Lecture 7: Processes-Scheduling

Professor G. Sandoval

Some slides adapted by G. Sandoval for CS3224, from Tanenbaum & Bo, Modern Operating Systems:4th ed.,
Also some Slides by Brendan Dolan-Gavitt and Stallings: Operatings Systems. Eight Edition.

Today

- Processes
- Process States
- Unix Processes
- XV6 Processes
- Scheduling
- XV6 Scheduling

Processes

- A *process* is the basic unit of execution and an abstraction for a running program.
- Splitting up the execution of a system into processes allows:
  - Isolation between programs
  - The ability to simulate running multiple programs at once with only one CPU

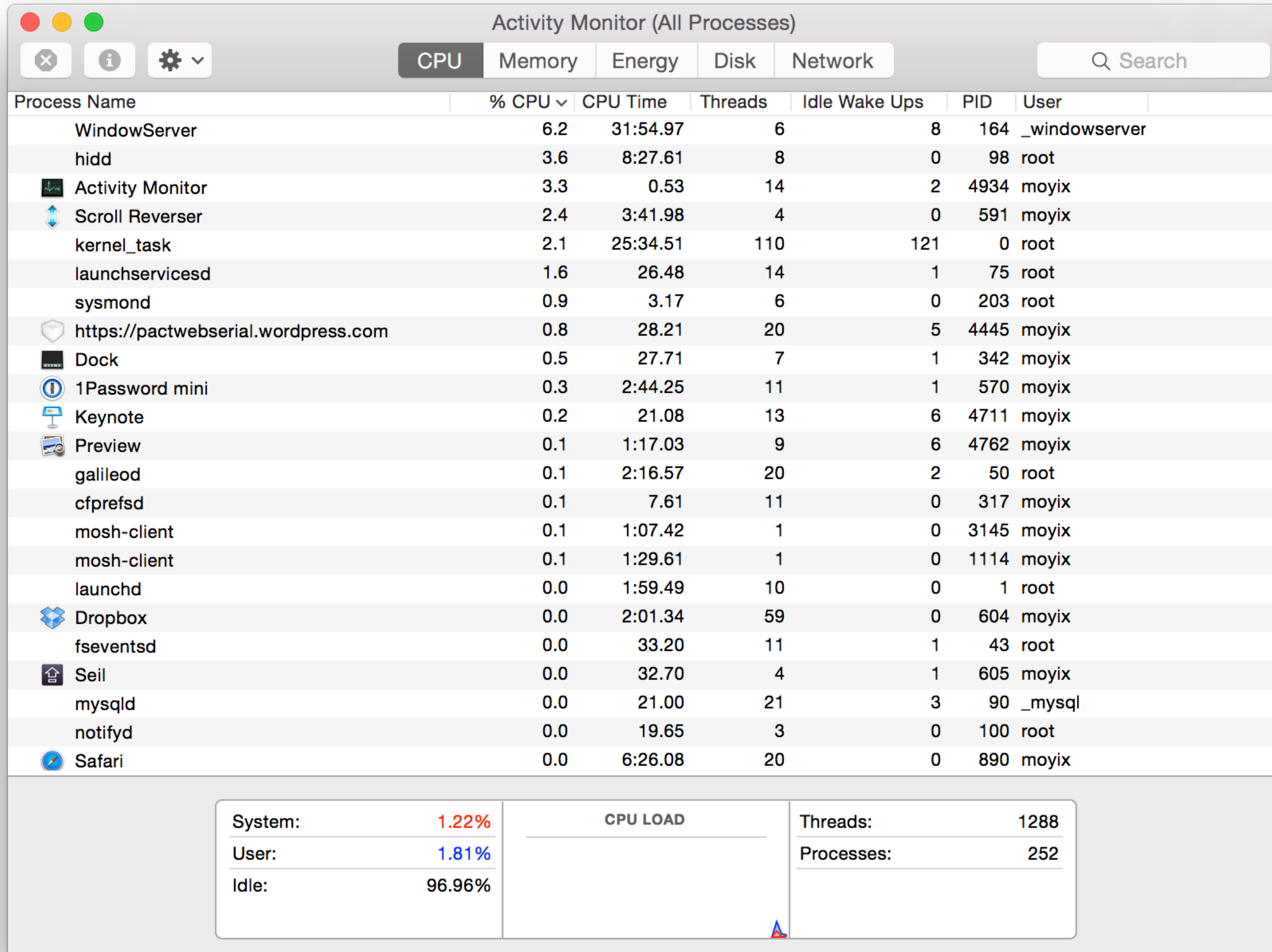
Processes in Parallel

- It is confusing to reason about many events happening at once
- The process model lets us consider each process as a single, *sequential* execution
- (For now, we'll consider just one CPU)

Processes Are Not Programs

- A *program* = instructions + data
- A *process* is the OS abstraction for a running program, its resources, input/output, etc.
- Process = memory (instructions + data + stack)
- Think of a process as an instance of a program, just like an Object is an instance of a class.
- So: multiple copies of a program can be running in different processes

Lots of Processes



Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
WindowServer	6.2	31:54.97	6	8	164	_windowserver
hidd	3.6	8:27.61	8	0	98	root
Activity Monitor	3.3	0.53	14	2	4934	moyix
Scroll Reverser	2.4	3:41.98	4	0	591	moyix
kernel_task	2.1	25:34.51	110	121	0	root
launchservicesd	1.6	26.48	14	1	75	root
sysmond	0.9	3.17	6	0	203	root
https://pactwebserial.wordpress.com	0.8	28.21	20	5	4445	moyix
Dock	0.5	27.71	7	1	342	moyix
1Password mini	0.3	2:44.25	11	1	570	moyix
Keynote	0.2	21.08	13	6	4711	moyix
Preview	0.1	1:17.03	9	6	4762	moyix
galileod	0.1	2:16.57	20	2	50	root
cfprefsd	0.1	7.61	11	0	317	moyix
mosh-client	0.1	1:07.42	1	0	3145	moyix
mosh-client	0.1	1:29.61	1	0	1114	moyix
launchd	0.0	1:59.49	10	0	1	root
Dropbox	0.0	2:01.34	59	0	604	moyix
fseventsd	0.0	33.20	11	1	43	root
Seil	0.0	32.70	4	1	605	moyix
mysqld	0.0	21.00	21	3	90	_mysql
notifyd	0.0	19.65	3	0	100	root
Safari	0.0	6:26.08	20	0	890	moyix

System:	1.22%	Threads:	1288
User:	1.81%	Processes:	252
Idle:	96.96%		

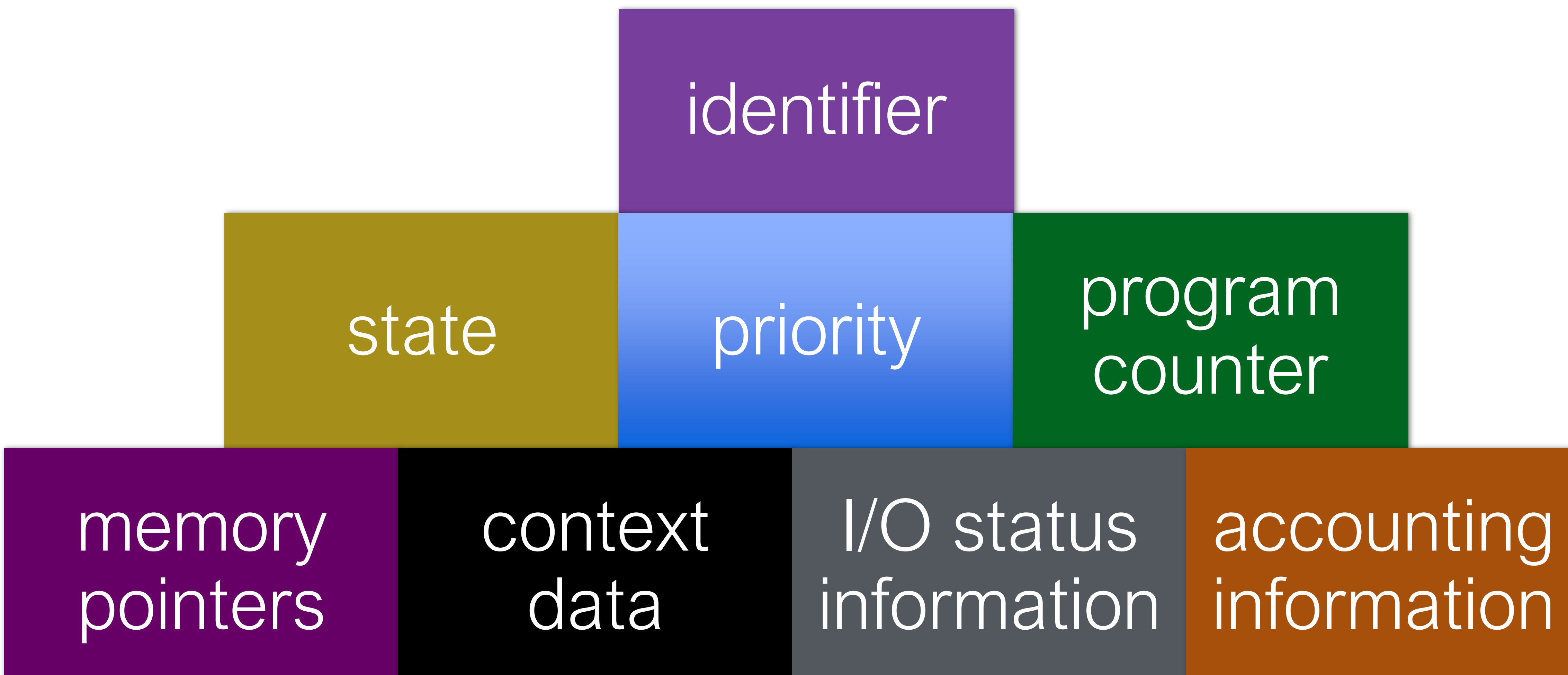
CPU LOAD

Bash on Ubuntu on Windows

```
cs3224> ps
  PID TTY          TIME CMD
 3940 tty4     00:00:02 bash
 6879 tty4     00:00:00 shell_soln
 6882 tty4     00:00:00 ps
cs3224>
```

