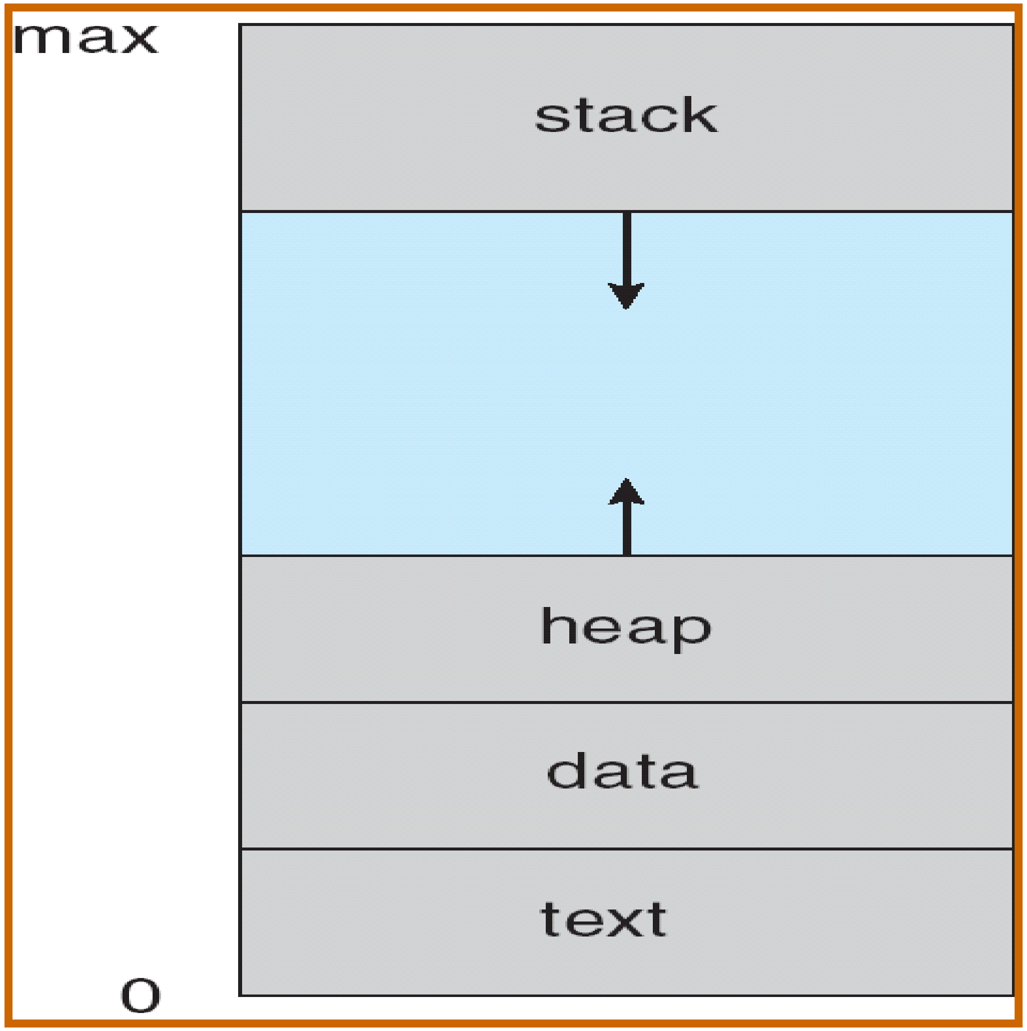
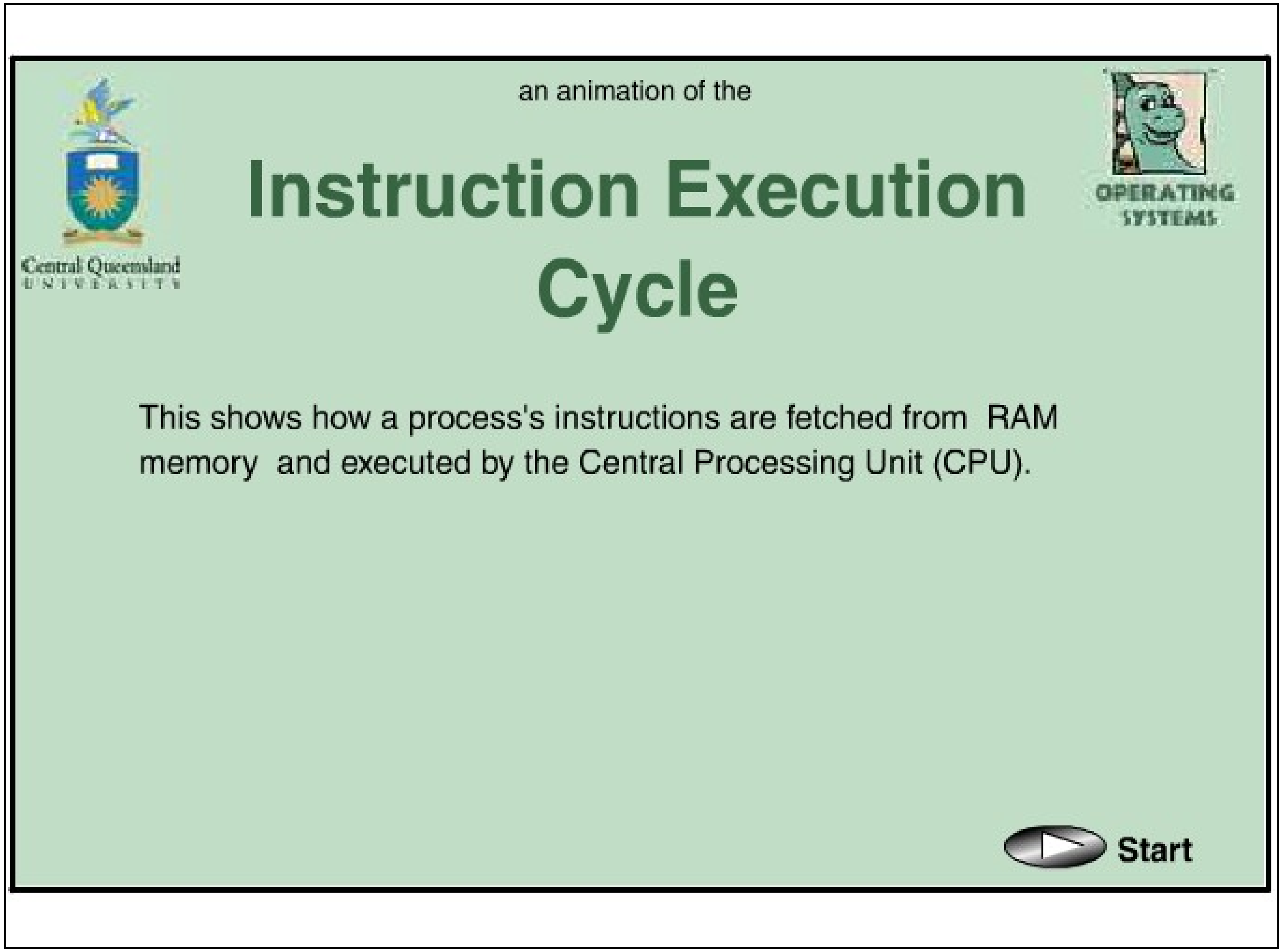
Operating Systems
Processes

A Biswas

Process Concept



Process Concept



an animation of the

Instruction Execution Cycle

This shows how a process's instructions are fetched from RAM memory and executed by the Central Processing Unit (CPU).

Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

Diagram of Process State

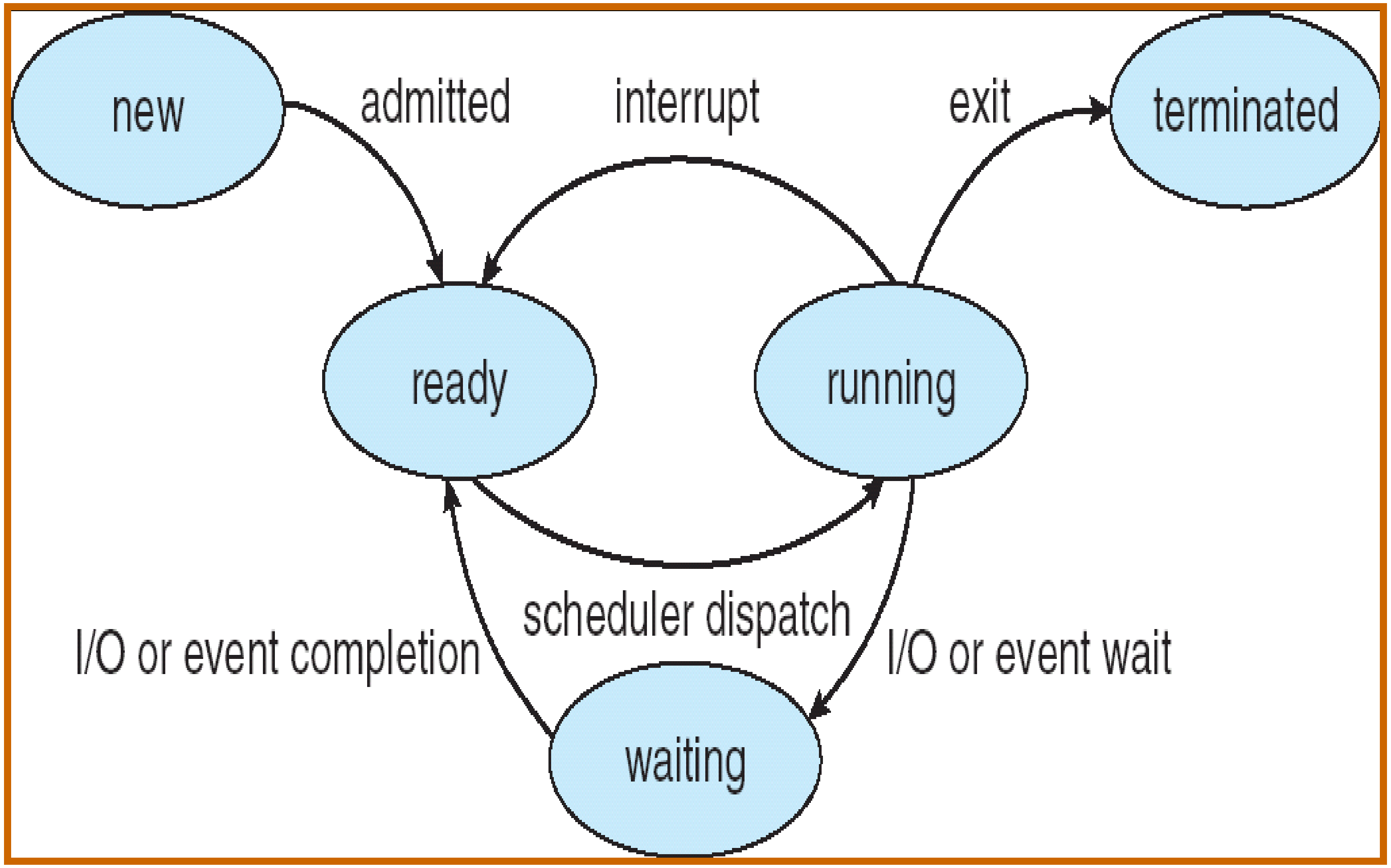
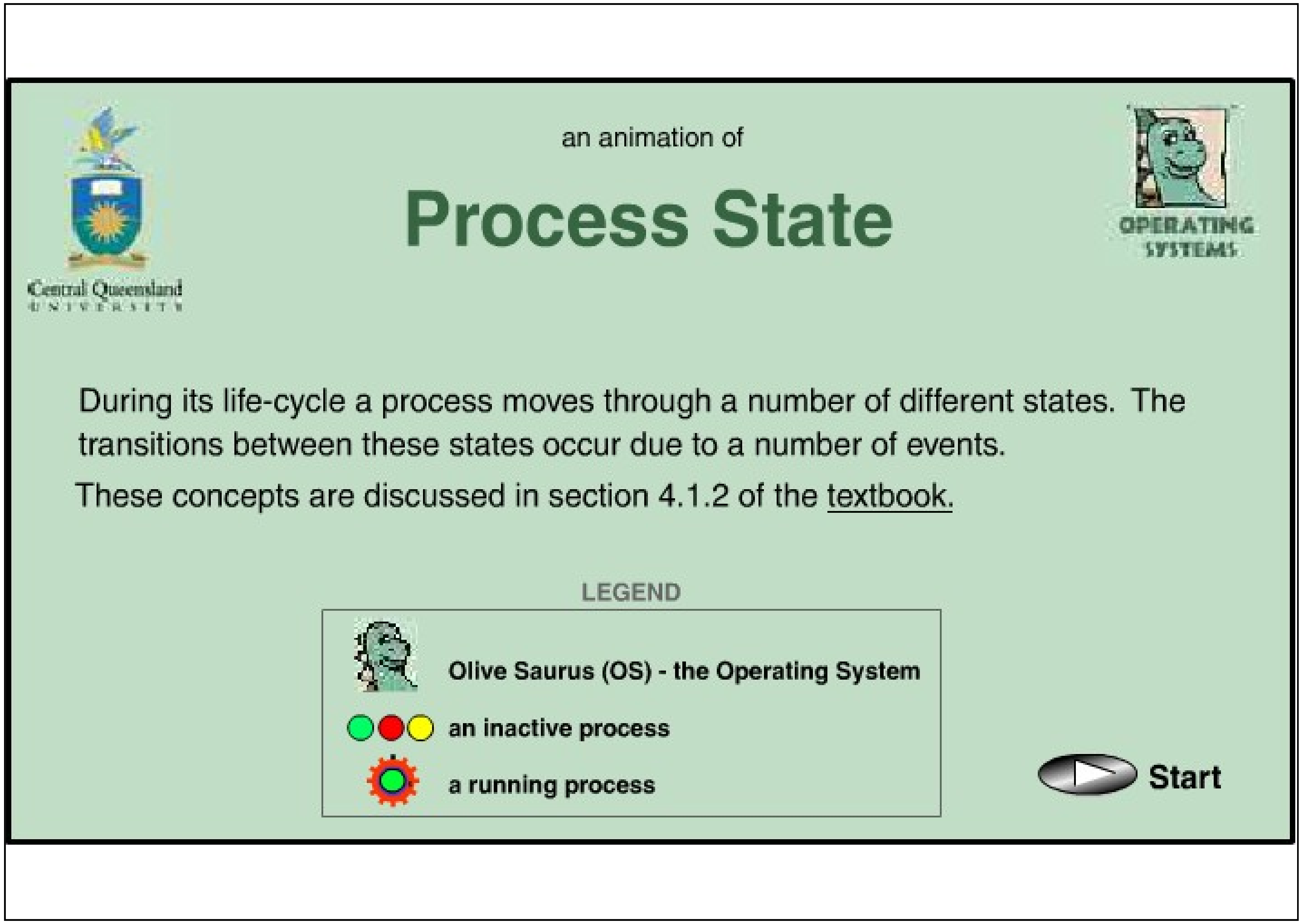


