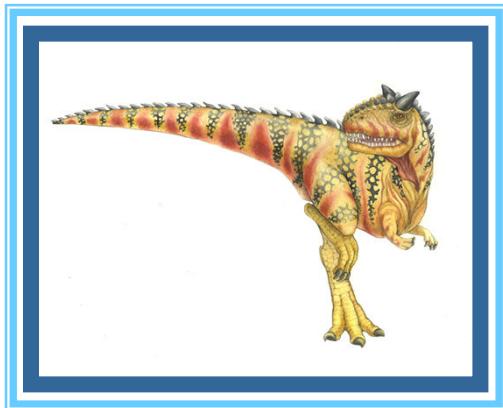


Tema 2: Procesos



- Chapter 3: Process Concept
- Chapter 4: Multithreaded Programming
- Chapter 5: Process Scheduling

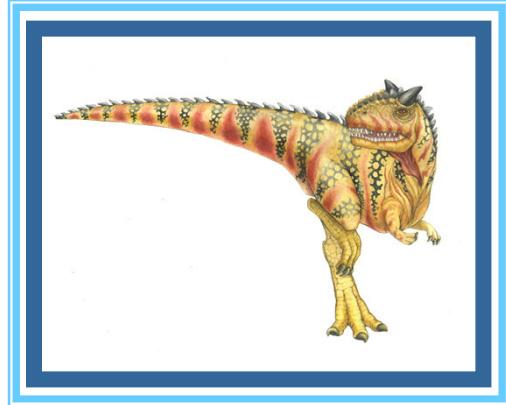
Silberschatz, Galvin and Gagne ©2009

Rudowsky ©2005

Walpole ©2010

Kubiatowicz ©20010

Chapter 3: Process-Concept



Silberschatz, Galvin and Gagne ©2009

Rudowsky ©2005

Walpole ©2010

Kubiatowicz ©20010



Chapter 3. Contents

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication





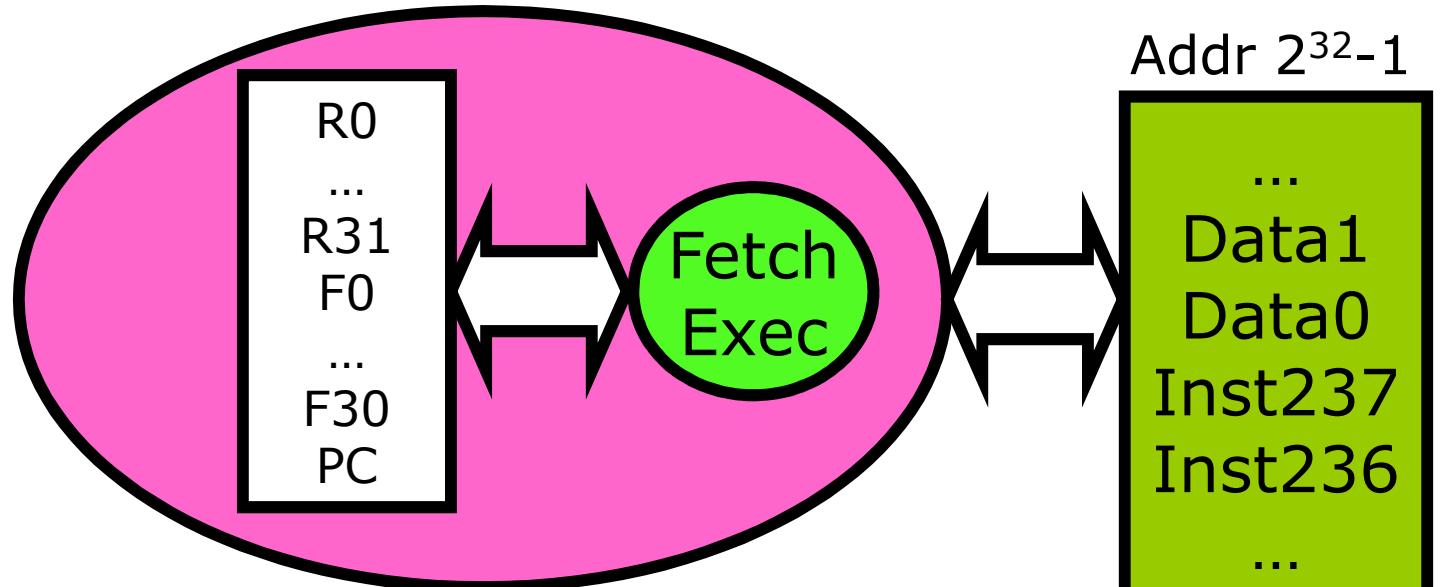
Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication





What happens during execution?



- Execution sequence:
 - Fetch Instruction at PC
 - Decode
 - Execute (possibly using registers)
 - Write results to registers/mem
 - PC = Next Instruction(PC)
 - Repeat

Addr $2^{32}-1$

...
Data1
Data0
Inst237
Inst236
...

Inst5
Inst4
Inst3
Inst2
Inst1
Inst0

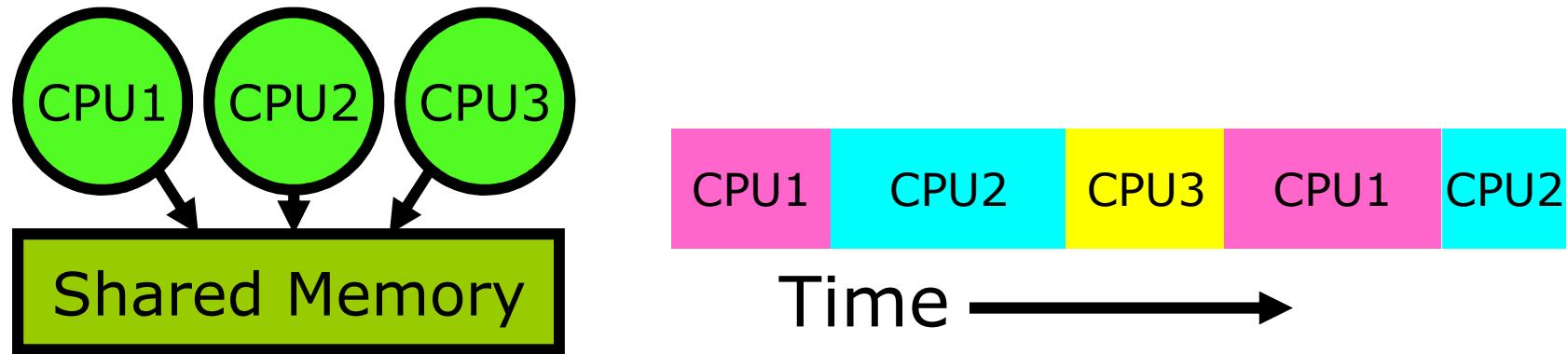
PC
PC
PC
PC
PC

Addr 0





Concurrent execution (single CPU)

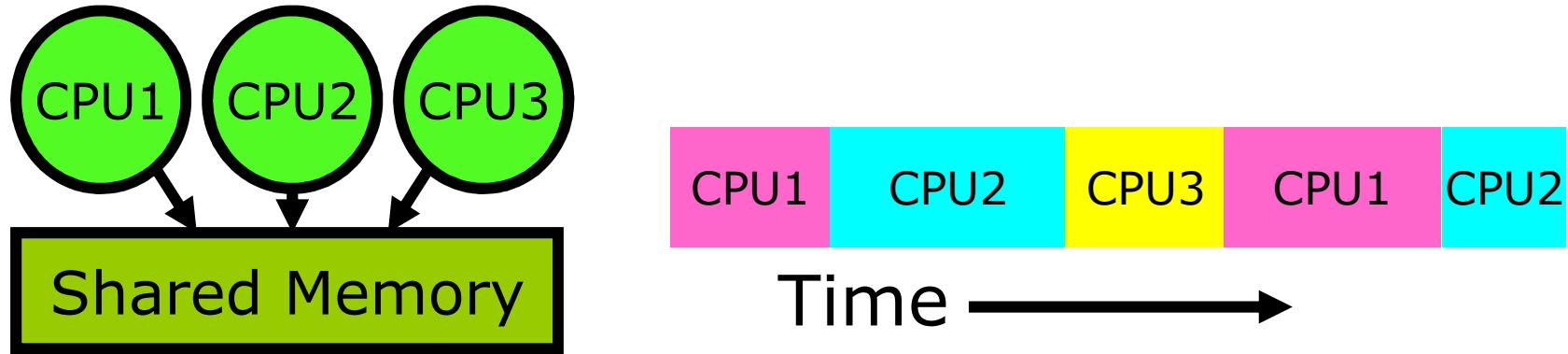


- The basic problem of concurrency involves resources:
 - Hardware: *single* CPU, single DRAM, single I/O devices
 - Multiprogramming API: users think they have exclusive access to shared resources
- How do we provide the illusion of multiple processors?: Multiplex in time!
- Basic Idea: Use Virtual Machine abstraction
 - Decompose hard problem into simpler ones
 - Abstract the notion of an executing program: **process**





Concurrent execution (single CPU)



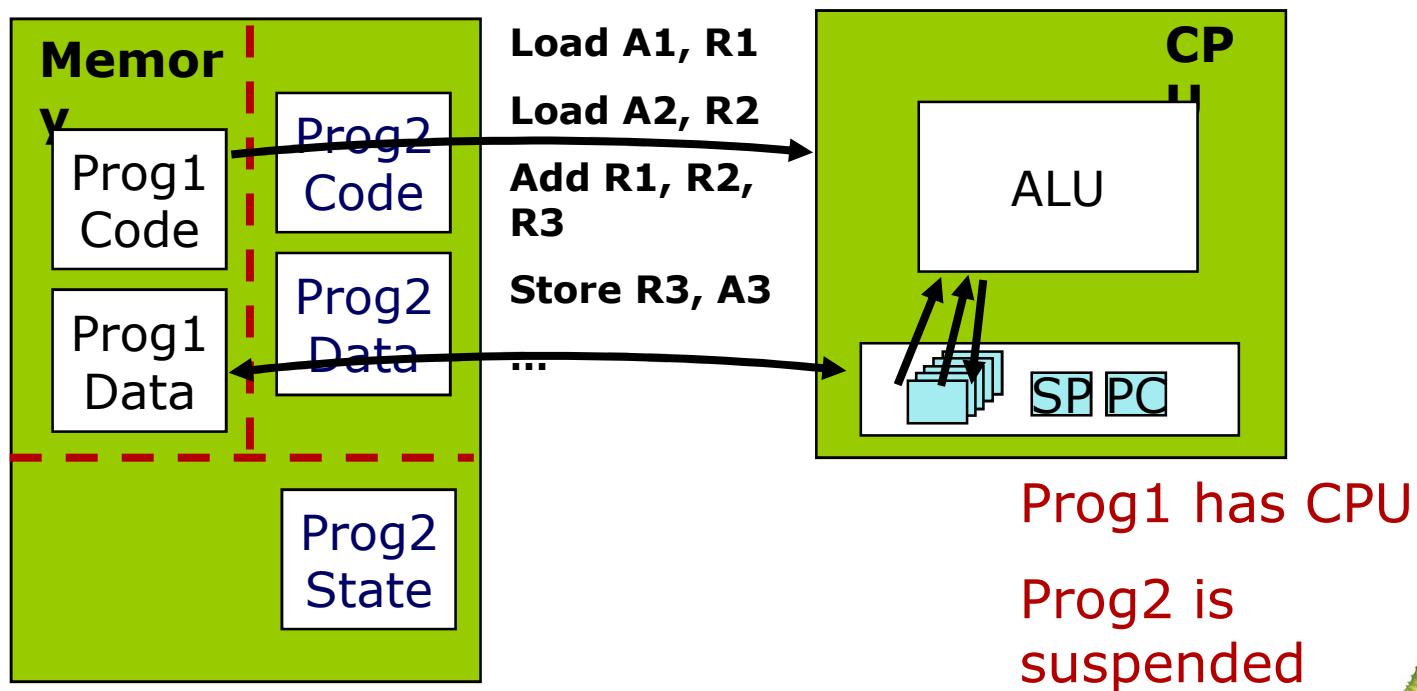
- Then, worry about multiplexing these abstract machines
 - Each virtual “CPU” ([process](#)) needs a structure to hold:
 - ▶ Program Counter (PC), Stack Pointer (SP)
 - ▶ Registers (Integer, Floating point, others...?)
 - How switch from one CPU to the next?
 - ▶ Save PC, SP, and registers in current state block
 - ▶ Load PC, SP, and registers from new state block
 - What triggers switch?
 - ▶ Timer, voluntary yield, I/O, other things