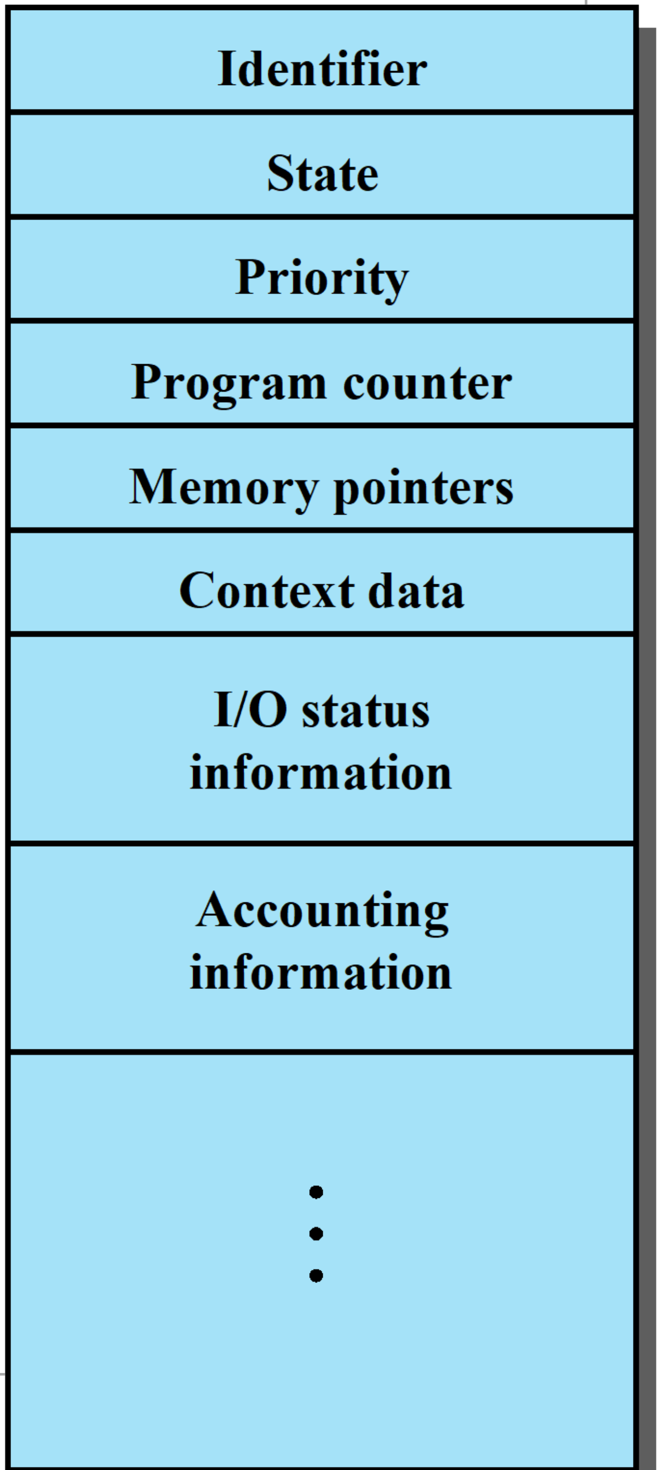
Process Elements

- While the program is executing, this process can be uniquely characterized by a number of elements, including:



Process Control Block

- Data Structure that contains the process elements
- Allows to store state and interrupt a running process and later resume execution as if the interruption had not occurred
- Created and managed by the operating system



Today

- Processes
- Process States
- Unix Processes
- XV6 Processes
- Scheduling
- XV6 Scheduling

Process Creation

- Processes can be created in many circumstances
  - At boot time
  - User request (e.g., typing a command)
  - A running program creates a new process on its own
- All of these are actually carried out by execution of a system call (with the exception of the *first* process, which is created directly by the kernel)

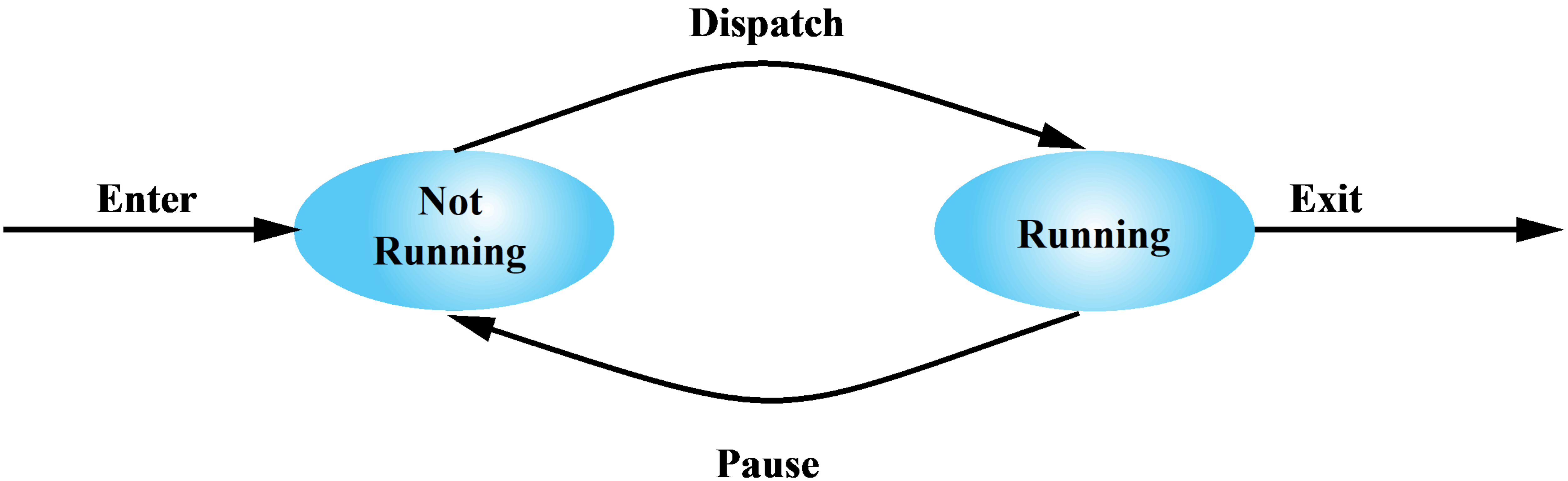
Process Termination

- There must be a means for a process to indicate its completion

- A batch job should include a HALT instruction or an explicit OS service call for termination

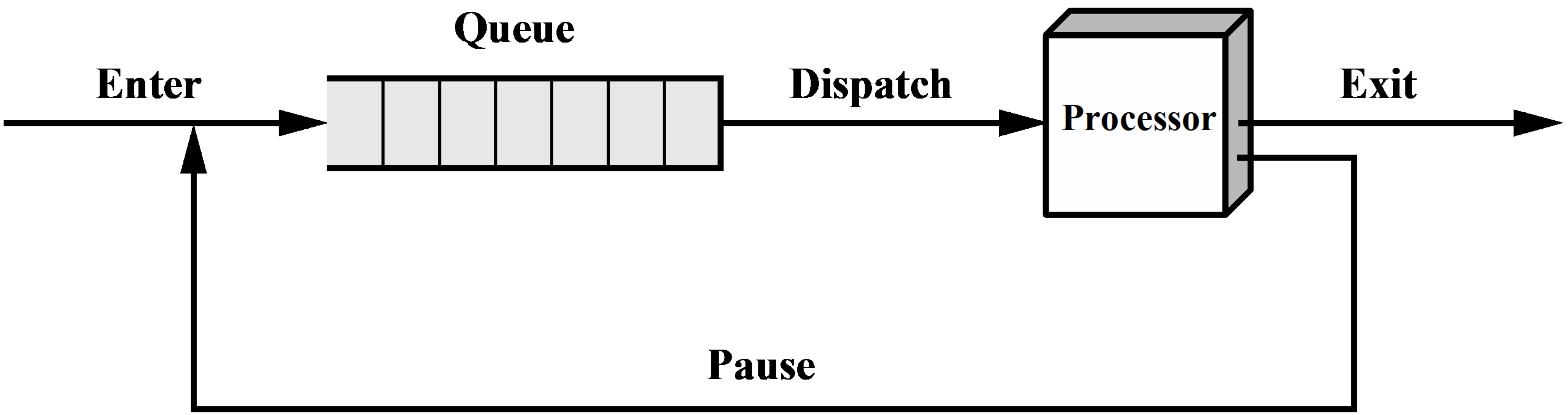
- For an interactive application, the action of the user will indicate when the process is completed (e.g. log off, quitting an application)