Diagram of Process State



an animation of

Process State

Central Queensland UNIVERSITY

OPERATING SYSTEMS

During its life-cycle a process moves through a number of different states. The transitions between these states occur due to a number of events.

These concepts are discussed in section 4.1.2 of the textbook.

LEGEND

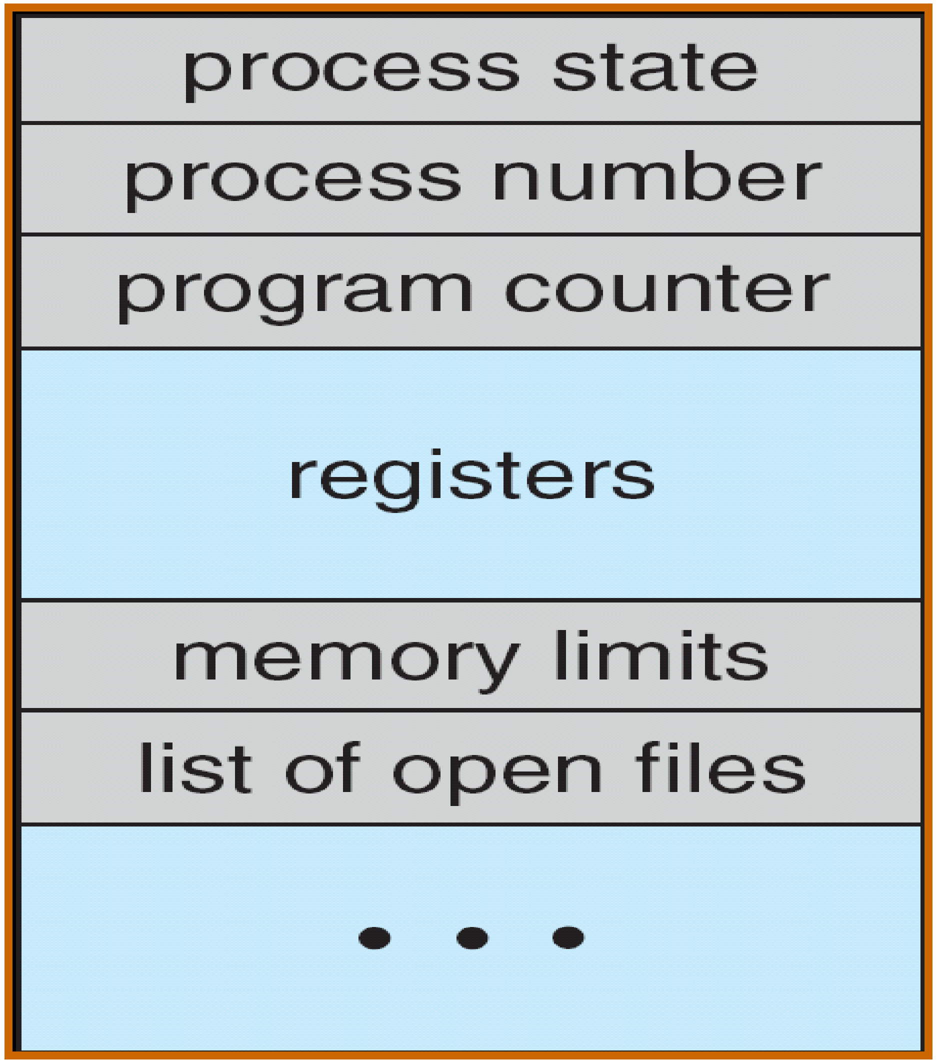
	Olive Saurus (OS) - the Operating System
🟢🔴🟡	an inactive process
●	a running process

Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process Control Block (PCB)



DATA STRUCTURE: THE PROCESS LIST

Operating systems are replete with various important **data structures**.

The **process list** is the first such structure. It is one of the simpler ones, but certainly any OS that has the ability to run multiple programs at once will have something akin to this structure in order to keep track of all the running programs in the system.

Sometimes people refer to the individual structure that stores information about a process as a **Process Control Block** (**PCB**).