Switching among multiple processes

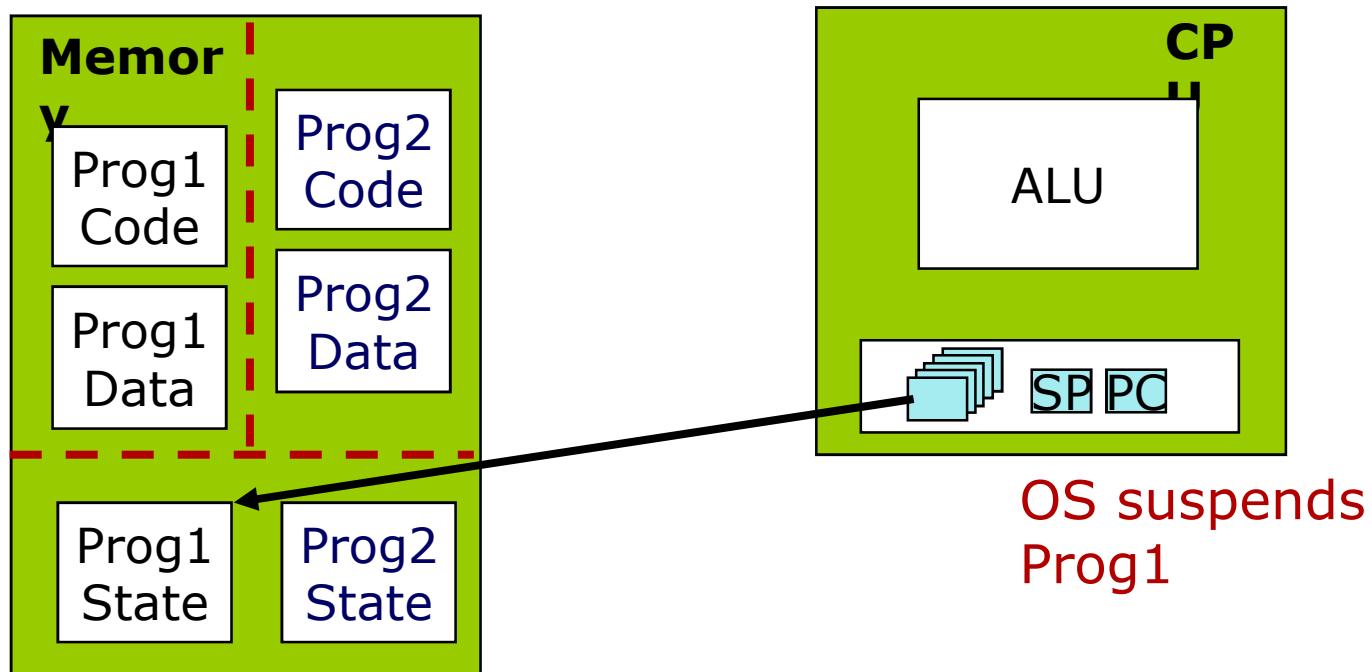
- Program instructions operate on operands in memory and (temporarily) in registers





Switching among multiple processes

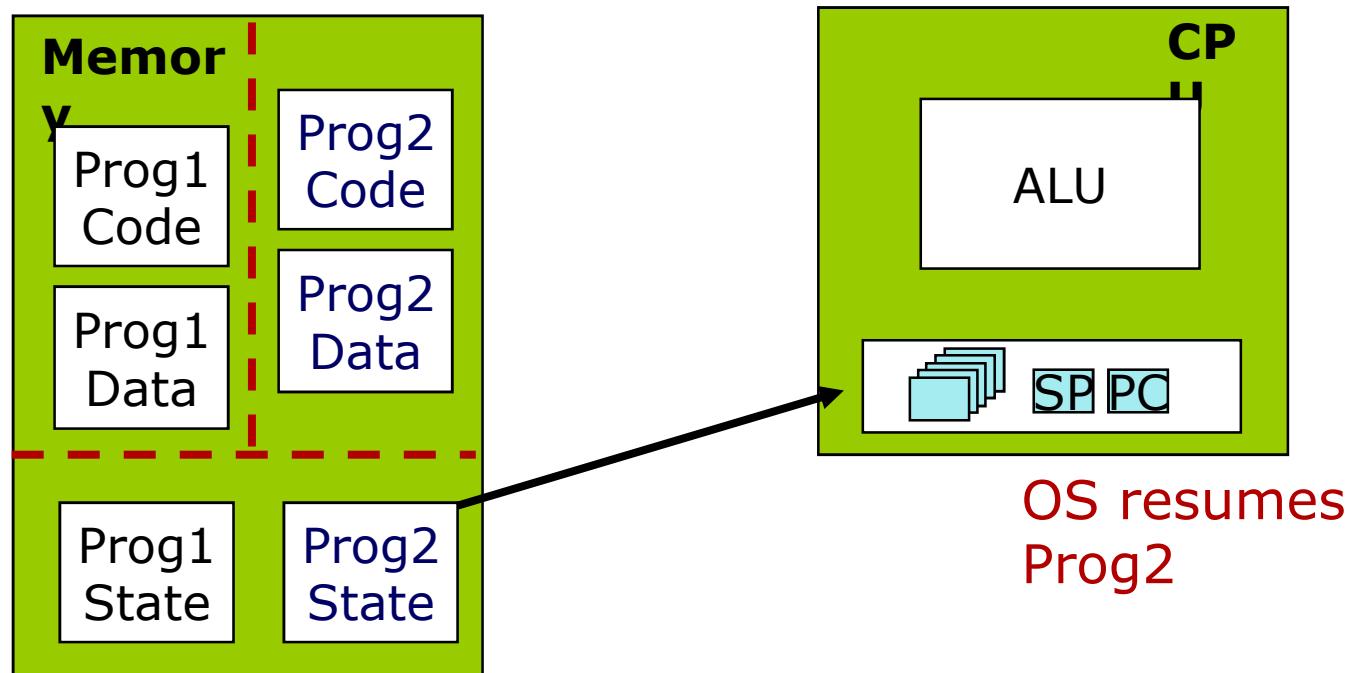
- Saving all the information about a process allows a process to be *temporarily suspended* and later *resumed* from the same point





Switching among multiple processes

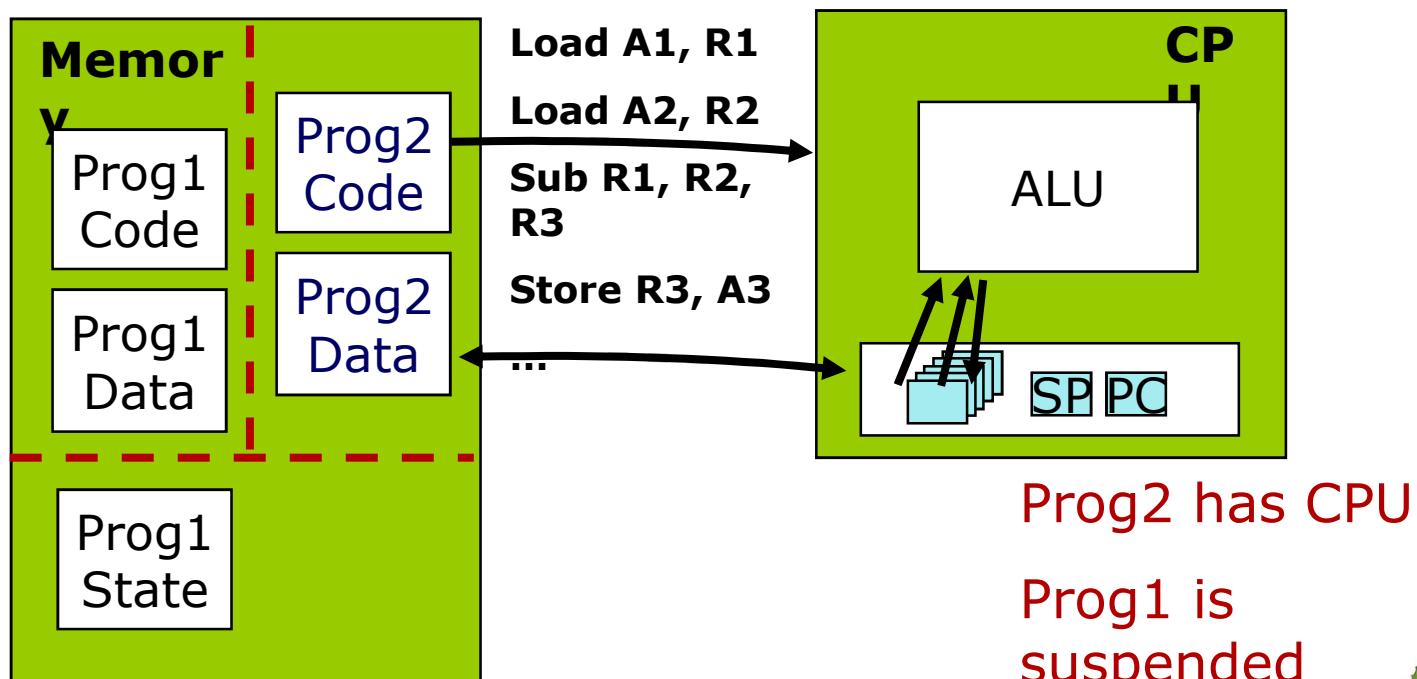
- Saving all the information about a process allows a process to be *temporarily suspended* and later *resumed*





Switching among multiple processes

- Program instructions operate on operands in memory and in registers





Process Concept

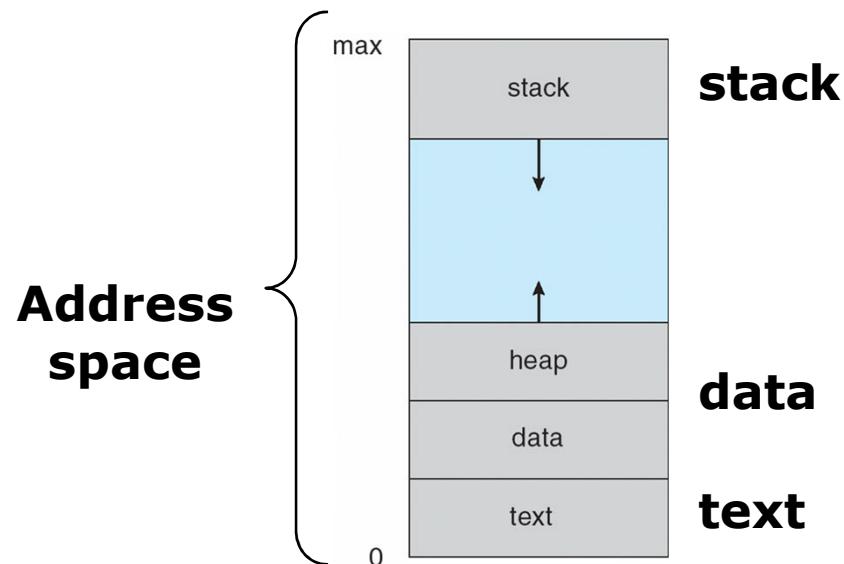
- An operating system executes a variety of programs:
 - Batch system – jobs
 - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
 - Program
 - ▶ description of how to perform an activity
 - ▶ instructions and static data values
 - Process
 - ▶ a snapshot of a program in execution
 - ▶ memory (program instructions, static and dynamic data values)
 - ▶ CPU state (registers, PC, SP, etc)
 - ▶ operating system state (open files, accounting statistics etc)





Process address space

- Each process runs in its own virtual memory *address space* that consists of:
 - *Stack space* – used for function and system calls
 - *Data space* – variables (both static and dynamic allocation)
 - *Text* – the program code (usually read only)



- Invoking the same program multiple times results in the creation of multiple distinct address spaces





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





The Basic Problem of Concurrency

- The basic problem of concurrency involves resources:
 - Hardware: single CPU, single DRAM, single I/O devices
 - Multiprogramming API: users think they have exclusive access to shared resources
- OS Has to coordinate all activity
 - Multiple users, I/O interrupts, ...
 - How can it keep all these things straight?





How to protect process from one another?