Two-State Process Model



(a) State transition diagram

Two-State Process Model

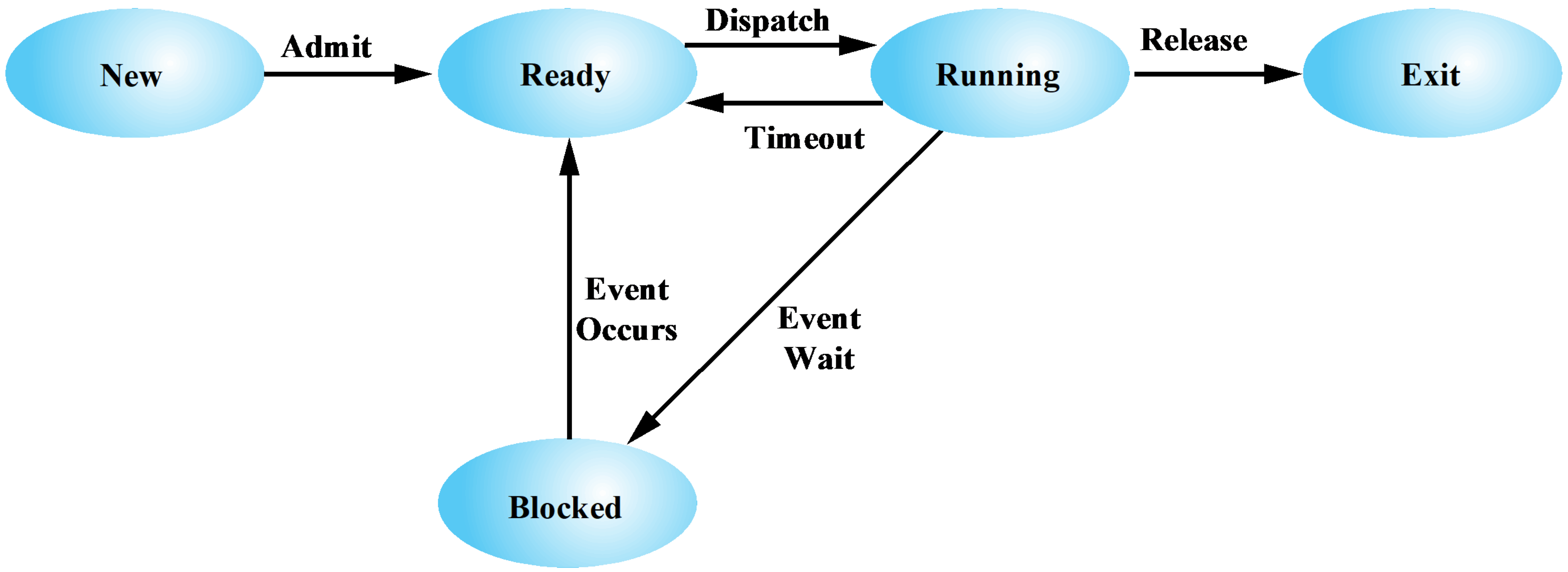


(b) Queuing diagram

Reasons for Process Termination

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

Five-State Process Model



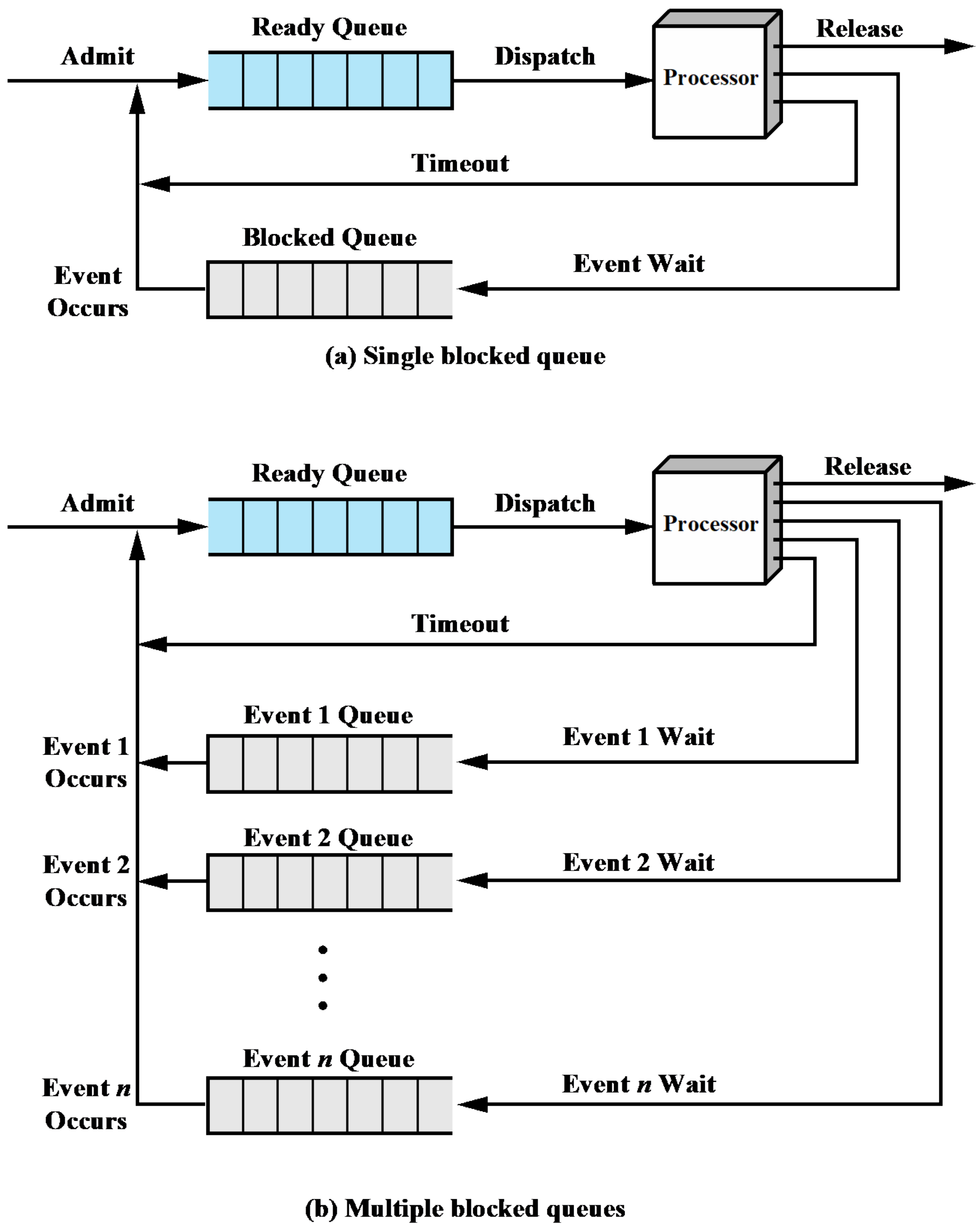
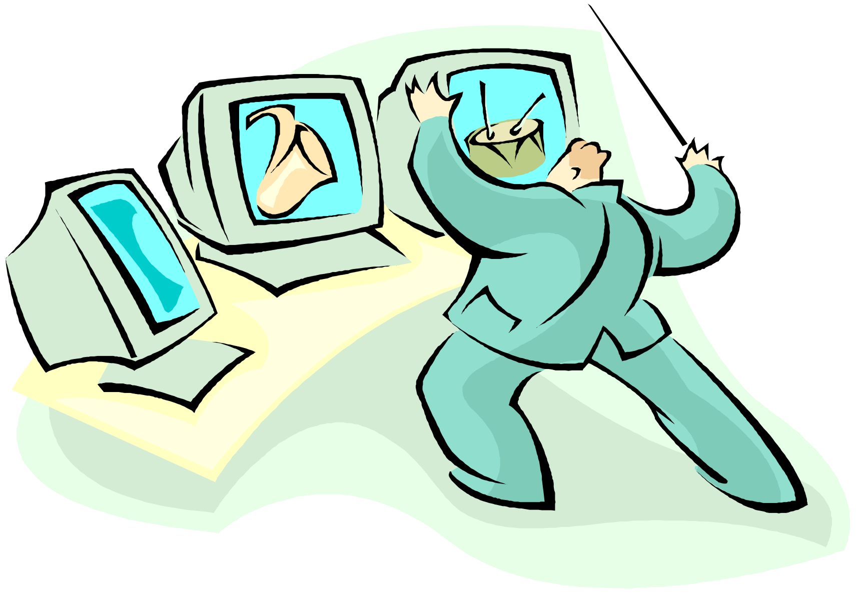
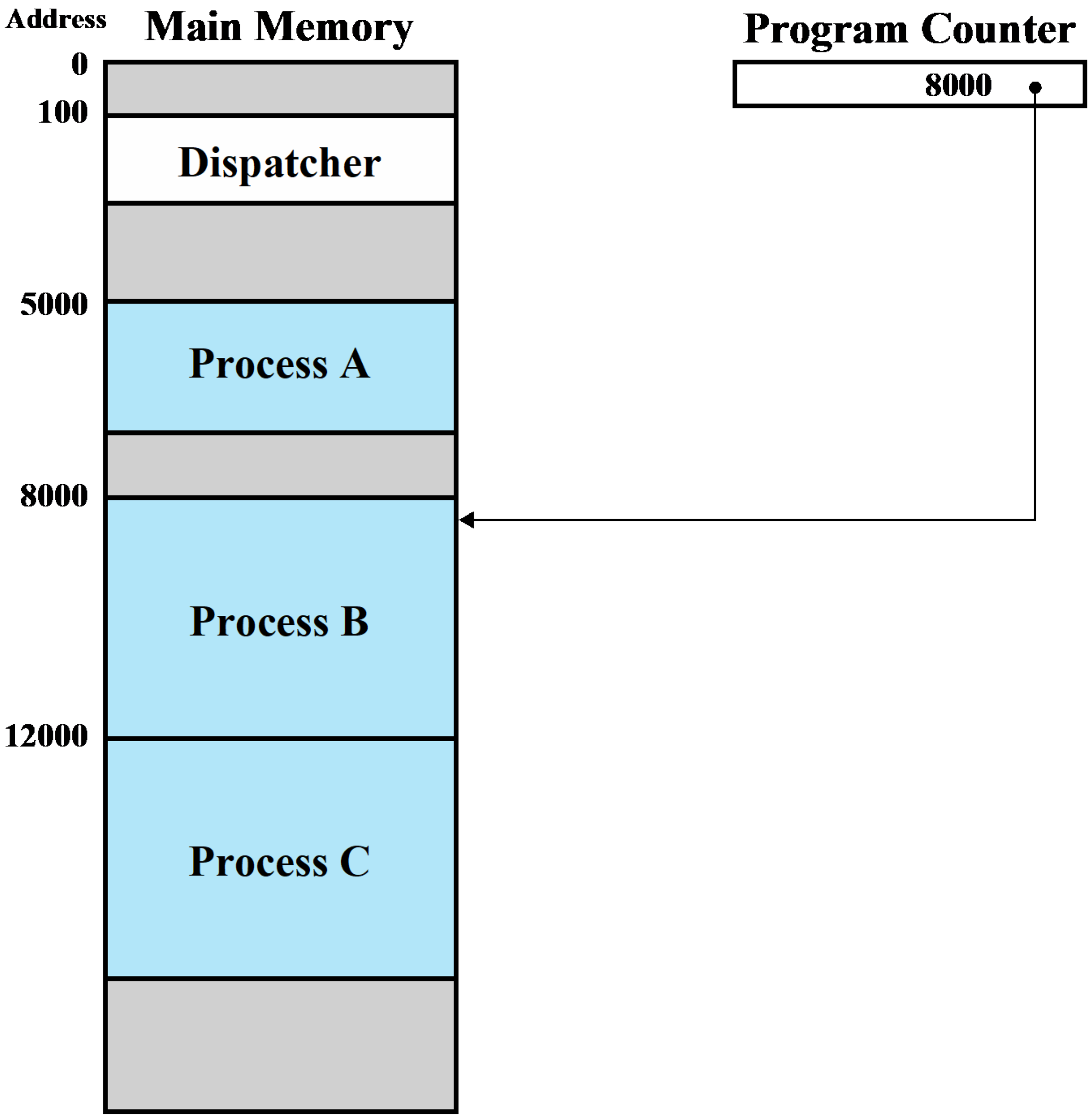