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};
```

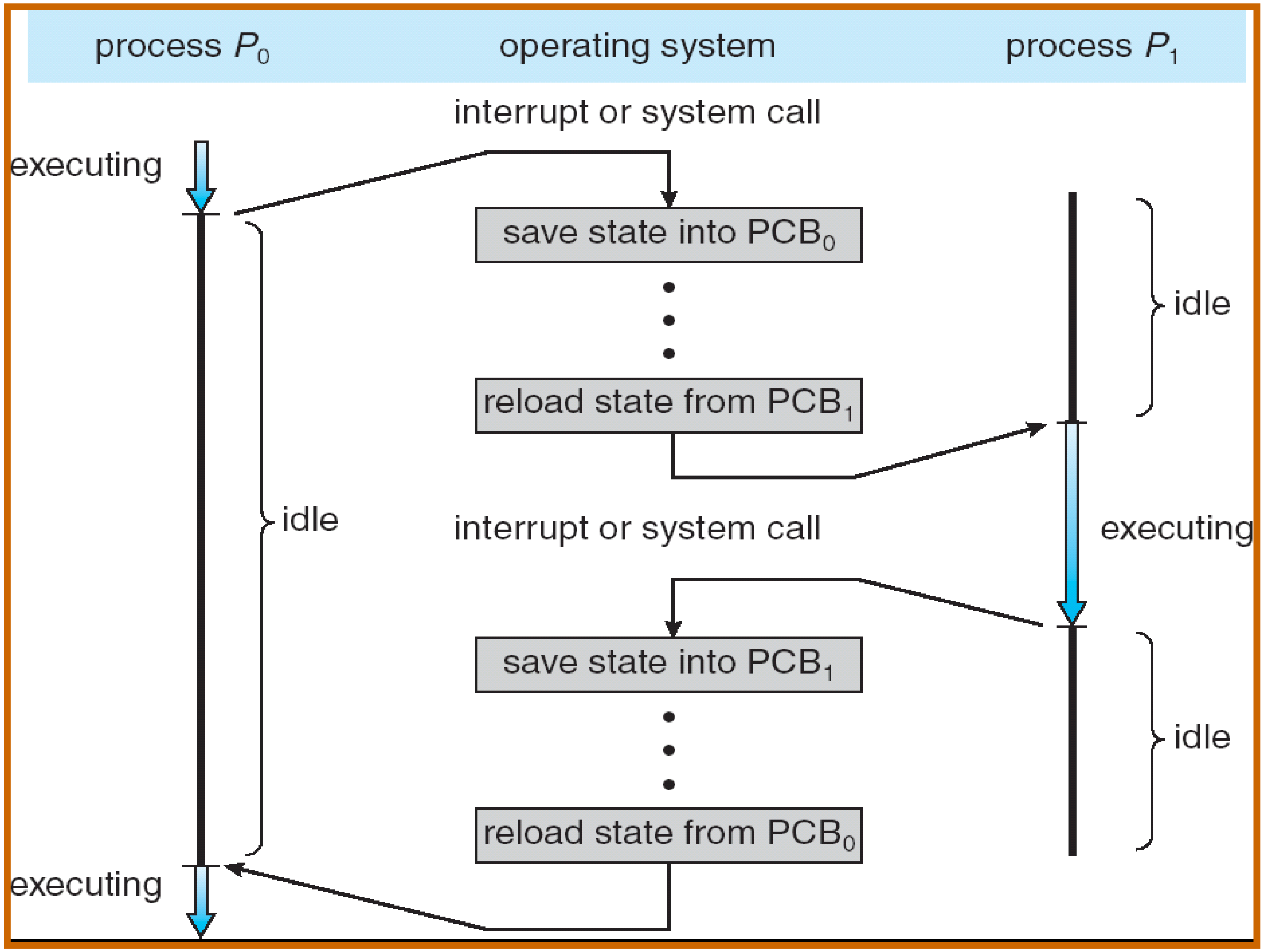
```
  // the different states a process can be in
  enum proc_state { UNUSED, EMBRYO, SLEEPING,
                    RUNNABLE, RUNNING, ZOMBIE };
```

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                   // Start of process memory
  uint sz;                     // Size of process memory
  char *kstack;                // Bottom of kernel stack
                               // for this process
  enum proc_state state;       // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent 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
  struct context context;      // Switch here to run process
  struct trapframe *tf;        // Trap frame for the
                               // current interrupt
};
```

Assignment:

Find out the details and complexity of these structures in Linux, UNIX, and Windows.

CPU Switch From Process to Process

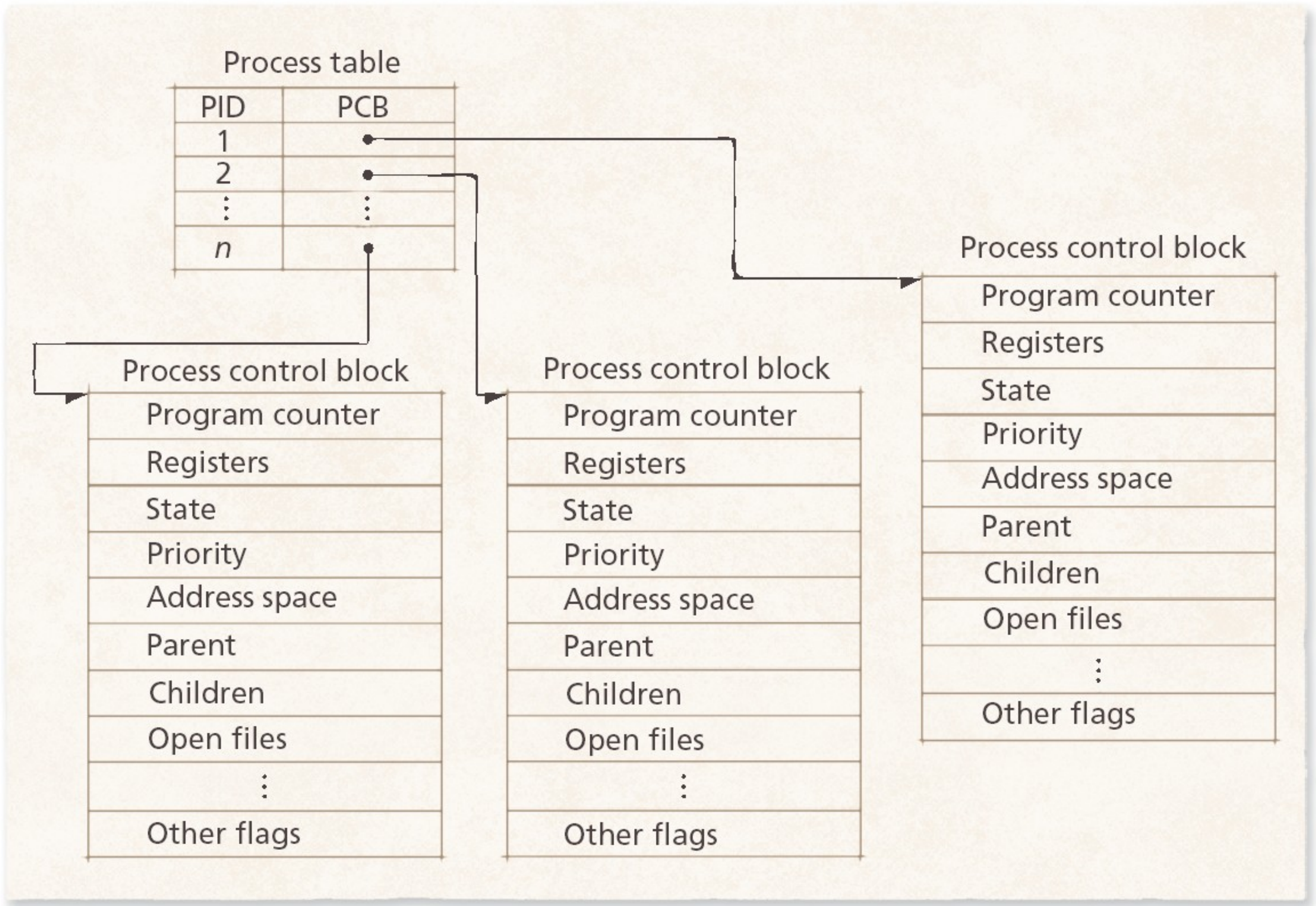


PCBs / Process Descriptors

- Process table
  - The OS maintains pointers to each process's PCB in a system-wide or per-user process table
  - Allows for quick access to PCBs
  - When a process is terminated, the OS removes the process from the process table and frees all of the process's resources

PCBs / Process Descriptors

Process table and process control blocks.