- Need four important things:
 1. Dual mode operation
 2. Protection of memory
 - ▶ Every task does not have access to all memory
 3. Protection of I/O devices
 - ▶ Every task does not have access to every device
 4. Protection of Access to Processor:
Preemptive switching from task to task
 - ▶ Use of timer
 - ▶ Must not be possible to disable timer from usercode





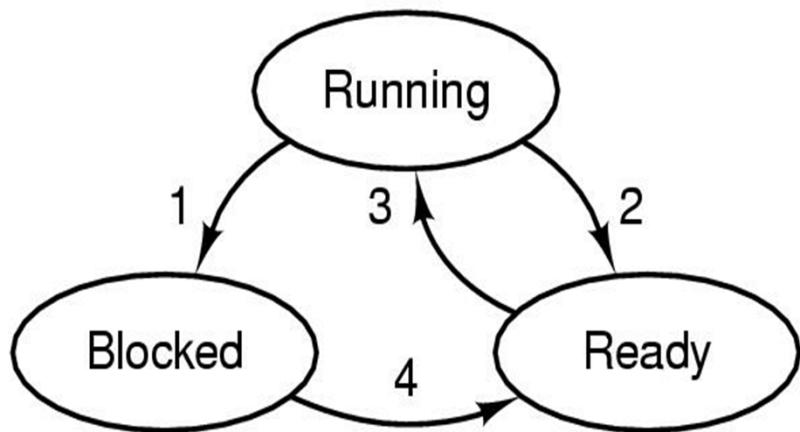
Chapter 3. Contents

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication





Process states



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

- Possible process states
 - running
 - blocked
 - ready





Process Tracking

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

(a) Trace of Process A

(b) Trace of Process B

(c) Trace of Process C

5000 = Starting address of program of Process A

8000 = Starting address of program of Process B

12000 = Starting address of program of Process C





Process Tracking

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

100 = Starting address of dispatcher program

shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;

Figure 3.3 Combined Trace of Processes of Figure 3.1





Process Tracking

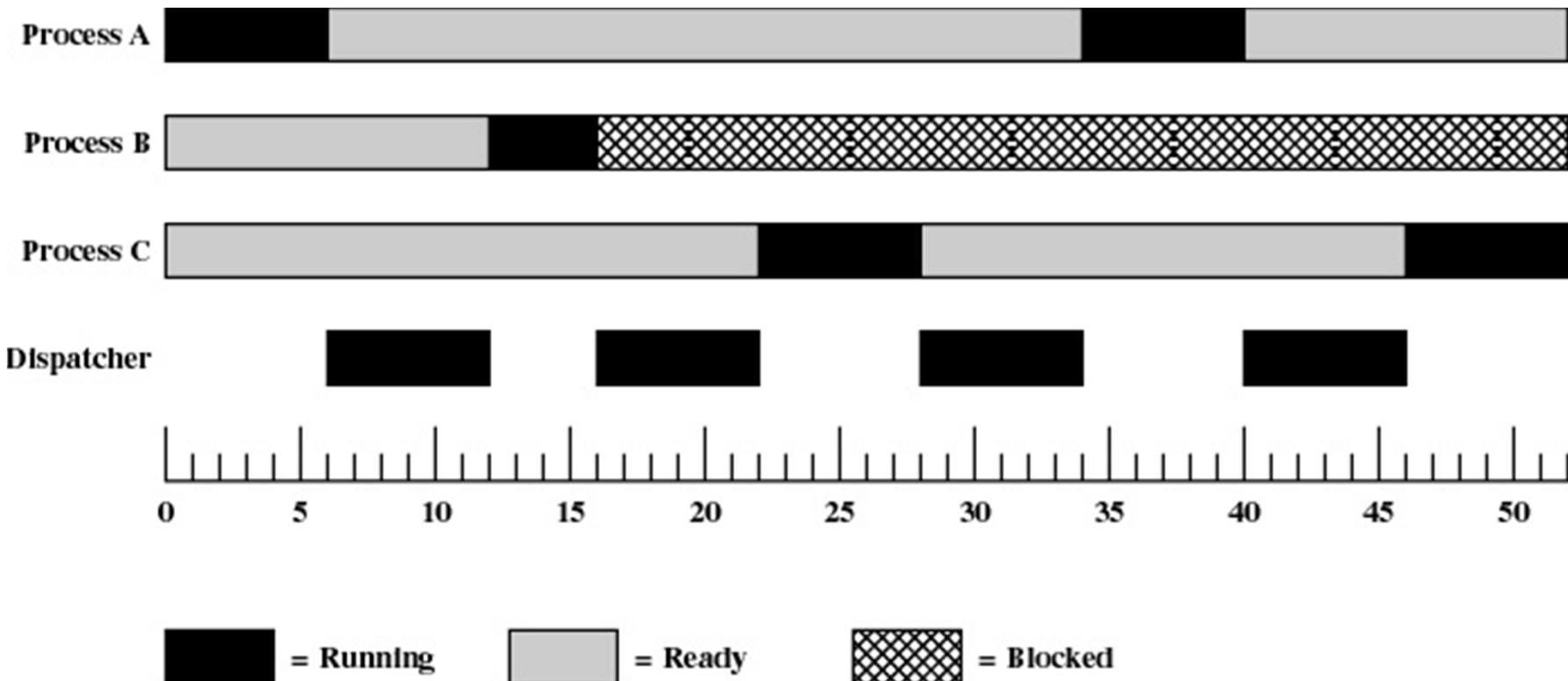


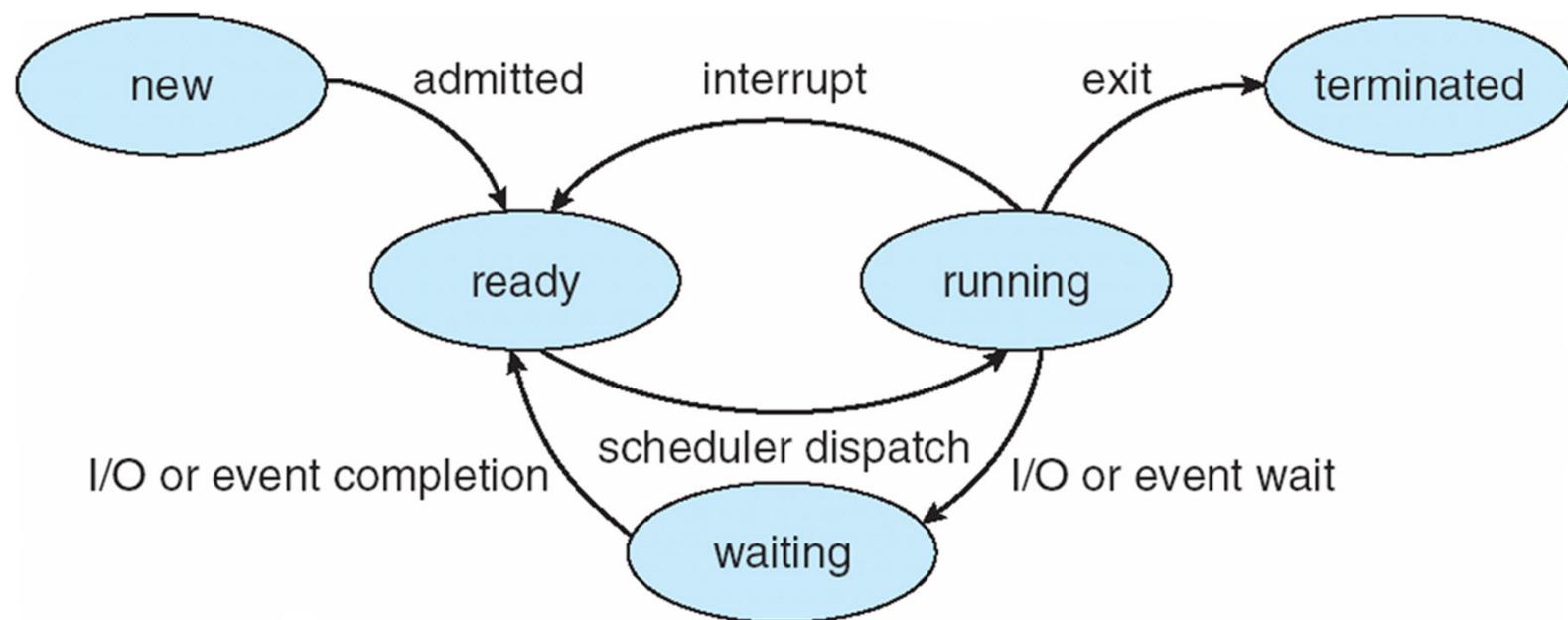
Figure 3.6 Process States for Trace of Figure 3.3





Process State

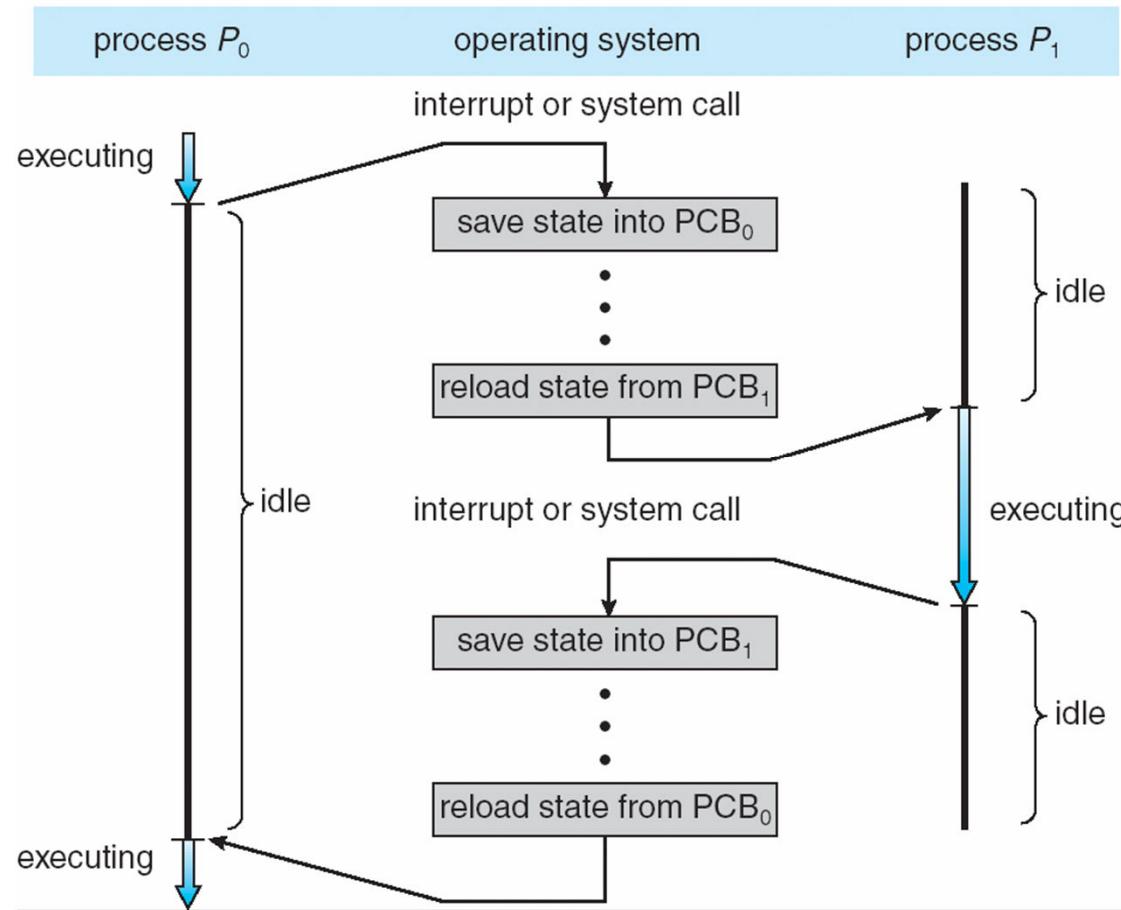
- As a process executes, it changes *state*
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur
 - **ready**: The process is waiting to be assigned to a processor
 - **terminated**: The process has finished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





CPU Switch From Process to Process

- This PCB is saved when a process is removed from the CPU and another process takes its place (context switch).