Figure 3.8 Queuing Model for Figure 3.6

Process Execution





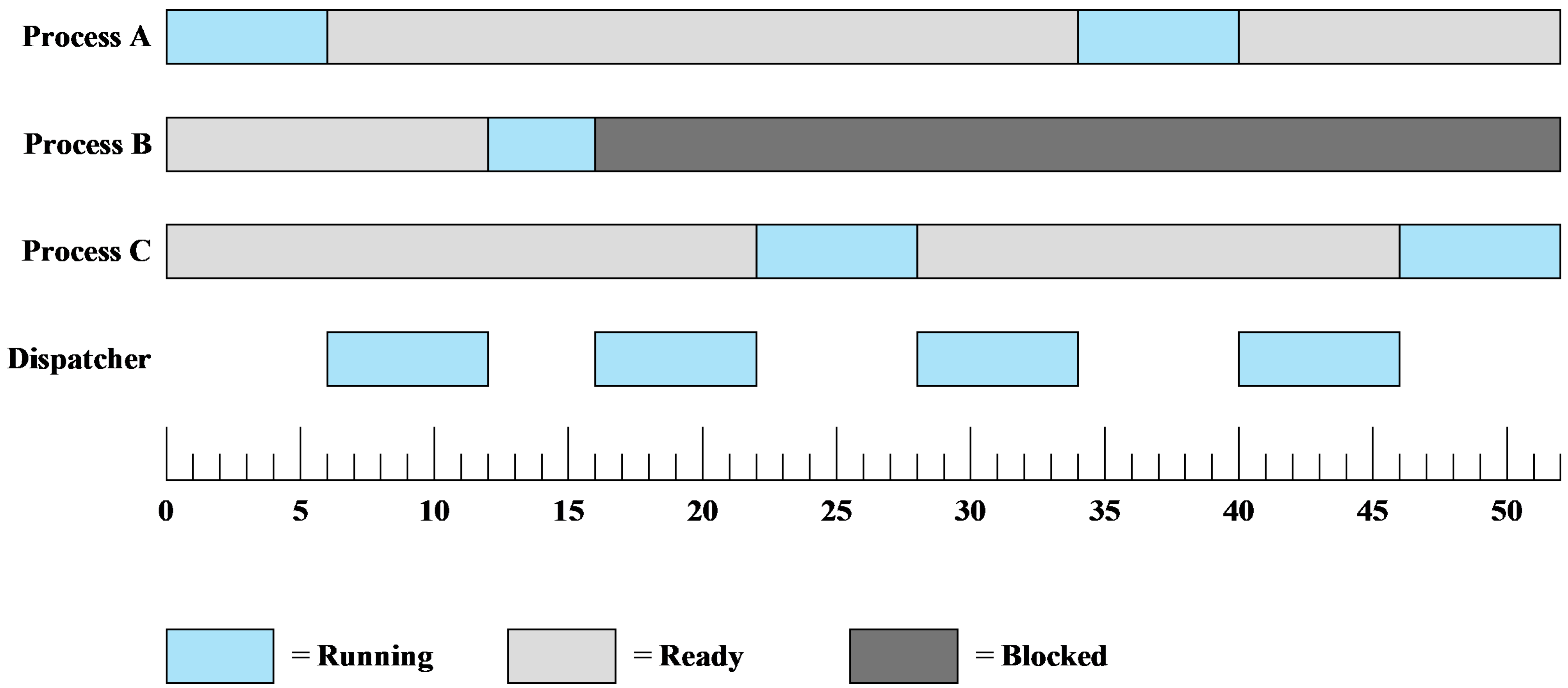
(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

1	5000
2	5001
3	5002
4	5003
5	5004
6	5005
-------------------- Timeout	
7	100
8	101
9	102
10	103
11	104
12	105
13	8000
14	8001
15	8002
16	8003
----------------I/O Request	
17	100
18	101
19	102
20	103
21	104
22	105
23	12000
24	12001
25	12002
26	12003
27	12004
28	12005
-------------------- Timeout	
29	100
30	101
31	102
32	103
33	104
34	105
35	5006
36	5007
37	5008
38	5009
39	5010
40	5011
-------------------- Timeout	
41	100
42	101
43	102
44	103
45	104
46	105
47	12006
48	12007
49	12008
50	12009
51	12010
52	12011
-------------------- Timeout	

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed



Process A
Process B
Process C
Dispatcher
0
5
10
15
20
25
30
35
40
45
50
= Running
= Ready
= Blocked