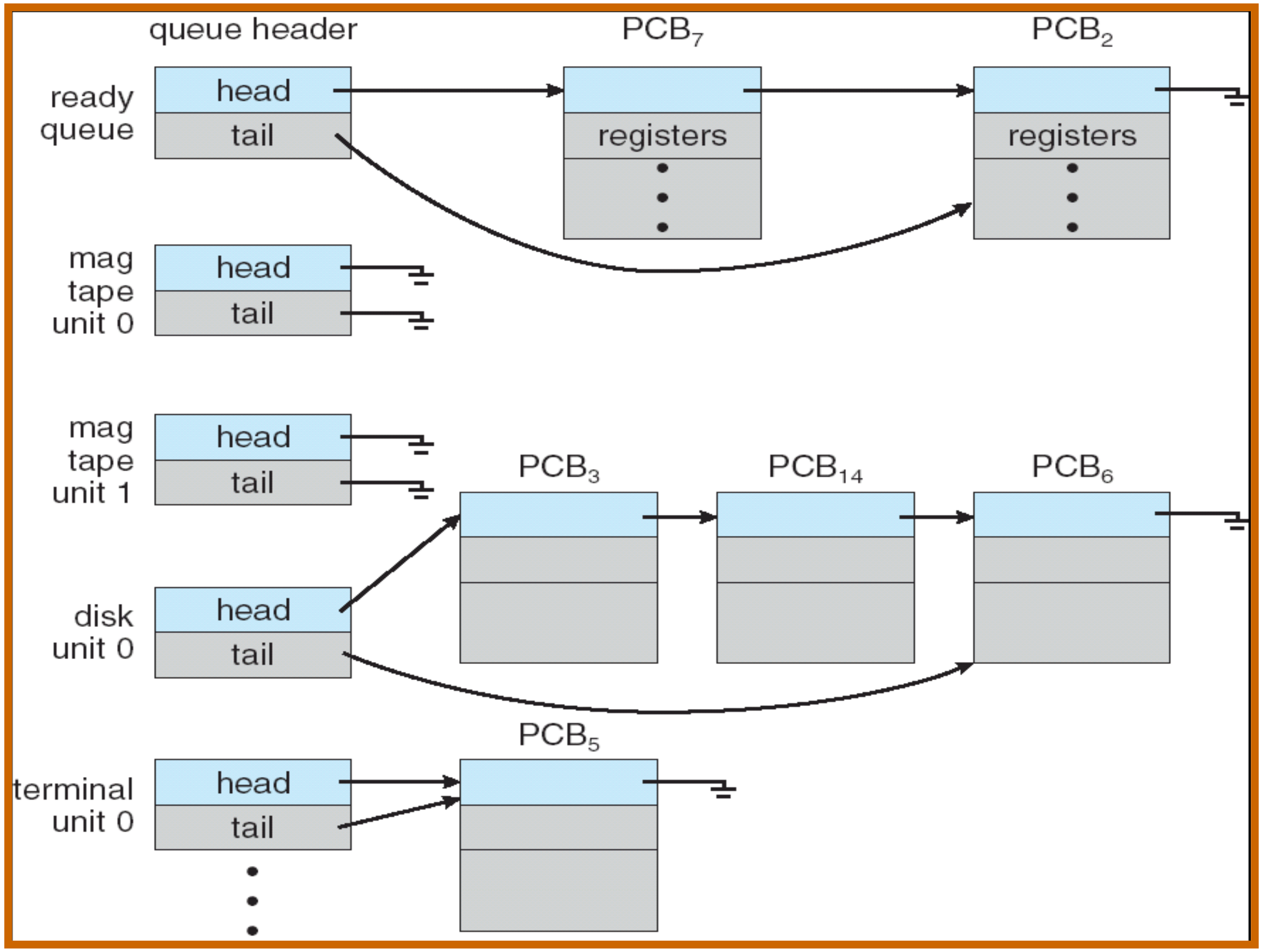


Process Scheduling Queues

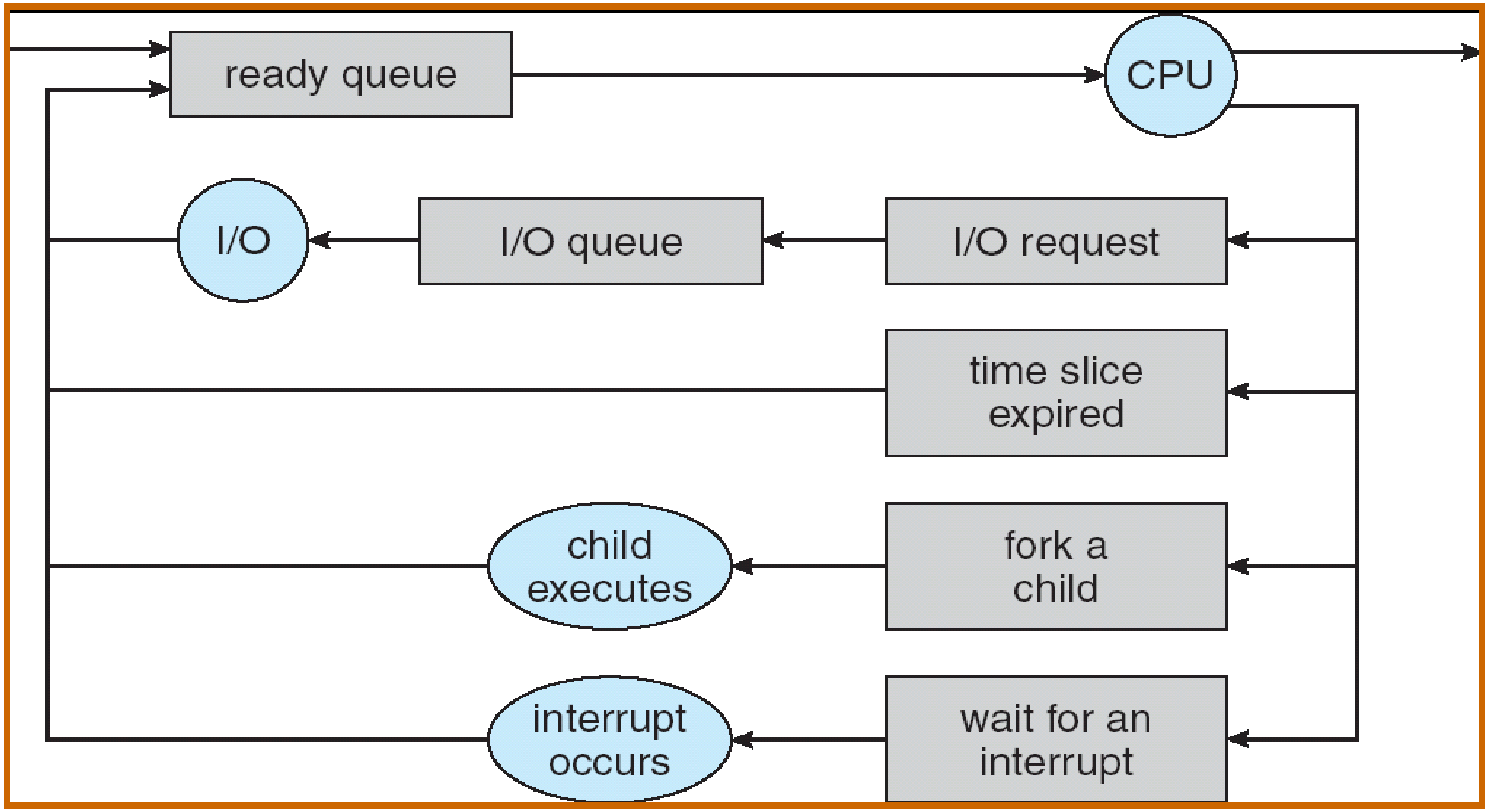
- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device

Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



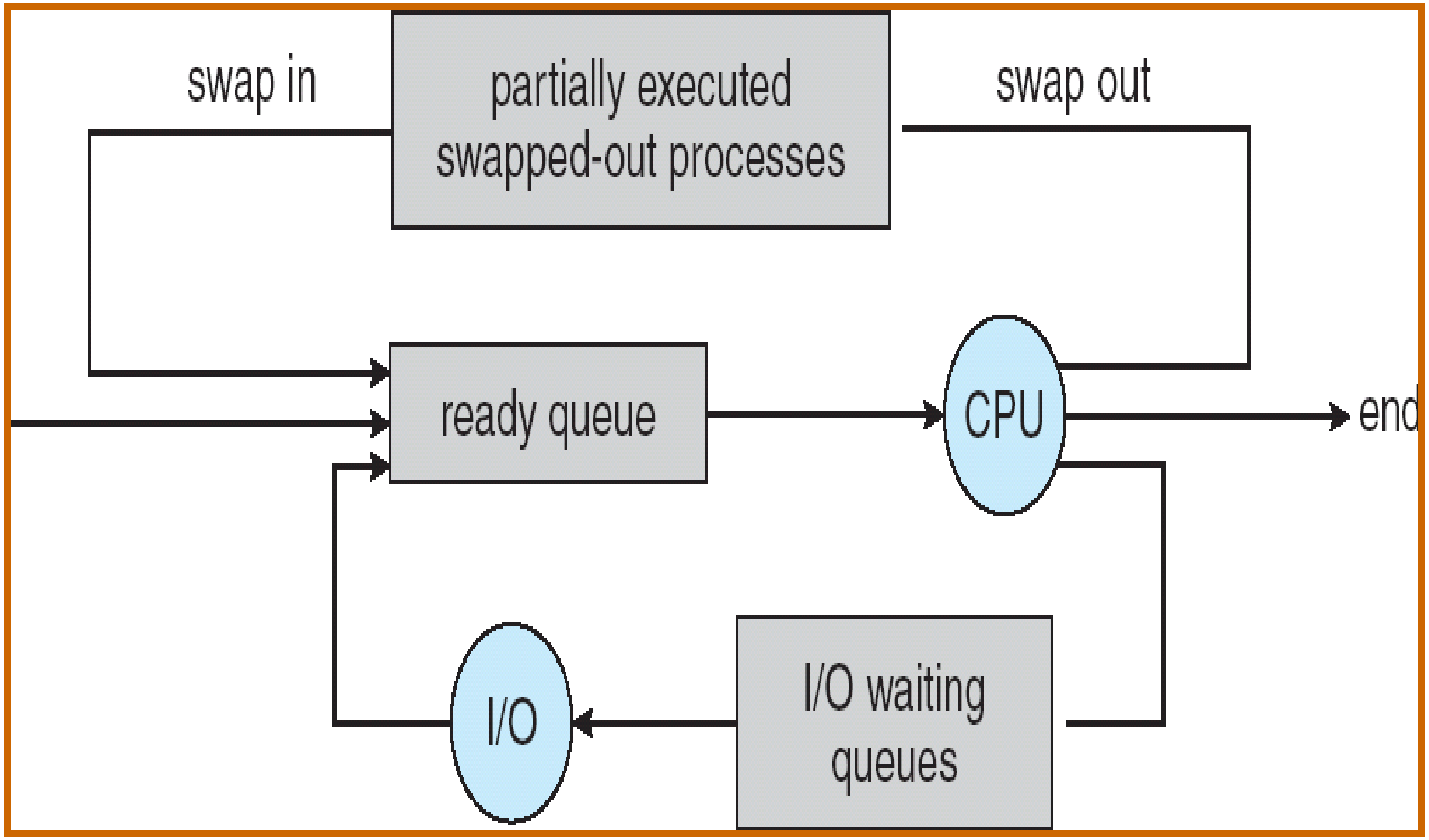
Representation of Process Scheduling



Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Addition of Medium Term Scheduling



Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

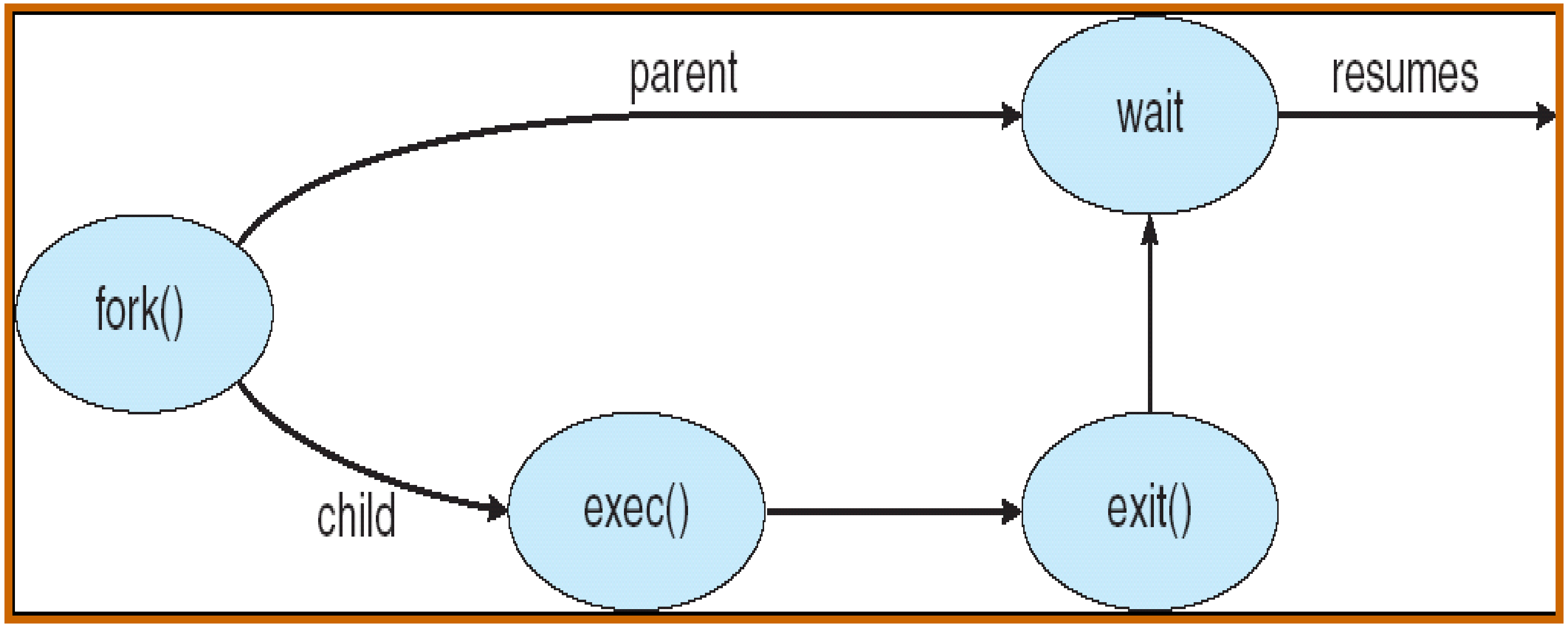
Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources,
  - Children share subset of parent's resources, or
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently, or
  - Parent waits until children terminate

Process Creation (Cont.)