Context Switch

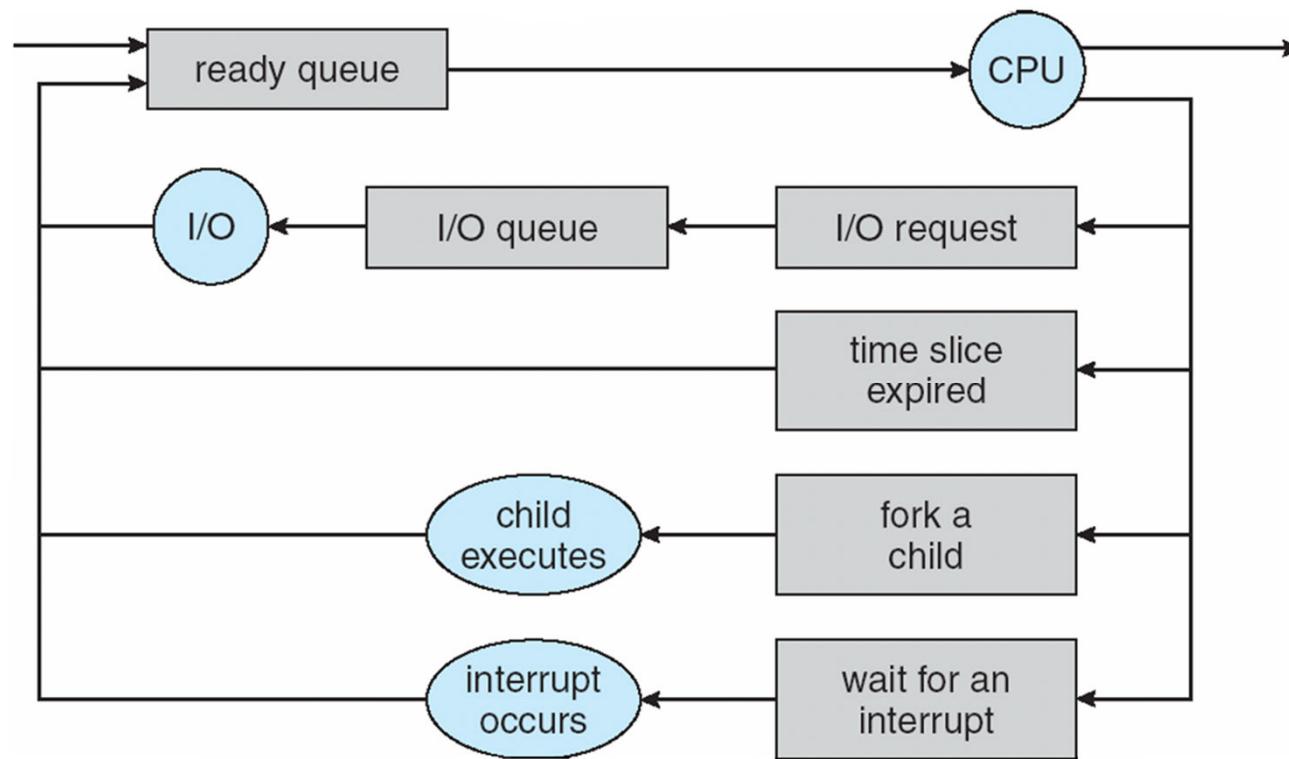
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support.
 - Varies from 1 to 1000 microseconds





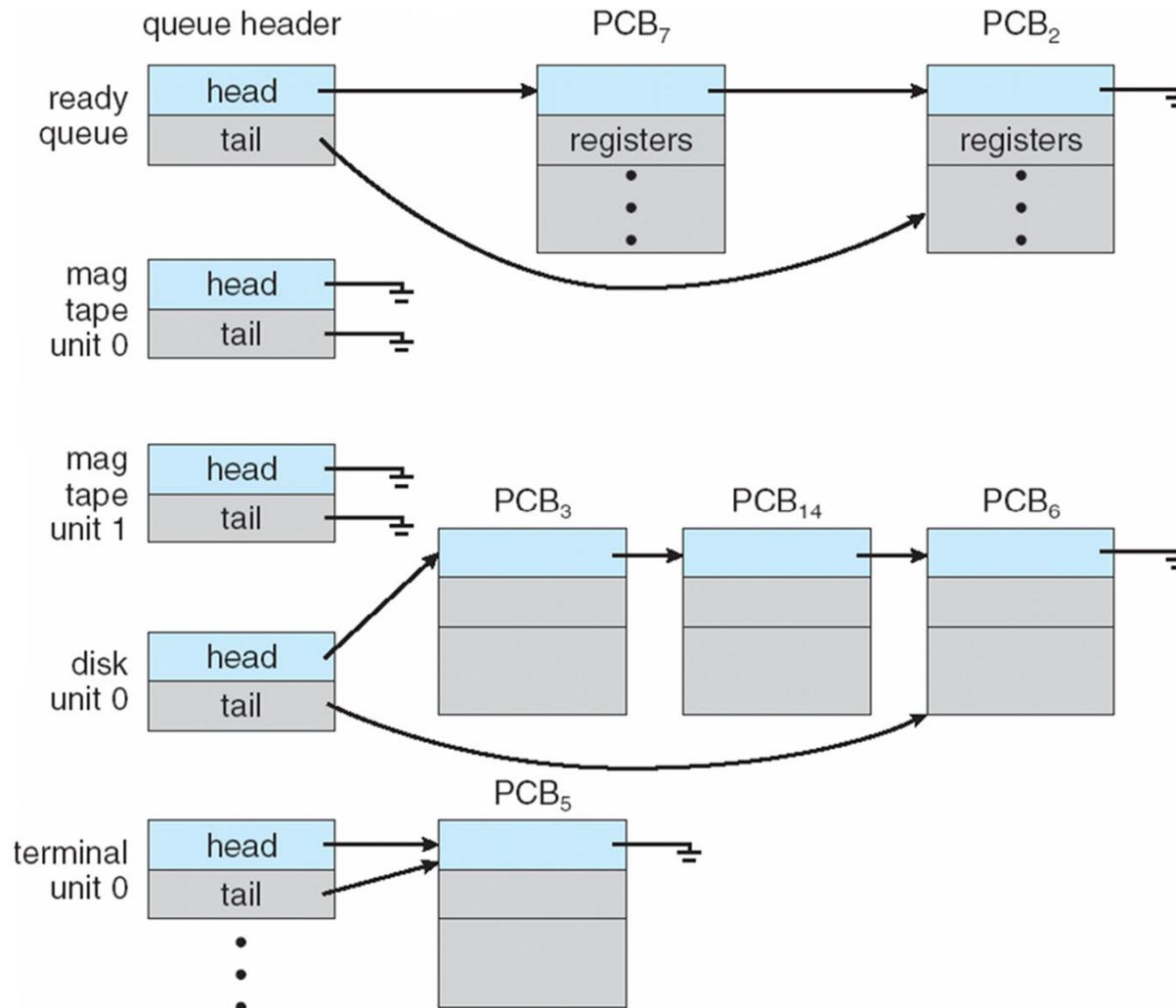
Process Scheduling Queues

- **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





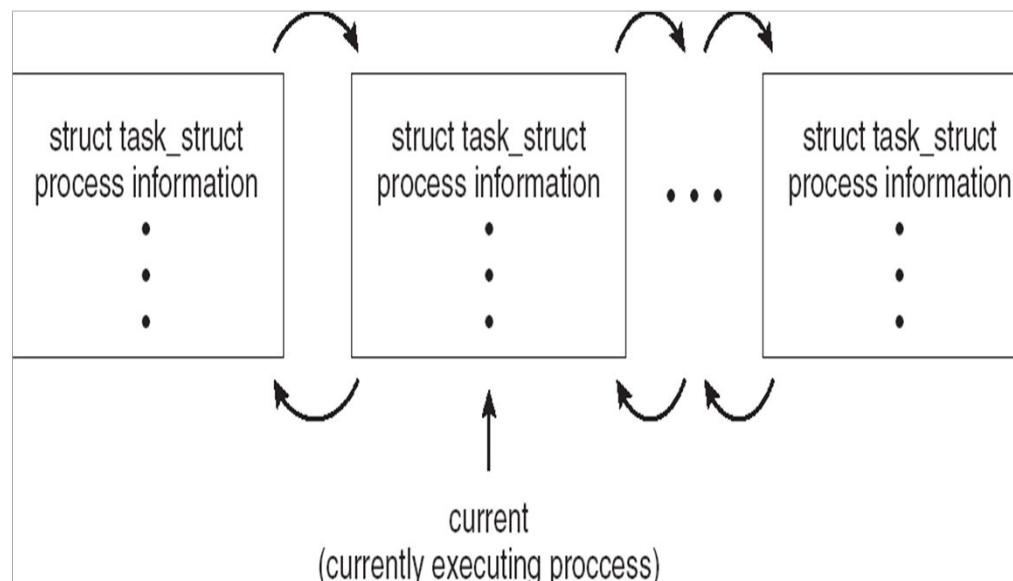
Ready Queue And Various I/O Device Queues





Active processes in Linux

- The process control block in Linux is represented by the C structure ***task_struct***
 - State of the process
 - Scheduling and memory management information
 - List of open files
 - Pointers to the process' parent and any of its children
- All active processes are represented using a doubly linked list of *task_struct* and the kernel maintains a pointer to the process currently executing





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





Schedulers (Cont)

- Short-term scheduler is invoked very frequently (once every 100 milliseconds) ⇒ must be fast or else waste too much time scheduling and not executing
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow).
 - The long-term scheduler controls the *degree of multiprogramming - the number of processes in memory.*
- Long term scheduler should select a good process mix of processes to better the system performance
 - **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.





Schedulers (Cont)

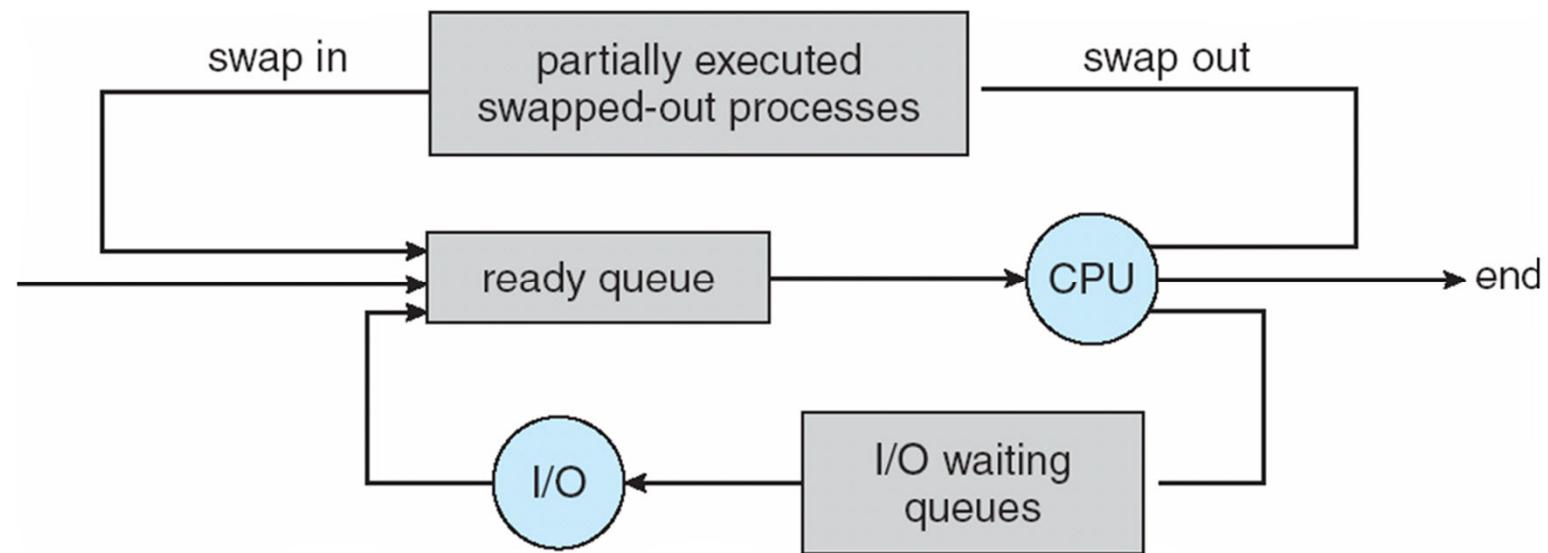
- Windows and UNIX put every new process in memory for the short term scheduler
 - With few users, little chance of running out of memory
 - If performance is bad users may just quit and solve scheduling problem





Addition of Medium Term Scheduling

- Time sharing OSs may introduce a medium term scheduler
- Removes processes from memory (and thus CPU contention) to reduce the degree of multiprogramming – **swapping**
- Swapping may be needed to improve the process mix or to free up memory if it has become overcommitted





Chapter 3. Contents

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication





How do processes get created?