Role of the Process Control Block

- The most important data structure in an OS
  - contains all of the information about a process that is needed by the OS
  - blocks are read and/or modified by virtually every module in the OS
  - defines the state of the OS
- Difficulty is not access, but protection
  - a bug in a single routine could damage process control blocks, which could destroy the system's ability to manage the affected processes
  - a design change in the structure or semantics of the process control block could affect a number of modules in the OS

Process Creation

- Once the OS decides to create a new process it:

1. assigns a unique process identifier to the new process
2. allocates space for the process
3. initializes the process control block
4. sets the appropriate linkages
5. creates or expands other data structures

Mechanisms for Interrupting the Execution of a Process

Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition
System Call	Explicit request	Call to an operating system function

System Interrupts

Interrupt

- Due to some sort of event that is external to and independent of the currently running process
    - clock interrupt
    - I/O interrupt
    - memory fault
- Time slice
    - the maximum amount of time that a process can execute before being interrupted

Trap

- An error or exception condition (syscall) generated within the currently running process
- OS determines if the condition is fatal
    - moved to the Exit state and a process switch occurs
    - action will depend on the nature of the error

Mode Switching

If no interrupts are pending the processor:

proceeds to the fetch stage and fetches the next instruction of the current program in the current process

If an interrupt is pending the processor:

sets the program counter to the starting address of an interrupt handler program

switches from user mode to kernel mode so that the interrupt processing code may include privileged instructions

Change of Process State

The steps in a full process switch are:

1. save the context of the processor
2. update the PCB of the process currently in the Running state
3. move the PCB of this process to the appropriate queue
4. select another process for execution
5. update the PCB of the process selected
6. update memory management data structures
7. restore the context of the processor to that which existed at the time the selected process was last switched out

If the currently running process is to be moved to another state (Ready, Blocked, etc.), then the OS must make substantial changes in its environment

Today

- Processes
- Process States
- Unix Processes
- XV6 Processes
- Scheduling
- XV6 Scheduling

Process Hierarchy (UNIX)

- In UNIX, each process (aside from init) has a *parent process* that created it
- This relationship creates a tree structure, rooted at the init process
- Processes are often managed as a group consisting of a single process and all its descendants
  - E.g., signals (such as SIGTERM, to request termination) are delivered to parent and all its descendants

Process Tree

```
systemd─┬─ModemManager─┬─{gdbus}
        │              └─{gmain}
        ├─acpid
        ├─agetty
        ├─cron
        ├─dbus-daemon
        ├─dhclient
        ├─fail2ban-server───2*[{fail2ban-server}]
        ├─master─┬─pickup
        │        └─qmgr
        ├─mosh-server───bash───tmux
        ├─9*[mosh-server───bash]
        ├─ntpd
        ├─polkitd─┬─{gdbus}
        │         └─{gmain}
        │
```

Killing Processes

- Processes can exit on their own using a system call
- One process can request that the OS terminate (*kill*) another process
  - Depending on the privilege of the requestor, this may or may not be granted

Killing Processes

- In most operating systems, process termination notifies the process to be terminated first so it can cleanly free resources
- More forceful methods exist:
  - kill -9
  - taskkill /f
- In all cases, the operating system will free the resources that it has granted to the process

Hierarchy and Termination (UNIX)

- If the parent process exits before its children, those children are said to be *orphaned*
- Orphaned child processes are *adopted* by init

Zombies (UNIX)

- *Zombies* are processes that have exited but still have an entry in the process table
- If a child process exits and the parent has not called wait() to get its exit status, the child becomes a zombie
- This is because the exit status is stored by the kernel-side data structure, and the kernel can't free it until it knows no one needs the exit status

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>

int main() {
    if (fork() == 0) {
        // Exit immediately in the child
        exit(1);
    }
    // Sleep forever without calling wait()
    // in the parent
    while(true);
    return 0;
}
```

USER	PID	%CPU	%MEM	VSZ	RSS	TT	STAT	STARTED	TIME	COMMAND
moyix	7622	100.0	0.0	2432748	548	s004	R+	8:05AM	0:15.23	./zombo
moyix	7623	0.0	0.0	0	0	s004	Z+	8:05AM	0:00.00	(zombo)

zombie

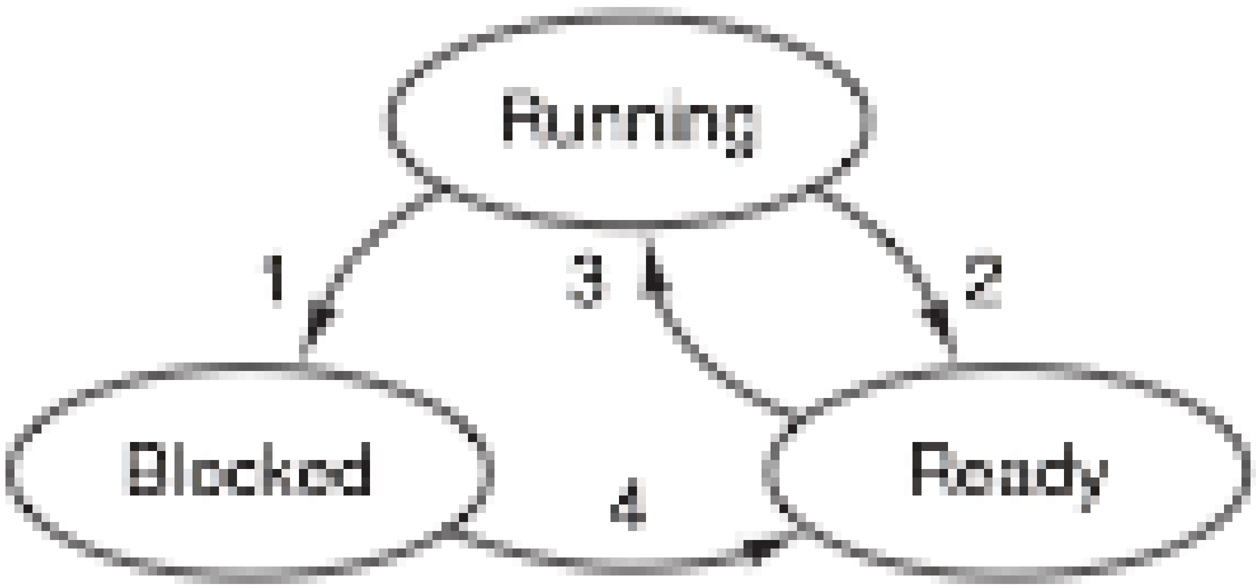
Process ~~Hierarchy~~ Anarchy (Windows)

- Windows does keep track of parent/child relationships
- These don't actually affect anything though
- Instead, when a process is created its parent gets a *handle* – a token that it can use to control the child
- Handles can be passed to other processes, and a process with sufficient privilege can obtain a handle to another process with OpenProcess

Process States

- Processes can generally be in one of three states:
  - *Running* – actively using the CPU right now
  - *Ready* – not currently running
  - *Blocked* – waiting for I/O
    - Even if the OS scheduler wants to run this process, it can't – nothing for it to do

Transitions



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

I/O Waiting

- What causes processes to block?
  - Waiting for input
  - Explicit sleep
- The kinds of input can be things like network I/O, disk, waiting for the user to click on something, etc.

Process Anatomy: Kernel

- The kernel **tracks** what processes are active in a *process table*
  - Note: in most OSes, this is not a literal array, but rather something like a doubly-linked list
- Each process gets a structure bit of metadata called the *process control block (PCB)*

Today

- Processes
- Process States
- Unix Processes
- XV6 Processes
- Scheduling
- XV6 Scheduling

Process Control Block in xv6