- Address space
  - Child duplicate of parent, or
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process’ memory space with a new program

Process Creation



fork()

The child isn’t an exact copy. Specifically, although it now has its own copy of the address space (i.e., its own private memory), its own registers, its own PC, and so forth, the value it returns to the caller of **fork()** is different.

Specifically, while the parent receives the PID of the newly-created child, the child is simply returned a 0. This differentiation is useful, because it is simple then to write the code that handles the two different cases.

C Program Forking Separate Process

```
int main()
{
      pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
            /* parent will wait for the child to complete */
            wait (NULL);
            printf ("Child Complete");
            exit(0);
    }
}
```

fork()

The output is not **deterministic**.

When the child process is created, there are now two active processes in the system: the parent and the child.

wait()

Sometimes, as it turns out, it is quite useful for a parent to wait for a child process to finish what it has been doing.

This task is accomplished with the wait() system call. the parent process calls wait() to delay its execution until the child finishes executing.

When the child is done, wait() returns to the parent.

exec

However, sometimes you want to run a different program; exec() does just that.

What it does: given the name of an executable, and some arguments, it takes the code from that executable and overwrites its current code segment with it; the heap and stack and other parts of the memory space of the programme reinitialized.

exec

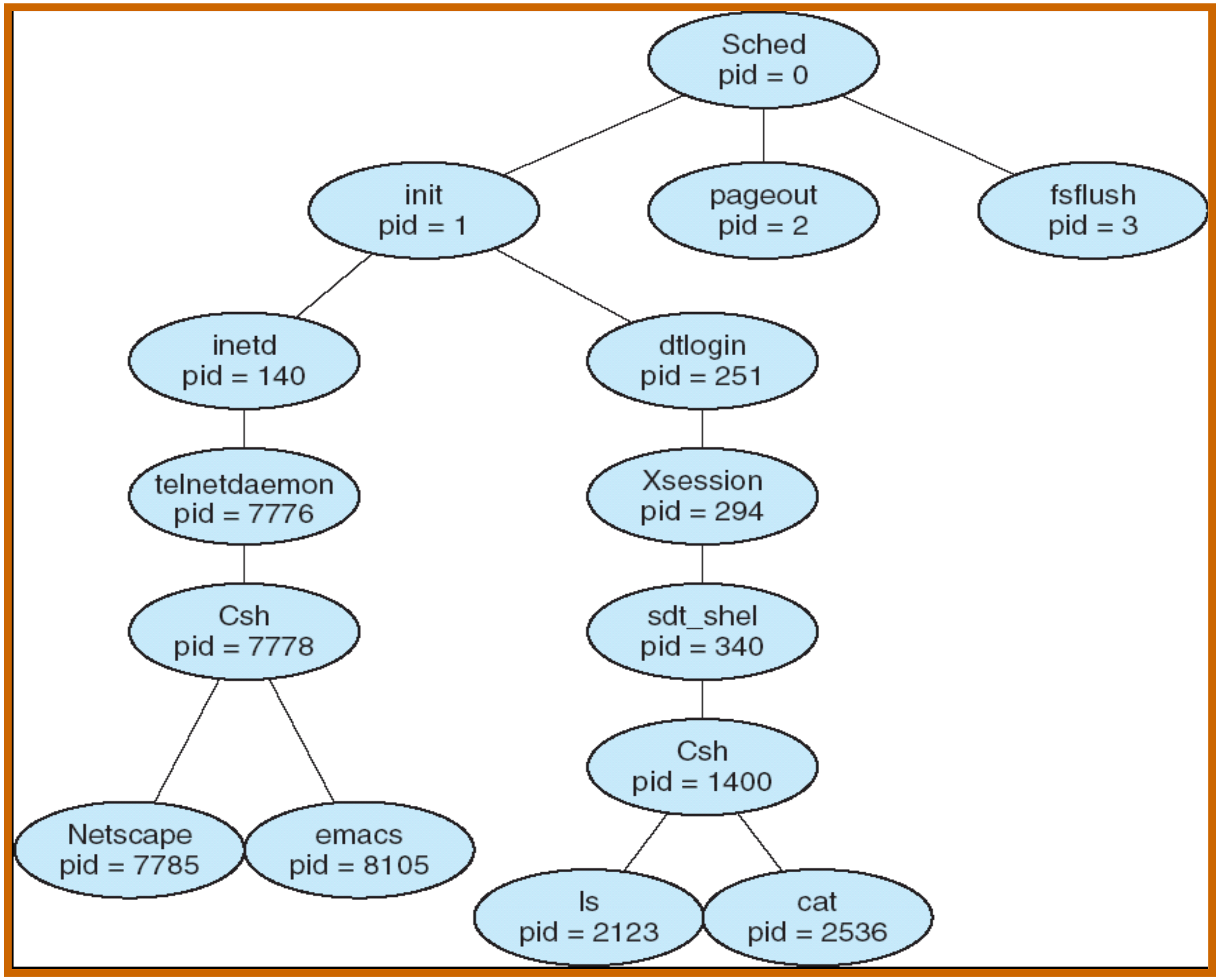
Then the OS simply runs that program, passing in any arguments as the argv of that process. Thus, it does not create a new process; rather, it transforms the currently running program into a different running program.

After the exec() in the child, it is almost as if p3.c never ran; a successful call to exec() never returns.

UNIX Shell

```
main()
{
    while(read from stdin) /* command */
    {
        if (fork()==0)
        {
            execve(command);
        }
    }
}
```

A tree of processes on a typical Solaris



Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - *cascading termination*

Book

“Advanced Programming in the UNIX Environment” W. Richard Stevens and Stephen A. Rago Addison-Wesley, 1992

Restricted Operations

Direct execution has the obvious advantage of being fast; the program runs natively on the hardware CPU and thus executes as quickly as one would expect.

But running on the CPU introduces a problem:

what if the process wishes to perform some kind of **restricted** operation, such as issuing an I/O request to a disk?

Restricted Operations

A process could simply read the entire disk and thus all protections would be lost.

User mode

Introduce a new processor mode, known as **user mode**; any code that runs in user mode is restricted in what it can do.

