. It should
// be proc->intena and proc->ncli, but that would
// break in the few places where a lock is held but
// there's no process.
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

CPU SCHEDULING

MULTIPLEXING XV6'S SLEEP AND WAKEUP

- "Switch" when a process...
  - waits for device or pipe I/O to complete
  - waits for child to exit
  - waits in the sleep() system call
- XV6 periodically forces a switch when a process is executing user instructions
  => illusion that each process has its own CPU

SWITCH BETWEEN USER PROCESSES



1. a user-kernel transition (system call or interrupt) to the old process's kernel thread
2. a context switch to the local CPU's scheduler thread
3. a context switch to a new process's kernel thread
4. a trap return to the user-level process

THE STRUCTURE OF SCHEDULING

- It arrange to enforce a set of invariants
  - if a process is RUNNING, a timer interrupt's yield must be able to switch away from the process
  - if a process is RUNNABLE, an idle CPU's scheduler must be able to run it
- And holds ptable.lock whenever those invariants are not true
  - that's why XV6 acquires lock in one thread (often in yield())
  - and releases the lock in a different thread (the scheduler thread or another next kernel thread)

MORE ABOUT PTABLE.LOCK

- To protect
  - allocation of process IDs
  - free process table slots
  - interplay between exit() and wait()
  - machinery to avoid lost wakeups
  - other things...

CONCLUSION OF SCHEDULING

- Maintaining the invariants by ptable.lock
- Round-Robin-like scheduling (without priority)
  - periodically switch user process (yield() by timer)
  - timer interrupt is controlled / triggered by the hardware