```
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)
};
```

xv6 Process Table

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

Process Threads (aside)

- Each process has a thread of execution that executes the process instructions.
- Thread can be suspended and later resumed.
- Much of the state of a thread (local vars, ret address) stored on thread's stack.
- Each process has 2 stacks: user-stack and kernel stack (p->kstack)

Process Threads

- Process executing user instructions => user stack in use, kernel stack empty

- Process enters kernel thru sys_call or interrupt => kernel executes on the process's kernel stack. User stack contains saved data but not actively used.

- Kernel stack is separate and protected from user code.

Process Threads

- When a process makes a system call:
  - processor switches to kernel stack
  - raises hardware privilege level.
  - Starts executing the kernel instructions that implement the system call.

Process Threads

- When System Call Completes:
    - Kernel returns to user space
    - hardware lowers privilege level.
    - Switches back to user stack
    - Resumes executing user instructions just after the system call

Kernel Stack

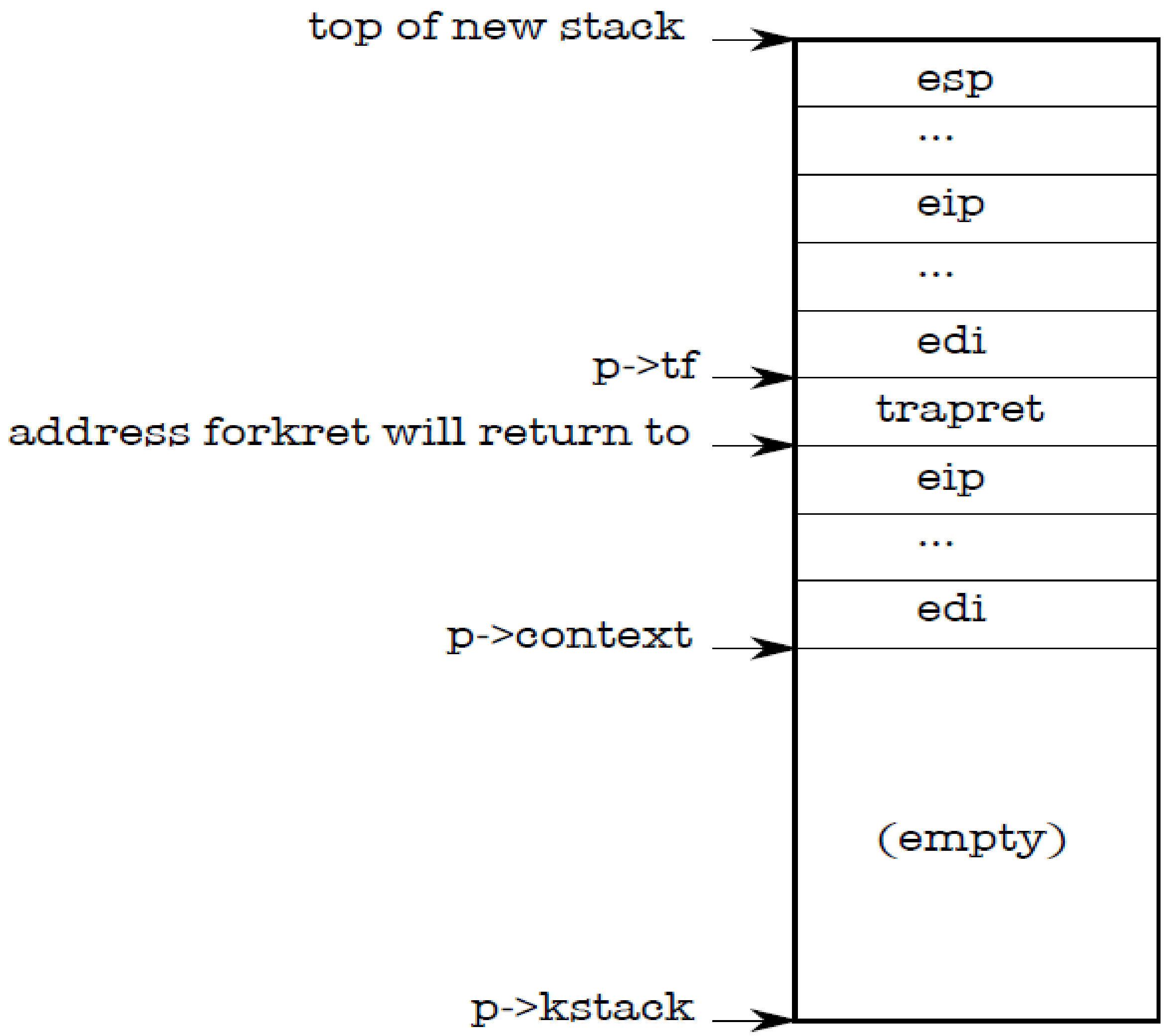
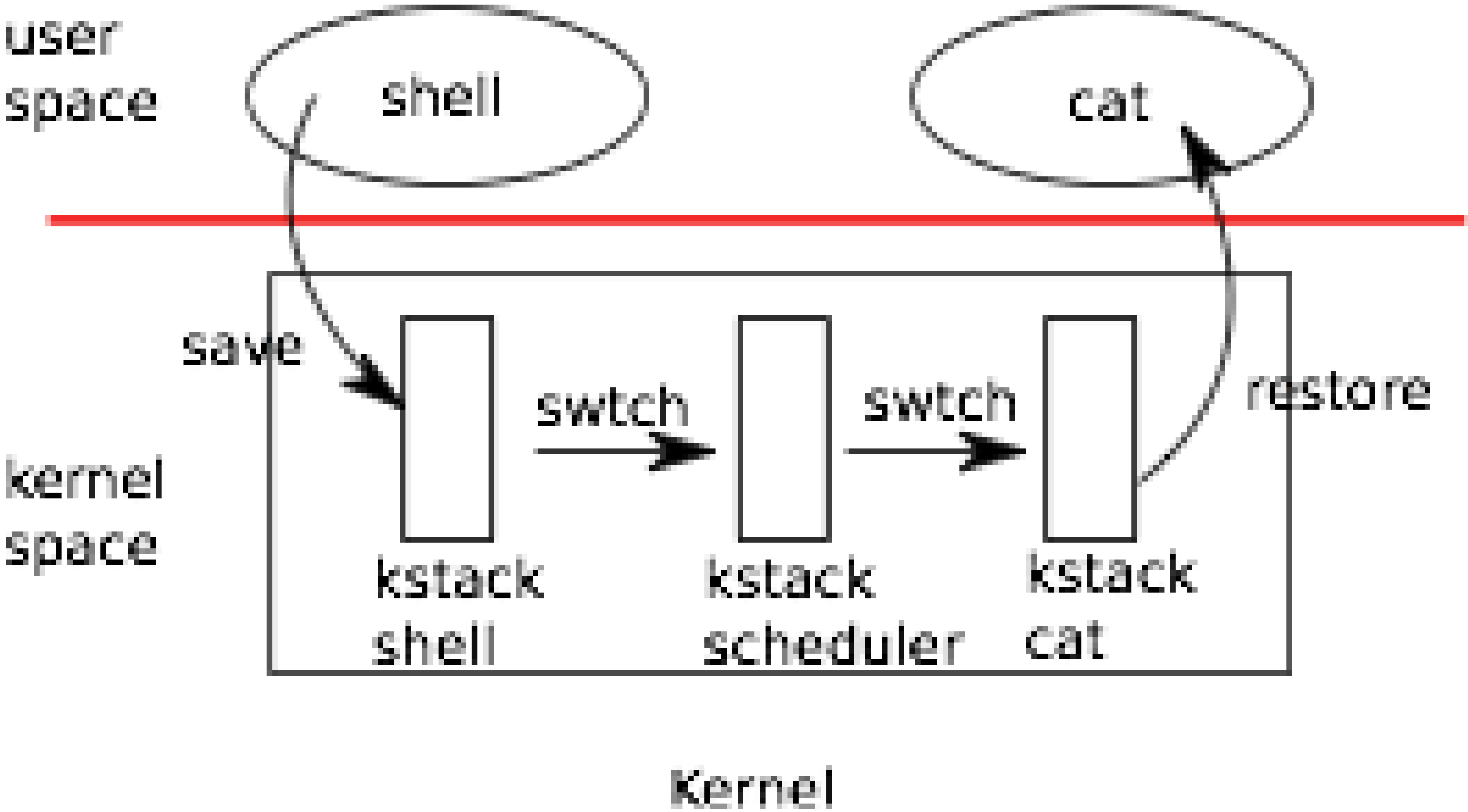


Figure 1-4. A new kernel stack.


user
space
shell
cat
save
swtch
swtch
restore
kernel
space
kstack
shell
kstack
scheduler
kstack
cat
Kernel

Create Proc

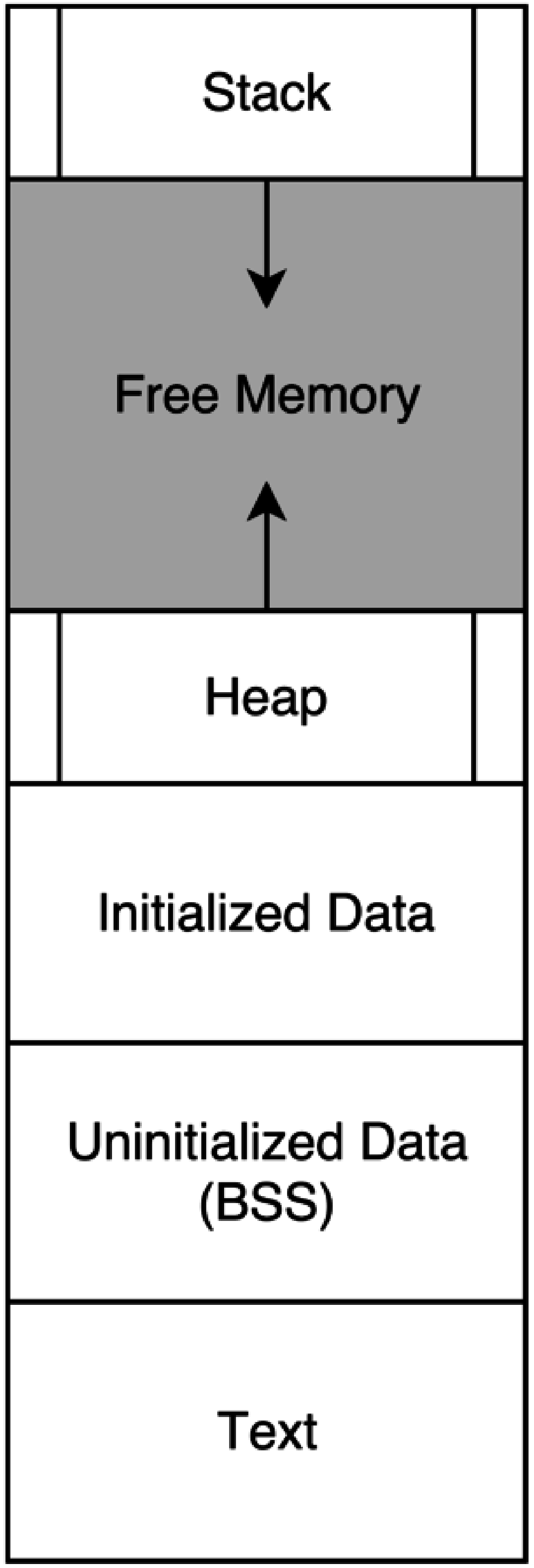
```c
// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  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;
```


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

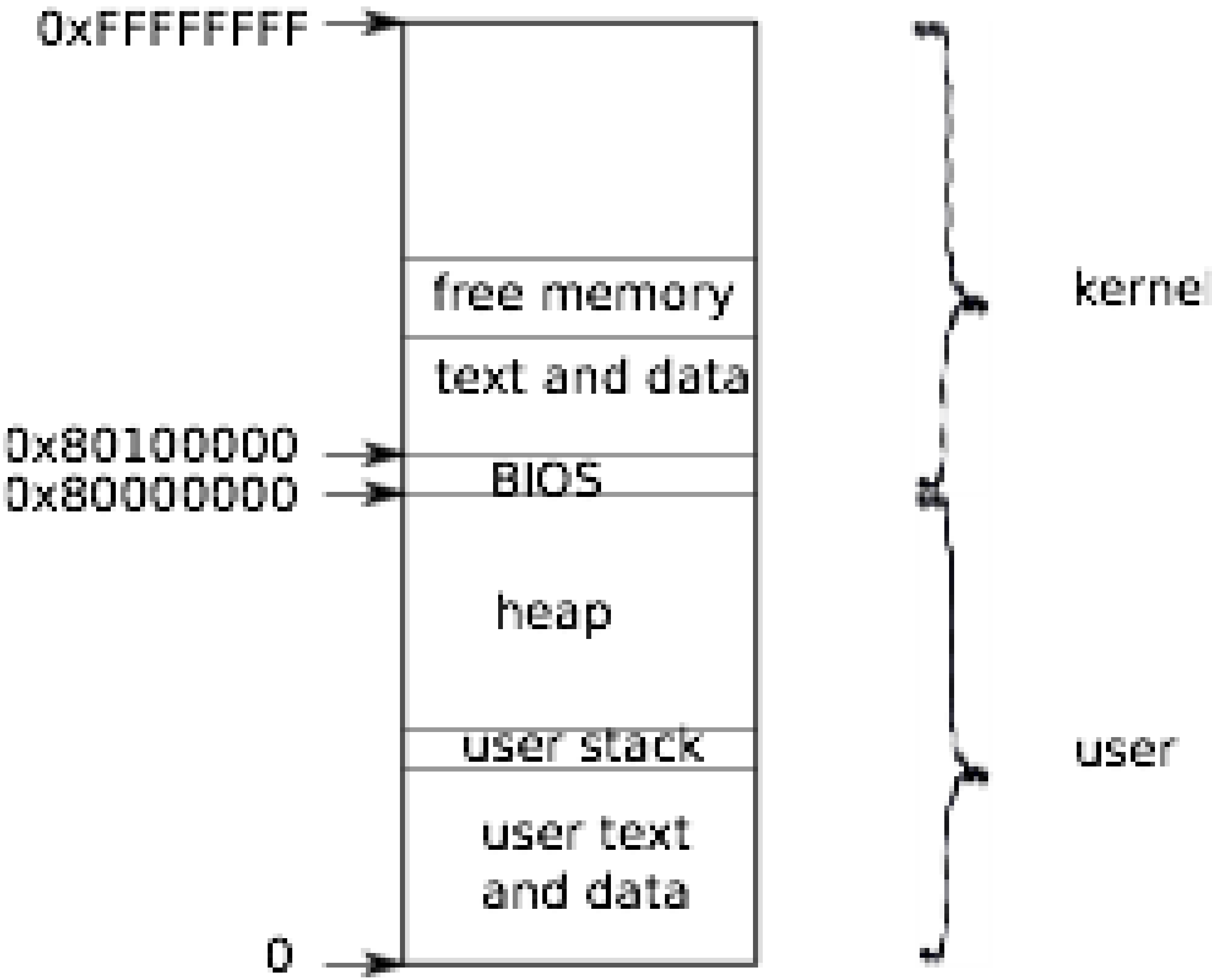