For example, when running in user mode, a process **can’t issue** any **I/O requests**; doing so would result in the processor raising an **exception**; the OS would then likely kill the process.

Kernel mode

In contrast to user mode is **kernel mode**, which the operating system (or kernel) runs in.

In this mode, code that runs can do what it likes, including **privileged** operations such as issuing I/O requests and executing all types of restricted instructions.

Kernel mode

What should a user process do when it wishes to perform some kind of privileged operation,

such as reading from disk?

To enable this, virtually all modern hardware provides the ability for user programs to perform a **system call**.

Kernel mode

System calls allow the kernel to carefully **expose** certain **key** pieces of functionality to user programs, such as **accessing** the **file system**, **creating** and **destroying processes**, **communicating** with other processes, and **allocating** more **memory**.

Most operating systems expose a **few hundred** such operations (early Unix systems exposed a much more concise subset of around twenty calls.

Kernel mode

Assignment:

See POSIX standard for details on what modern Unix systems expose.

Kernel mode

To execute a system call, a program must execute a special **trap instruction**. This instruction simultaneously **jumps into** the **kernel** and raises the **privilege** level to **kernel mode**; once in the kernel, the system can now perform whatever privileged operations are needed (if allowed), and thus do the required work for the calling process.

Kernel mode

When **finished**, the OS calls a special **return-from-trap** instruction, which returns into the calling user program while simultaneously **reducing** the **privilege level** back to user mode.

Kernel mode

The hardware needs to be careful when executing a trap, in that it must make sure to **save** enough of the **caller’s register state** in order to be able to return correctly when the OS issues the return from-trap instruction.

Kernel mode

On x86, for example, the processor will **push** the **program counter**, **flags**, and a **few** other **registers** onto a **stack**; the **return-from-trap** will **pop** these values off the stack and resume execution of the user-mode program.

Other hardware systems use different conventions, but the basic concepts are similar across platforms.

Kernel mode

The kernel does so by setting up a **trap table** at **boot** time. When the machine boots up, it does so in privileged (kernel) mode, and thus is free to configure machine hardware as need be.

One of the first things the OS thus does is to tell the hardware what code to run when certain exceptional events occur.

For example, what code should run when a hard-disk interrupt takes place, when a keyboard interrupt occurs, or when program makes a system call?

Kernel mode

The OS informs the hardware of the **locations** of these **trap handlers**, usually with some kind of special instruction.

Once the **hardware** is informed, it remembers the location of these handlers until the machine is next rebooted, and thus the hardware knows what to do (i.e., what code to jump to) when system calls and other exceptional events take place.

Kernel mode

Being able to execute the instruction to tell the hardware **where the trap tables are** is a very powerful capability.

Thus, it is also a **privileged** operation.

If you try to execute this instruction in user mode, the kernel won’t let you, and you can probably guess what will happen.

What you might think about: what types of horrible things could you do to a system if you could install your own trap table?

Switching between Processes

If a process is running on the CPU, this by definition means the OS is not running.

If the OS is not running, how can it do anything at all?

While this sounds almost philosophical, it is a real problem: there is clearly no way for the OS to take an action if it is not running on the CPU.

Switching between Processes

A **timer** device can be programmed to **raise** an **interrupt** every so many milliseconds; when the interrupt is raised, the **currently** running process is **halted**, and a preconfigured **interrupt handler** in the OS runs.

At this point, the OS has **regained control** of the CPU, and thus can do what it pleases: stop the current process from running, and start a new one running.

Switching between Processes

The OS must inform the hardware of which code to run when the timer interrupt occurs; thus, at boot time, the OS does exactly that.

Second, also during the **boot sequence**, the OS must **start** the **timer**, which is of course a privileged operation. Once the timer has begun, the OS can thus feel safe in that control will eventually be returned to it, and thus the OS is free to run user programs.

The timer can also be turned off (also a privileged operation) !!! (when is it turned off?)

Switching between Processes

Note that the hardware has some responsibility when an interrupt occurs, in particular to save enough of the state of the program that was running when the interrupt occurred such that a subsequent return-from-trap instruction will be able to resume the running program correctly.

Switching between Processes

This set of actions is quite **similar** to the behavior of the hardware during an explicit **system-call trap** into the kernel, with various registers thus getting saved (e.g., onto a kernel stack) and thus easily restored by the return-from-trap instruction.

Saving and Restoring Context

Now that the OS has regained control, whether cooperatively via a system call, or more forcefully via a timer interrupt, a decision has to be made: **whether** to continue running the **currently-running** process, or switch to a **different** one.

This decision is made by a part of the operating system known as the **scheduler**.

Saving and Restoring Context

If the decision is made to switch, the OS then executes a low-level piece of code which is referred as a **context switch**.

Saving and Restoring Context

A context switch is conceptually simple: all the OS has to do is save a few register values for the currently-executing process and restore a few for the soon-to-be-executing process.

By doing so, the OS thus ensures that when the return-from-trap instruction is finally executed, instead of returning to the process that was running, the system resumes execution of another process.

Saving and Restoring Context

To save the context of the currently-running process, the OS will execute some low-level assembly code to save the general purpose

registers, PC, as well as the kernel stack pointer of the currently running process, and then restore said registers, PC, and switch to the kernel stack for the soon-to-be-executing process.

Saving and Restoring Context

By **switching stacks**, the kernel enters the call to the switch code in the context of one process (the one that was interrupted) and returns in the **context** of another (the soon-to-be-executing one).

When the OS then finally **executes** a **return-from-trap** instruction, the soon-to-be executing process becomes the **currently-running** process. And thus the context switch is complete.

How Long Context Switches take

A natural question you might have is: how long does something like a context switch take? Or even a system call?

For those of you that are curious, there is a tool called **lmbench** that measures exactly those things, as well as a few other performance measures that might be relevant.

How Long Context Switches take

Results have improved quite a bit over time, roughly tracking processor performance.

For example, in 1996 running Linux 1.3.37 on a 200-MHz P6 CPU, system calls took roughly **4 microseconds**, and a context switch roughly 6 microseconds.

Modern systems perform almost an order of magnitude better, with sub-microsecond results on systems with 2- or 3-GHz processors.

Measurement

gettimeofday

rdtsc

Measurement

lmbench

1. Run two processes on a single CPU.
2. Set up two pipes between them.
3. The first process then issues a write to the first pipe, and waits for a read on the second; upon seeing the first process waiting for something to read from the second pipe, the OS puts the first process in the blocked state, and

lmbench Measurement

4. Switches to the other process, which reads from the first pipe and then writes to the second.

5. When the second process tries to read from the first pipe again, it blocks, and thus the back-and-forth cycle of communication continues.

6. By measuring the cost of communicating like this repeatedly, *lmbench* can make a

good estimate of the cost of a context switch.

Measurement

Bind a process to a particular processor; on Linux, for example, the **sched_setaffinity**() call is what you’re looking for.

By ensuring both processes are on the same processor, you are making sure to measure the cost of the OS stopping one process and restoring another on the same CPU.

End