Principal events that cause process creation

- System initialization
- Initiation of a batch job
- User request to create a new process
- Execution of a process creation system call from another process





Process Creation

- Parent process create **children** processes, which, in turn create other processes, forming a tree of processes
 - special system calls for communicating with and waiting for child processes
 - each process is assigned a unique identifying number or process ID (PID)
- Child processes can create their own child processes
 - Forms a hierarchy
 - UNIX calls this a "process group"
 - Windows has no concept of process hierarchy
 - ▶ all processes are created equal
- Resource sharing
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources





Process Creation (Cont)

■ Execution

- Parent and children execute concurrently
- Parent waits until children terminate

■ Address space

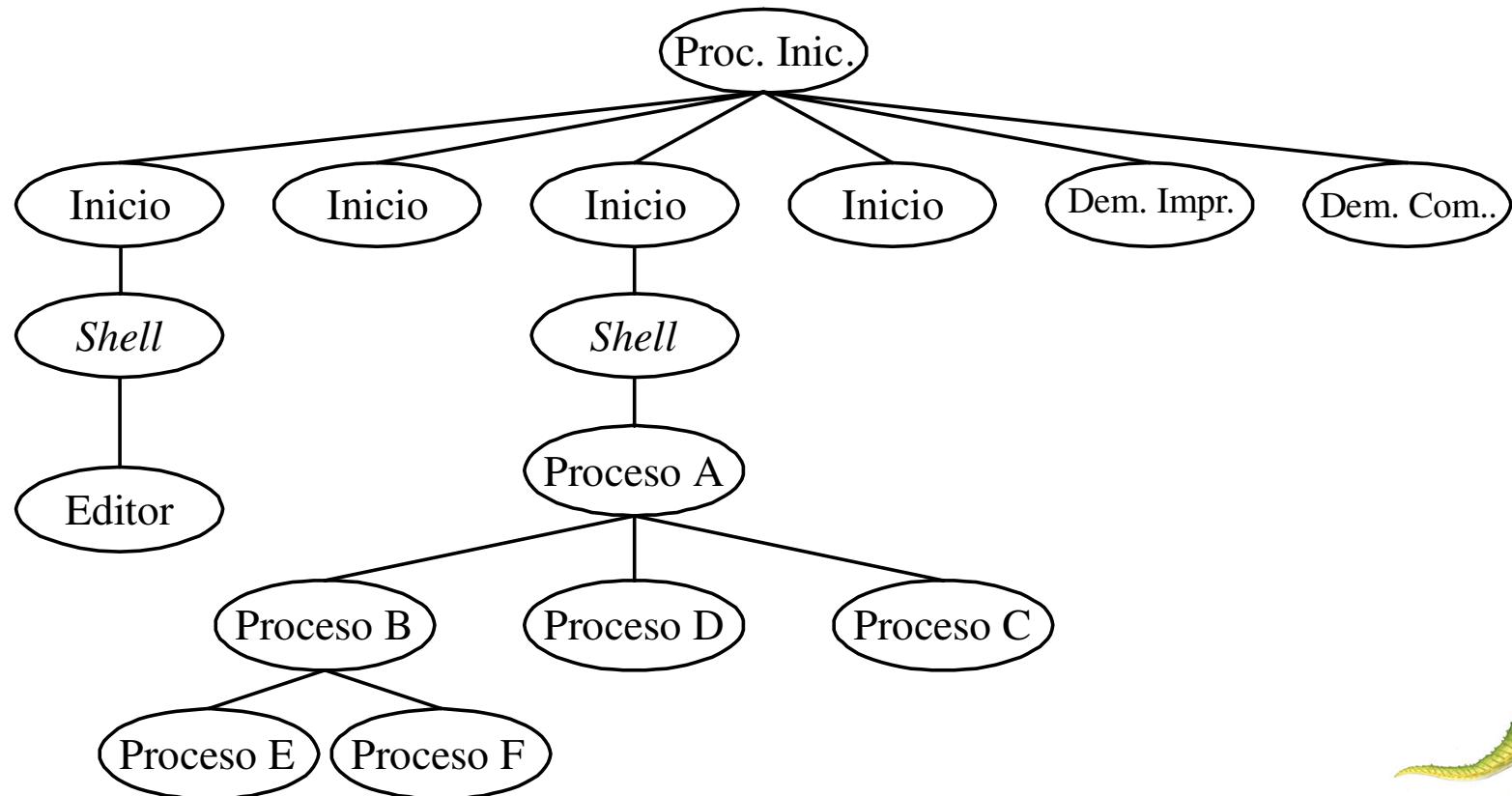
- Child duplicate of parent (UNIX)
 - ▶ Child has copy of parent's address space
 - Enables easy communication between the two
 - The child process' memory space is replaced with a new program which is then executed. Parent can wait for child to complete or create more processes
- Child has a program loaded into it directly (DEC VMS)
- Windows NT supports both models





A tree of processes in Unix

- The first process is Init (padre del resto)
- Init creates login daemons
- Login becomes a shell
- Using the shell user spawns new processes





Process creation in UNIX

■ Process creation

- **fork()** system call creates new process which has a copy of the address space of the original process and returns in both processes (parent and child), but with a different return value
 - ▶ Simplifies parent-child communication
 - ▶ Both processes continue execution
- **execvp()** system call used after a **fork()** to replace the process' address space with a new program.
 - ▶ Loads a binary file into memory and starts execution
 - ▶ Parent can then create more children processes or issue a **wait()** system call to move itself off the ready queue until the child completes





Process creation in UNIX

- All processes have a unique process id
 - *getpid()*, *getppid()* system calls allow processes to get their information

- Process termination, signaling
 - *signal()*, *kill()* system calls allow a process to be terminated or have specific signals sent to it





Example: process creation in UNIX

csh (pid = 22)

```
...
pid = fork()
if (pid == 0) {
    // child...
    ...
    exec("ls");
}
else {
    // parent
    wait();
}
...
```





Process creation in UNIX example

csh (pid = 22)

```
...
pid = fork()
if (pid == 0) {
    // child...
    ...
    exec("ls");
}
else {
    // parent
    wait();
}
...
```

csh (pid = 24)

```
...
pid = fork()
if (pid == 0) {
    // child...
    ...
    exec("ls");
}
else {
    // parent
    wait();
}
...
```





Process creation in UNIX example

csh (pid = 22)

```
...
pid = fork()
if (pid == 0) {
    // child...
    ...
    exec("ls"...);
}
else {
    // parent
    wait();
}
...
```

csh (pid = 24)

```
...
pid = fork()
if (pid == 0) {
    // child...
    ...
    exec("ls"...);
}
else {
    // parent
    wait();
}
...
```





Process creation in UNIX example

csh (pid = 22)

```
...
pid = fork()
if (pid == 0) {
    // child...
    ...
    exec("ls"...);
}
else {
    // parent
    wait();
}
...
```

csh (pid = 24)

```
...
pid = fork()
if (pid == 0) {
    // child...
    ...
    exec("ls"...);
}
else {
    // parent
    wait();
}
...
```





Process creation in UNIX example

csh (pid = 22)

```
...
pid = fork()
if (pid == 0) {
    // child...
    ...
    exec("ls"...);
}
else {
    // parent
    wait();
}
...
```

ls (pid = 24)

```
//ls program

main() {
    //look up dir
    ...
}
```





Process creation in Unix (fork)

- Fork creates a new process by *copying* the calling process
- The new process has its own
 - memory address space
 - ▶ Instructions (copied from parent)
 - ▶ Data (copied from parent)
 - ▶ Stack ?? (empty)
 - register set (copied from parent)
 - Process table entry in the OS





Process Termination

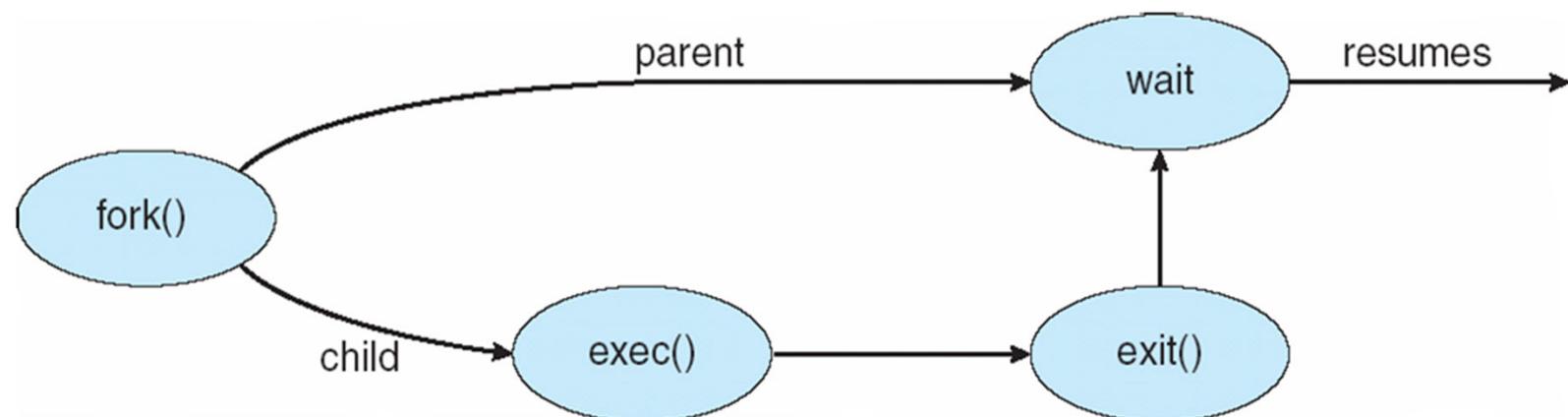
- Process executes last statement and asks the operating system to delete it by using the **exit()** system call
 - Process may return a status value to parent via **wait()**
 - Process' resources are deallocated by operating system.
- Parent may terminate execution of children processes **abort()** in UNIX
 - Child has exceeded allocated resources.
 - Task assigned to child is no longer required.
 - Parent is exiting.
 - ▶ Some operating systems do not allow child to continue if its parent terminates - **cascading termination**.





Process Termination in Unix

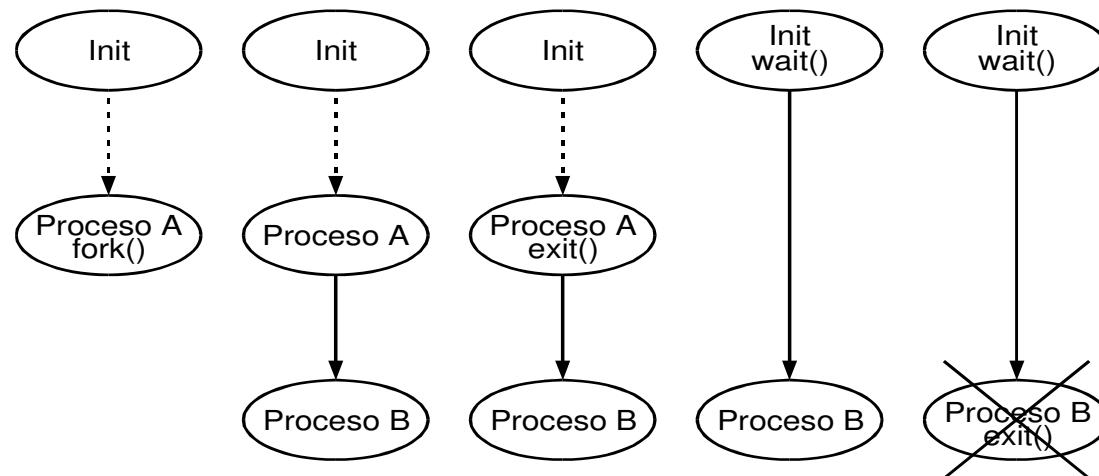
- In UNIX, a process can be terminated via the **exit** system call.
 - Parent can wait for termination of child by the **wait** system call
 - **wait** returns the process identifier of a terminated child so that the parent can tell which child has terminated



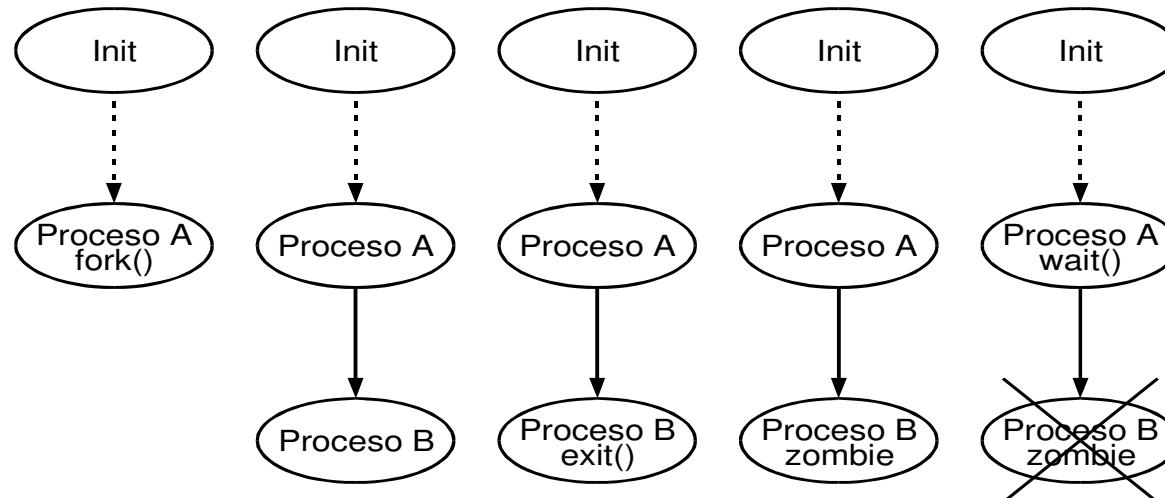


Process Termination in Unix

- If a parent terminates, all children are assigned the **init** process as their new parent