```c

  // 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;
}
```

Process Anatomy: User Space

- Text segment: actual program code
- Data segment: program data
  - .data - initialized, read/write variables
  - .bss - uninitialized, read/write variables
  - .rodata - initialized, read-only variables (constants)
- Stack
- Heap - dynamic allocations



xv6 Address Space Layout



Today

- Processes
- Process States
- Unix Processes
- XV6 Processes
- Scheduling
- XV6 Scheduling

Scheduling

- To coordinate the running of multiple processes, the OS includes a *scheduling algorithm* that decides what process will be run when, and for how long
- There are tons of different scheduling algorithms with different goals:
  - Maximize CPU utilization
  - Maximize I/O throughput
  - Make the system feel responsive to the user
  - Meet real-time deadlines

Other Scheduling Considerations

- May want different process *priorities*
- May want to handle I/O bound vs CPU-bound processes differently
- May want to ensure *fairness* – make sure each process gets time to run

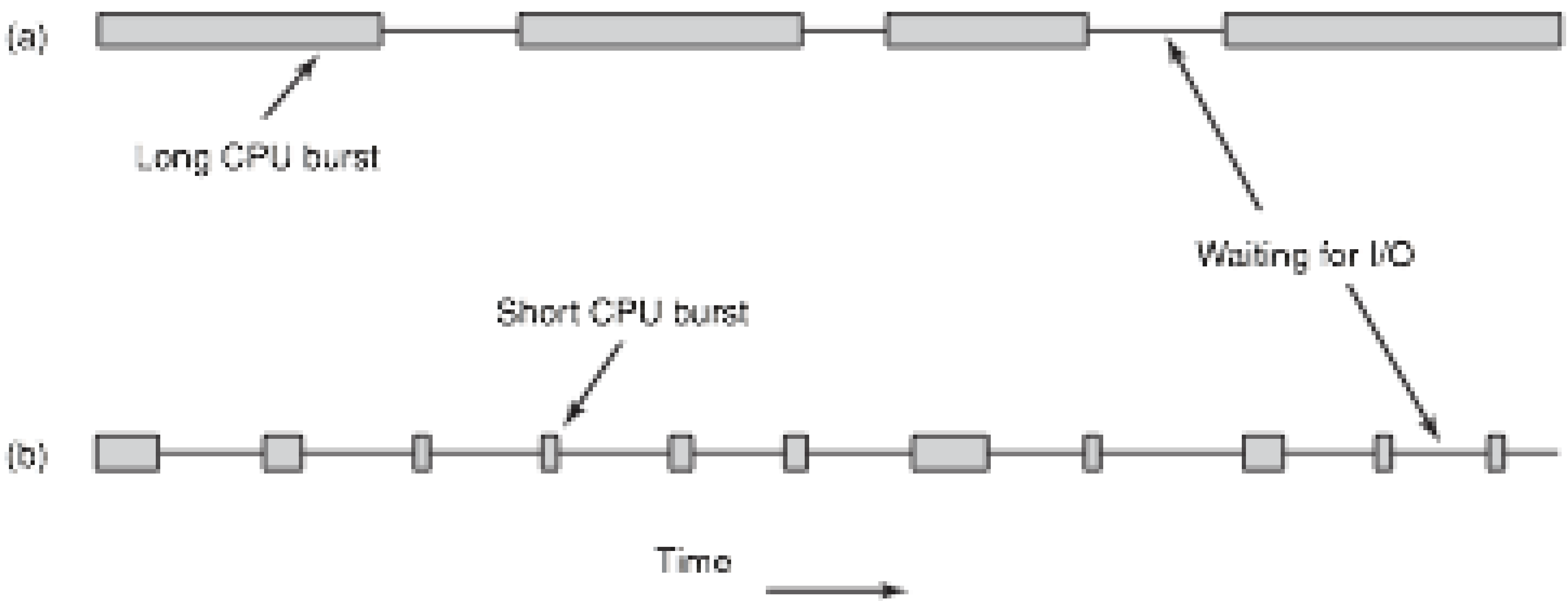


Figure 2-39. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

When to Schedule

- At timed intervals
  - We can use the hardware *timer interrupt* for this
- When a process exits
  - No process is running now so we must choose one
- When a process blocks on I/O
- When an interrupt happens
  - May signal that I/O is done, and we want to unblock the relevant process

Preemption

- Schedulers can be further classified by whether or not processes can choose when they stop running

- Non-Preemptive schedulers let processes run until they block on I/O or yield voluntarily

- Preemptive schedulers take advantage of a timer interrupt

Round Robin Scheduling

- A simple, *preemptive* scheduling algorithm
- Run first process until its quantum is used up
- Move that process to the end and run the next process until its quantum is used up
- Simple, fair

Today

- Processes
- XV6 Processes
- Scheduling
- XV6 Scheduling

Multiplexing

- An operating system is likely to run with more processes than there are processors!
- OS provides the illusion that it has it’s own (virtual) processor
- The OS does this by multiplexing multiple virtual processor on a single physical processor

Multiplexing

- XV6 multiplexes by switching each processor from one process to another in 2 situations:

  - **Sleep and wakeup** mechanism – wait for device or pipe (I/O)

  - Periodically forces switch when **quantum** ends

Multiplexing Challenges

1. How to switch from one process to another? Implementation is tricky!

2. How to do context switching transparently?

3. Third how to handle multiple CPUs and avoid race conditions

4. Free memory after exit, but careful with kernel memory.

xv6 Scheduler

- xv6 uses *round robin*
- Whenever scheduling happens, loop over the array until we find a runnable process and switch to it
- When the timer runs or the process yields, we return to the scheduler loop
- Importantly, we return to the loop at the exact place we left off – this is what makes it *fair* and ensures we don't starve processes later in the list

Context (proc.h)