- Zombie: childrens terminate (exit) and father doesn't call to wait





Chapter 3. Contents

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication





Cooperating Processes

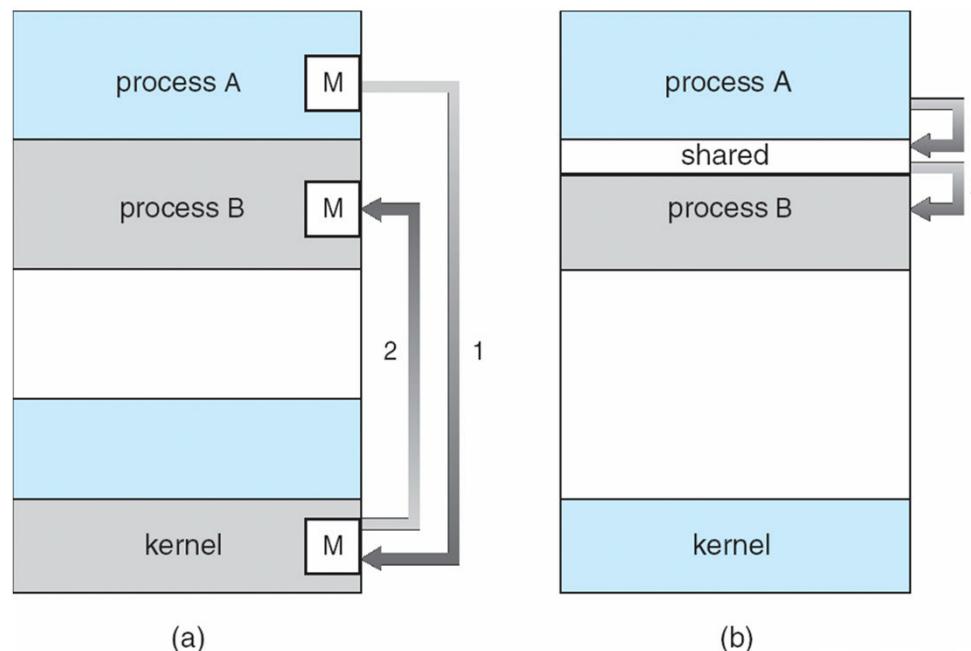
- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
 - Information sharing
 - Computation speed-up via parallel sub-tasks
 - Modularity by dividing system functions into separate processes
 - Convenience - even an individual may want to edit, print and compile in parallel
- Cooperating processes need an **interprocess communication (IPC)** mechanism to exchange data
- Two models of IPC
 - Shared memory
 - Message passing



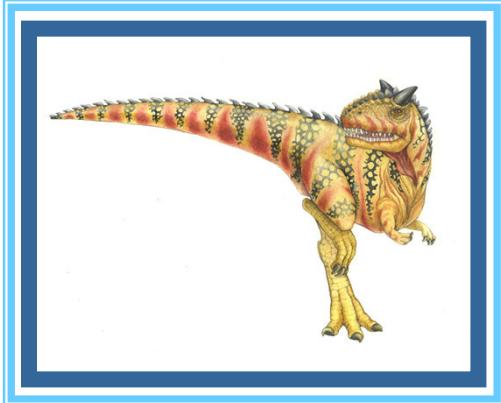


Interprocess Communication

- Message passing
 - useful for small amounts of data
 - easier to implement than shared memory
 - requires system calls and thus intervention of the kernel
- Shared memory
 - maximum speed (speed of memory) and convenience
 - system calls required only to establish the shared memory regions; further I/O does not require the kernel



Chapter 4: Multithreaded Programming



Silberschatz, Galvin and Gagne ©2009

Rudowsky ©2005

Walpole ©2010

Kubiatowicz ©20010



Chapter 4. Contents

- Overview
- Multithreading Models
- Thread Libraries
- Operating System Examples
- Signals





Objectives

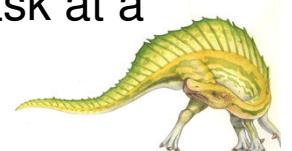
- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads





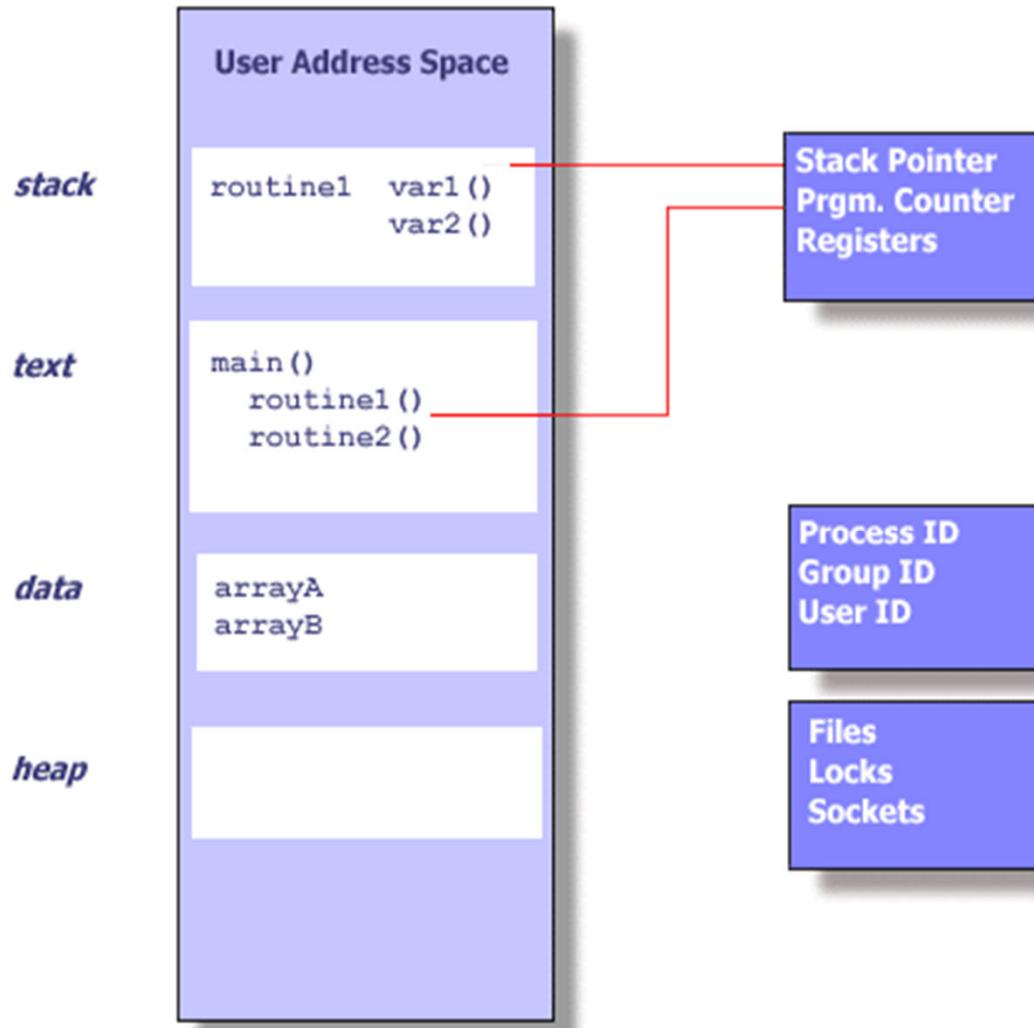
Overview

- Many software packages are **multi-threaded**
 - Web browser: one thread display images, another thread retrieves data from the network
 - Word processor: threads for displaying graphics, reading keystrokes from the user, performing spelling and grammar checking in the background
 - Web server: instead of creating a process when a request is received, which is time consuming and resource intensive, server creates a thread to service the request
- A thread is sometimes called a **lightweight process**
 - It is comprised over a thread ID, program counter, a register set and a stack
 - It shares with other threads belonging to the same process its code section, data section and other OS resources (e.g., open files)
 - A process that has multiples threads can do more than one task at a time



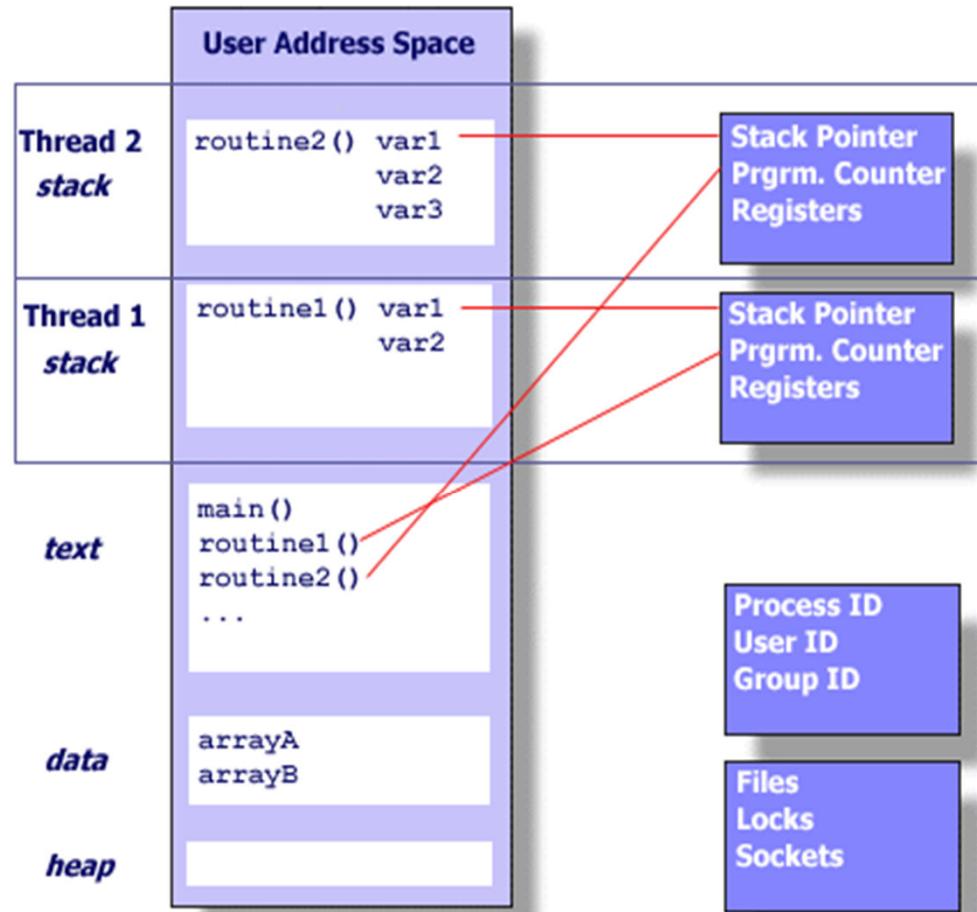


Single thread state within a process





Multiple threads in an address space





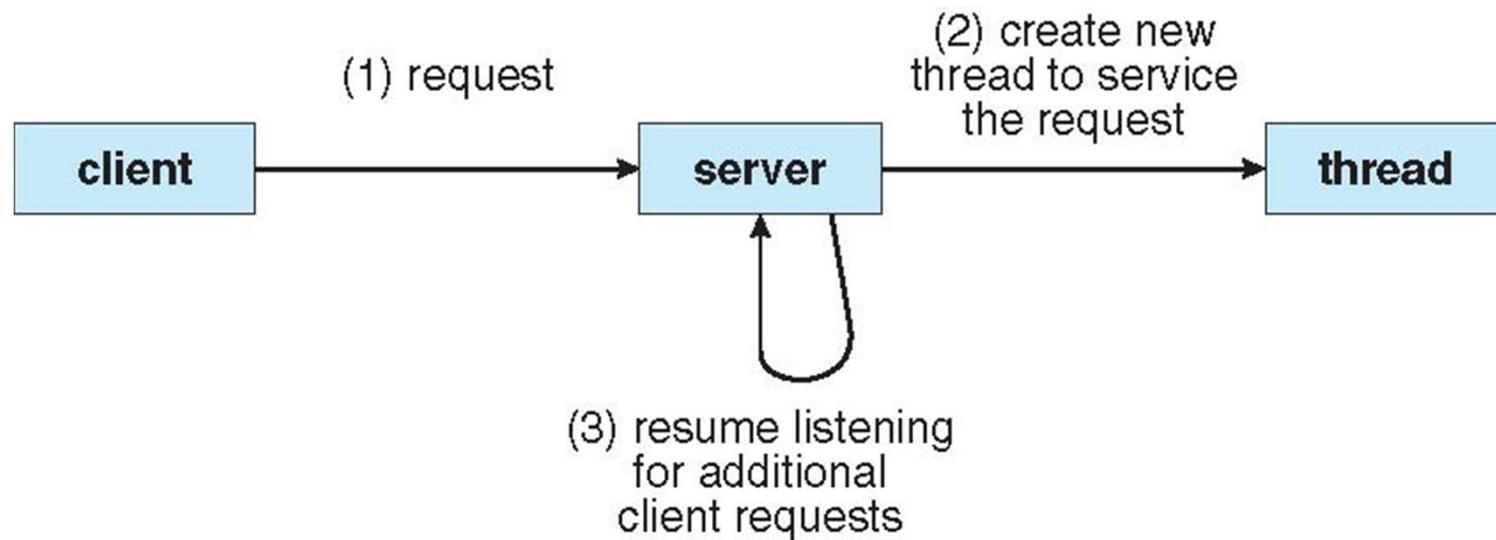
Benefits

- Responsiveness
 - One part of a program can continue running even if another part is blocked
- Resource Sharing
 - Threads of the same process share the same memory space and resources
- Economy
 - Much less time consuming to create and manage threads than processes
 - Solaris 2: creating a process is 30 times slower than creating a thread, context switching is 5 times slower
- Scalability: ***Utilization of MP Architectures***
 - Each thread can run in parallel on a different processor



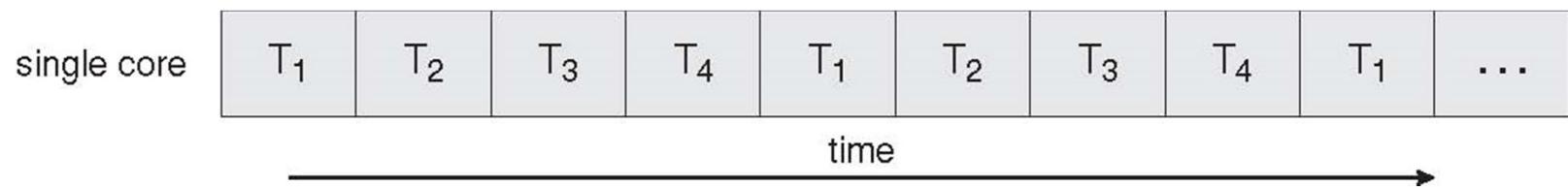


Multithreaded Server Architecture





Concurrent Execution on a Single-core System





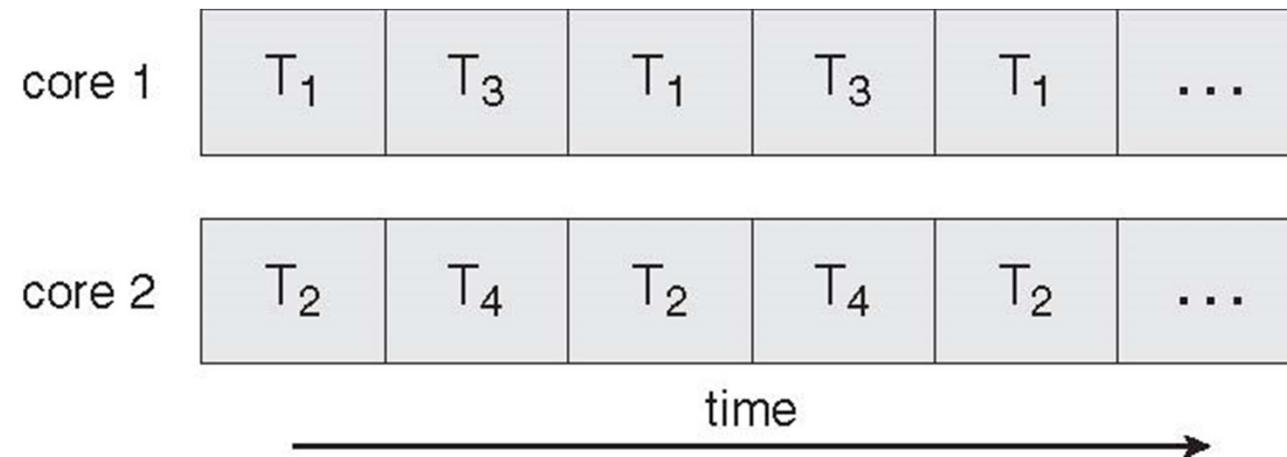
Multicore Programming

- Multicore systems putting pressure on programmers, challenges include
 - **Dividing activities**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**





Parallel Execution on a Multicore System





Chapter 4. Contents

- Overview
- Multithreading Models
- Thread Libraries
- Operating System Examples
- Signals





User Threads

- Thread management done by user-level threads library without the intervention of the kernel
 - Fast to create and manage
 - If the kernel is single threaded, any user-level thread performing a blocking system call will cause the entire process to block
- Three primary thread libraries:
 - POSIX [Pthreads](#)
 - Win32 threads
 - Java threads





Kernel Threads

- Supported by the Kernel
 - Slower to create and manage than user threads
 - If thread performs a blocking system call, the kernel can schedule another thread in the application for execution
 - In multi-processor environments, the kernel can schedule threads on multiple processors
- Examples
 - Windows XP/2000
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X





Multithreading Models

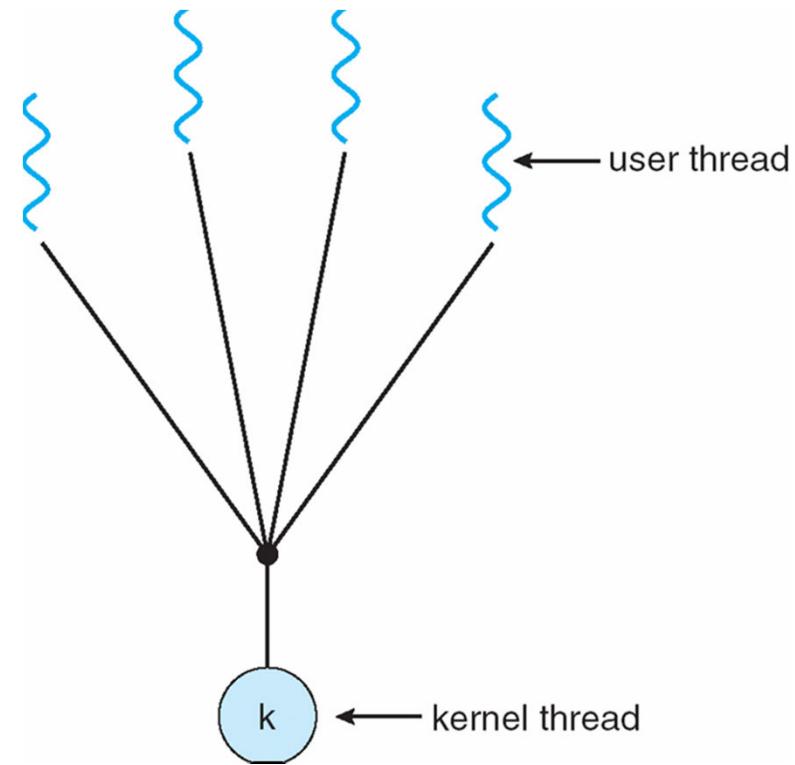
- Many-to-One
- One-to-One
- Many-to-Many





Many-to-One

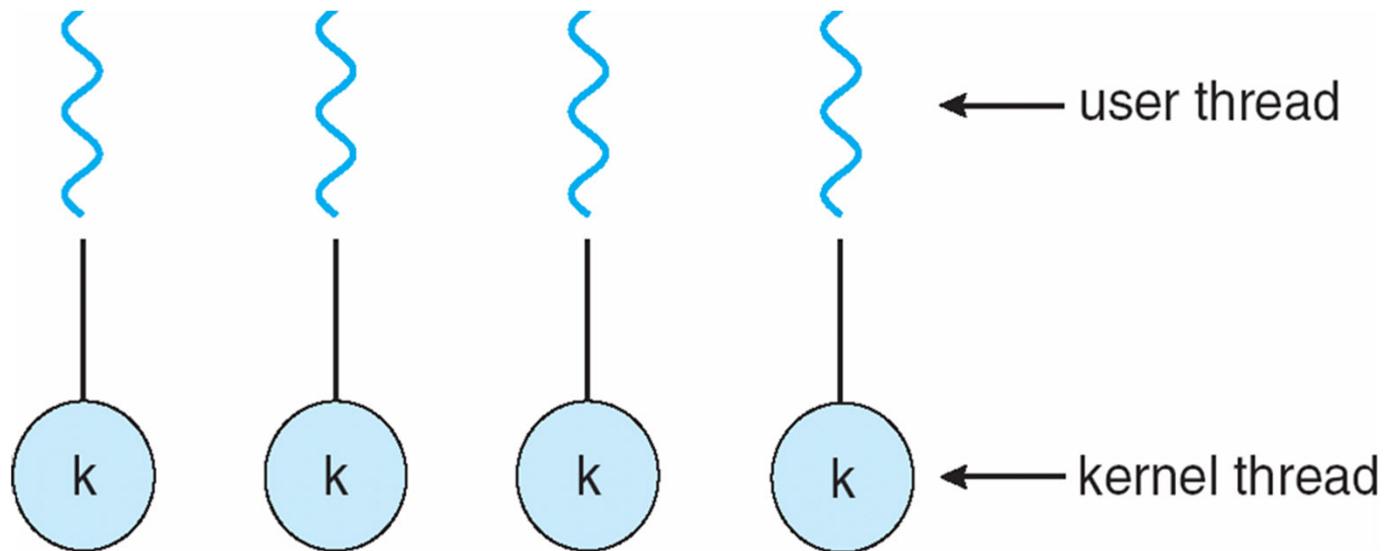
- Many user-level threads mapped to single kernel thread
 - Efficient - thread management done in user space
 - Entire process will block if a thread makes a blocking system call
 - Only one thread can access the kernel, no parallel processing in MP environment
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads





One-to-One

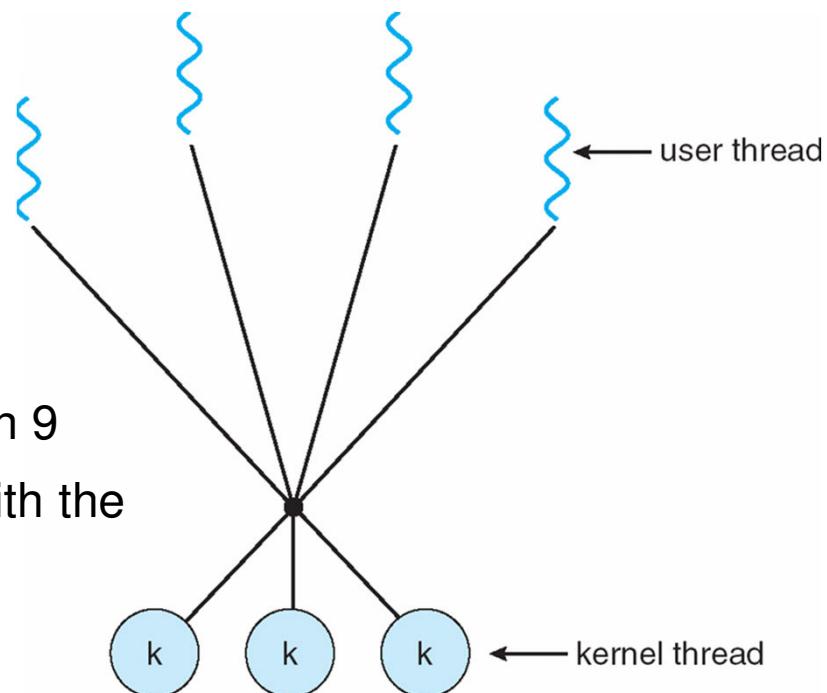
- Each user-level thread maps to kernel thread
 - Another thread can run when one thread makes a blocking call
 - Multiple threads can run in parallel on a MP machine
 - Overhead of creating a kernel thread for each user thread
 - Most implementations limit the number of threads supported
- Examples: Windows NT/XP/2000, Linux, Solaris 9 and later





Many-to-Many Model

- Allows many user level threads to be mapped to smaller or equal number of kernel threads.
- As many user threads as necessary can be created
- Corresponding kernel threads can run in parallel on a multiprocessor
- When a thread performs a blocking system call, the kernel can schedule another thread for execution



- Examples:
 - Solaris prior to version 9
 - Windows NT/2000 with the *ThreadFiber* package