```c
// 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;
};
```

```
Void scheduler(void)
{
  struct proc *p;

  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.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

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

Context Switching

- The actual scheduler context switch (to start executing the process) happens here:
- Let's examine these in more detail

```
// Switch to chosen process.  It is the process's job
// to release ptable.lock and then reacquire it
// before jumping back to us.
proc = p;
switchuvm(p);
p->state = RUNNING;
swtch(&cpu->scheduler, proc->context);
switchkvm();
```

switchuvm

- This function does two things:
  - Switches the *task state segment* to the user-mode one
  - Changes the current *virtual address space* to the process's

Task State Segment (TSS)

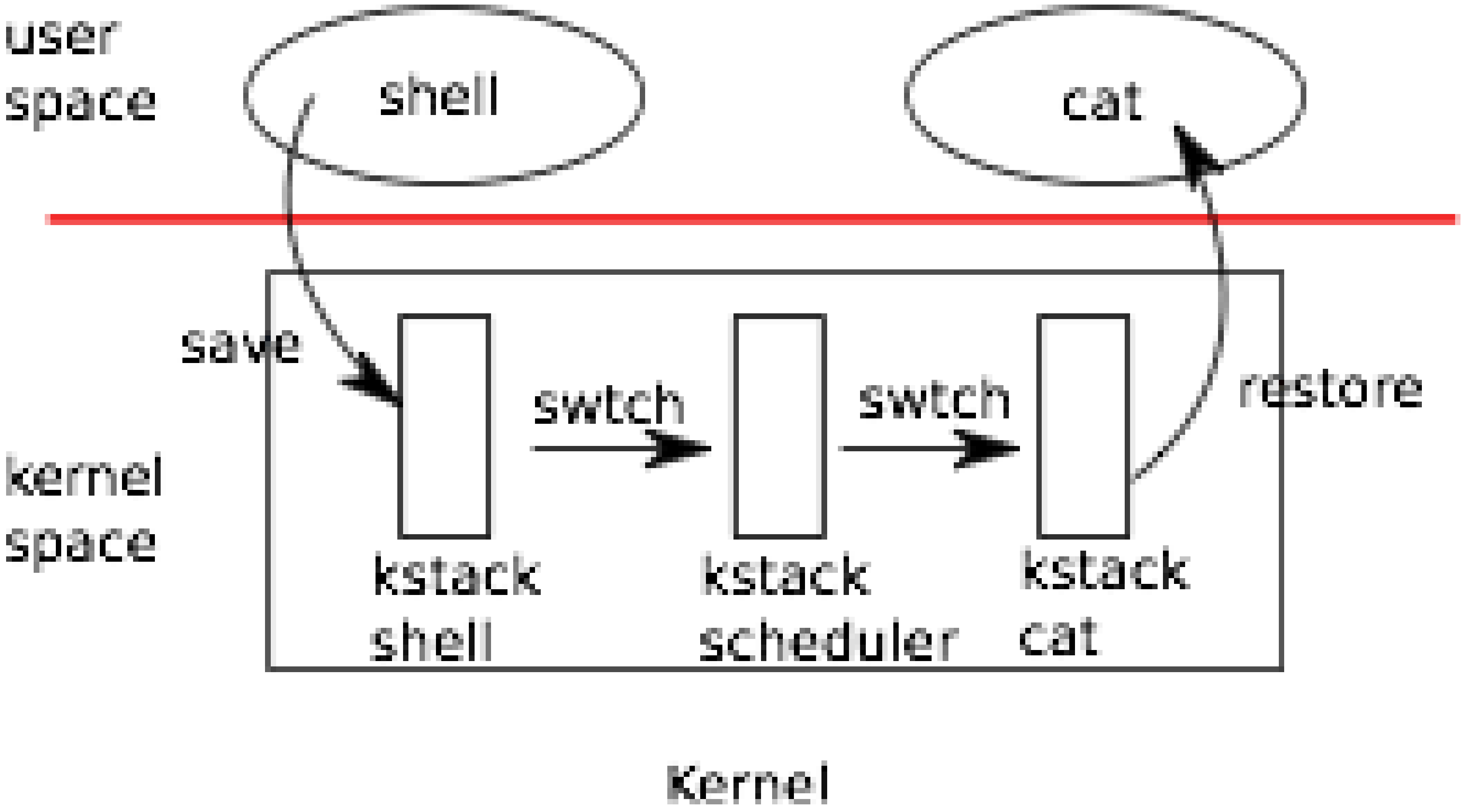
- The TSS is a special structure defined by x86 intended to help with *context switching*
- However, hardware context switching is slow, so most OSes make minimal use of the TSS (*software context switching*)
- Some parts are still necessary though:
    - Location of kernel stack for *this process*
    - Segment descriptor for kernel stack
- It is possible to have multiple TSS (e.g. one for each process), but in practice a single one is used for all processes

31	15	0	
I/O Map Base Address	Reserved	T	100
Reserved	LDT Segment Selector		96
Reserved	GS		92
Reserved	FS		88
Reserved	DS		84
Reserved	SS		80
Reserved	CS		76
Reserved	ES		72
EDI			68
ESI			64
EBP			60
ESP			56
EBX			52
EDX			48
ECX			44
EAX			40
EFLAGS			36
EIP			32
CR3 (PDBR)			28
Reserved	SS2		24
ESP2			20
Reserved	SS1		16
ESP1			12
Reserved	SS0		8
ESP0			4
Reserved	Previous Task Link		0

Reserved bits. Set to 0.

Figure 7-2. 32-Bit Task-State Segment (TSS)

swtch


user
space
shell
cat
save
swtch
swtch
restore
kernel
space
kstack
shell
kstack
scheduler
kstack
cat
Kernel

```c
void
scheduler(void)
{
  struct proc *p;

  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.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

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

  }
}
```

swtch

- Does the actual work of switching between the kernel scheduler context and the kernel process context
- Only needs to save a few registers:

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

What About the Others?

- Don't need to save segment registers because those are the same for all kernel contexts
- Don't need to save eax, ecx, edx – in the gcc calling convention, these are not assumed to persist across function calls
- So any code that calls swtch() will automatically preserve them

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.

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

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

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

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

Load arguments into eax and edx
Notice do this before switching stacks below

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.

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

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

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

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

Push the register state

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.

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

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

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

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

Note: EIP is *implicitly* saved by executing the call – call pushes the EIP of the next instruction onto the stack

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

swtch-ing Back

- When a process gives up the CPU, yield() is called
- yield() makes the process runnable and then calls sched()
  - Note: *not* the same as scheduler()

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  proc->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

sched()

```
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state.
void
sched(void)
{
  int intena;

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

sched()

```
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state.
void
sched(void)
{
  int intena;

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

Where does this go?

Here

```
void
scheduler(void)
{
 struct proc *p;

 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.
   proc = p;
   switchuvm(p);
   p->state = RUNNING;
   swtch(&cpu->scheduler, proc->context);
   switchkvm();

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

 }
}
```