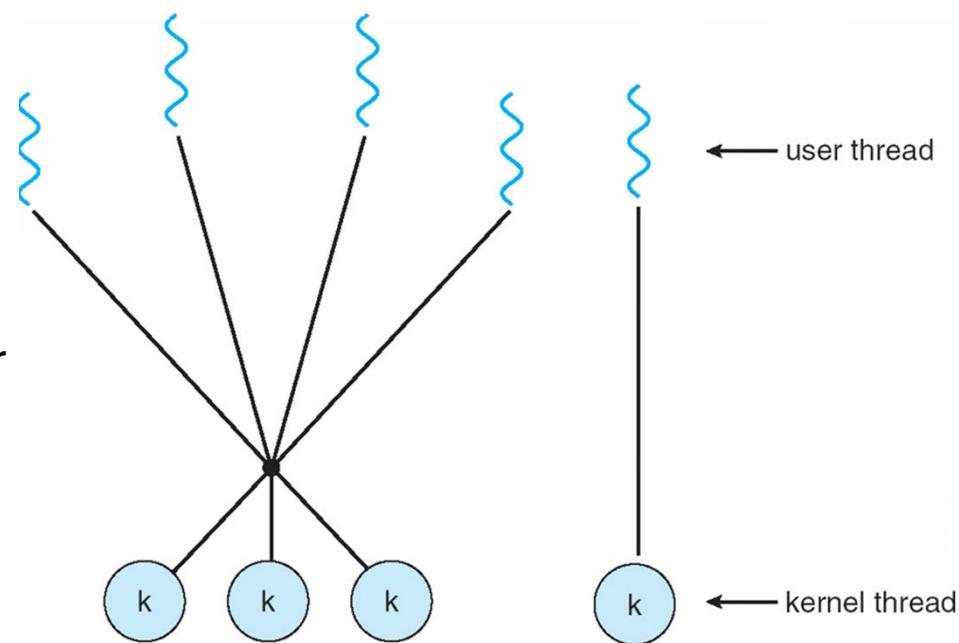


Two-level Model

- Has all the functionality of the many-to many model but also allows a user-level thread to be bound to a kernel thread

- Examples

- IRIX
- HP-UX
- Tru64 UNIX
- Solaris 8 and earlier





Chapter 4. Contents

- Overview
- Multithreading Models
- Thread Libraries
- Operating System Examples
- Signals





Thread Libraries

- [Thread library](#) provides programmer with API for creating and managing threads
- Two primary ways of implementing
 - No kernel support, strictly in user space so no system calls involved
 - Kernel level directly supported by the OS. All code and data structures for the library exists in kernel space
 - ▶ An API call typically invokes a system call
- Three main libraries in use
 - ▶ POSIX (Portable Operating System Interface) threads
 - ▶ Win32
 - ▶ Java





Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- Pthreads is an IEEE and Open Group certified product
- The API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)





Pthreads

- POSIX standard threads (Pthreads)
- First thread exists in main(), typically creates the others
- `pthread_create (thread,attr,start_routine,arg)`
 - Returns new thread ID in “thread”
 - Executes routine specified by “start_routine” with argument specified by “arg”
 - Exits on return from routine or when told explicitly





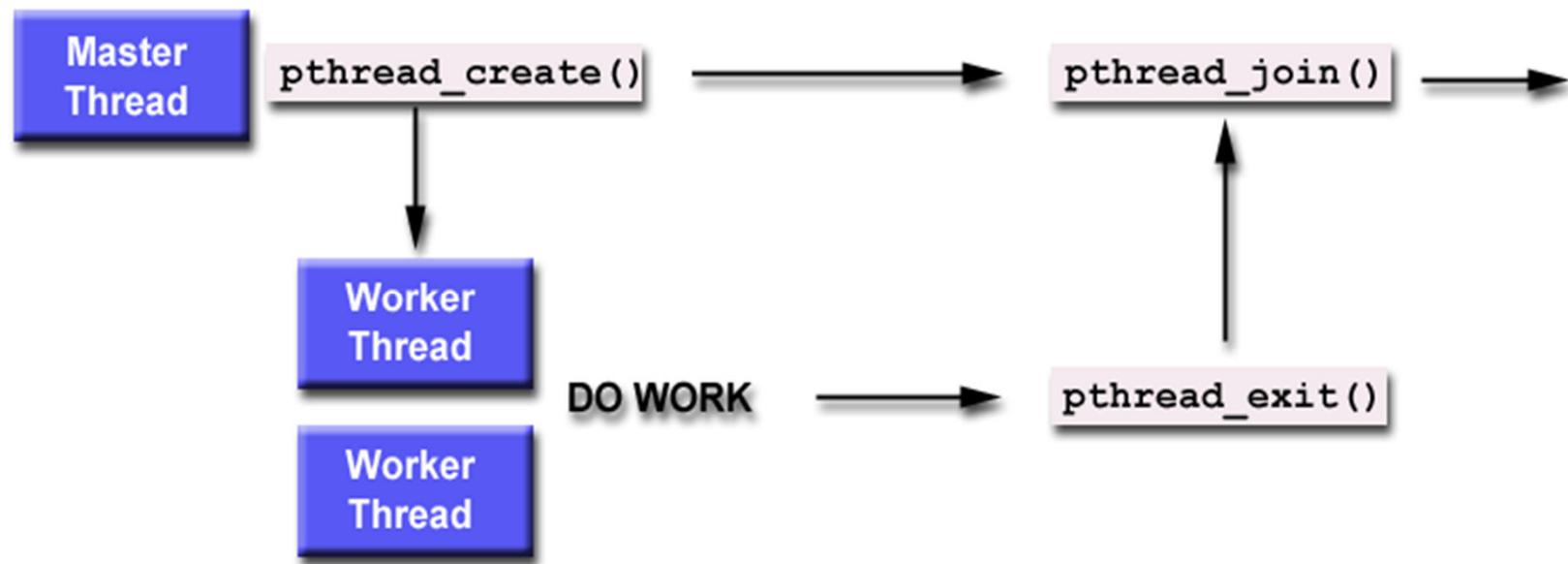
Pthreads (continued)

- **pthread_exit (status)**
 - Terminates the thread and returns “status” to any joining thread
- **pthread_join (threadid,status)**
 - Blocks the calling thread until thread specified by “threadid” terminates
 - Return status from pthread_exit is passed in “status”
 - One way of synchronizing between threads
- **pthread_yield ()**
 - Thread gives up the CPU and enters the ready queue





Using create, join and exit primitives





An example Pthreads program

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    printf("\n%d: Hello World!\n", *(int *)threadid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0; t<NUM_THREADS; t++)
    {
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc)
        {
            printf("ERROR: return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
        pthread_exit(NULL);
    }
}
```

Program Output

```
Creating thread 0
Creating thread 1
0: Hello World!
1: Hello World!
Creating thread 2
Creating thread 3
2: Hello World!
3: Hello World!
Creating thread 4
4: Hello World!
```





Chapter 4. Contents

- Overview
- Multithreading Models
- Thread Libraries
- Operating System Examples
- Signals





Operating System Examples

- Windows XP Threads
- Linux Thread





Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.





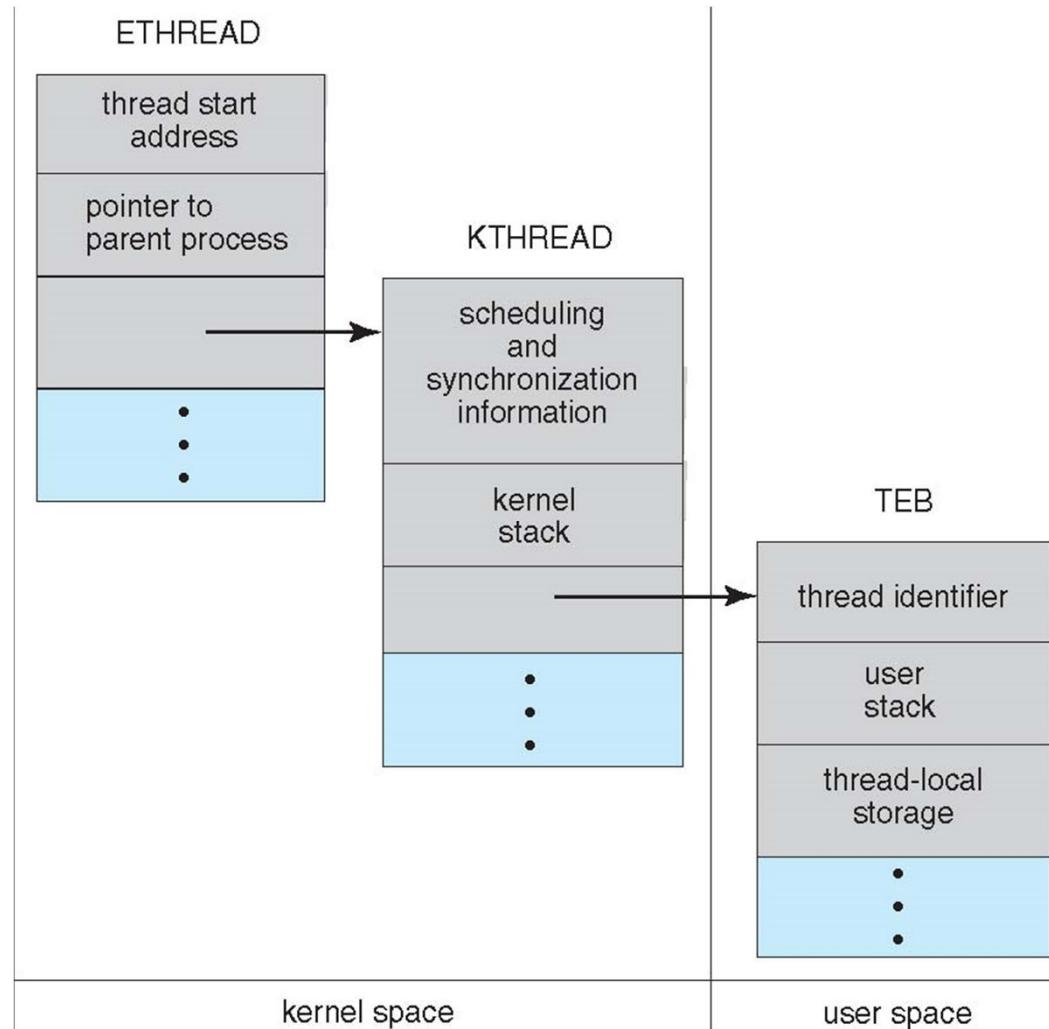
Windows XP Threads

- Implements the one-to-one mapping
- Each thread contains
 - A thread id
 - Register set
 - Separate user and kernel stacks
 - Private data storage area
- A Windows XP application runs as a separate process and each process may contain one or more threads
- The register set, stacks, and private storage area are known as the **context** of the threads
- The primary data structures of a thread include:
 - ETHREAD (executive thread block)
 - KTHREAD (kernel thread block)
 - TEB (thread environment block)
- Also provides support for a fiber library which implements the many-to-many model





Windows XP Threads





Chapter 4. Contents

- Overview
- Multithreading Models
- Thread Libraries
- Operating System Examples
- Signals





Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
 - **SIGKILL** -9- (Kill : terminate immediately)
 - **SIGTERM** -15- (Termination : request to terminate)
 - **SIGHLD** -17- (Child process terminated, stopped or continued)
 - **SIGCONT** -18- (Continue if stopped)
 - **SIGSTOP** -19- (Stop executing temporarily)
 - **SIGTTIN** -21. (Background process attempting to read from tty)
 - **SIGTTOU** -22- (Background process attempting to write to tty)





Signal Handling

- A **signal handler** is used to process signals
 1. Signal is generated by particular event
 2. Signal is delivered to a process
 3. Signal is handled
- Options:
 - Deliver the signal to the thread to which the signal applies
 - Deliver the signal to every thread in the process
 - Deliver the signal to certain threads in the process
 - Assign a specific thread to receive all signals for the process



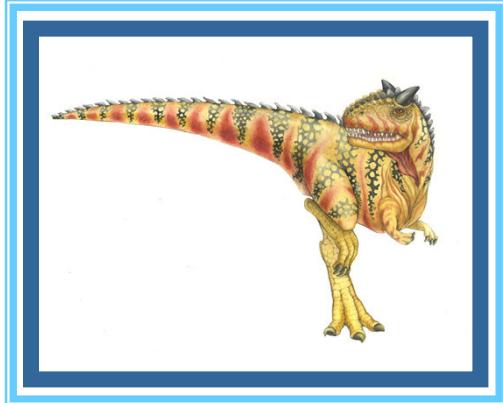


Signal Handling in Unix

- Signal handlers can be installed with the **signal()** system call.
 - When the signal is intercepted , the signal handler is invoked
- If a signal handler is not installed for a particular signal, the default handler is used.
- .The process can also specify two default behaviors, without creating a handler:
 - ignore the signal (SIG_IGN)
 - use the default signal handler (SIG_DFL)
- The **sigprocmask()** call can be used to block and unblock delivery of signals.



Chapter 5: Process Scheduling



Silberschatz, Galvin and Gagne ©2009

Rudowsky ©2005

Walpole ©2010

Kubiatowicz ©20010



Chapter 5. Contents

- Basic Concepts. Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling. Multiple-Processor Scheduling
- Operating Systems Examples





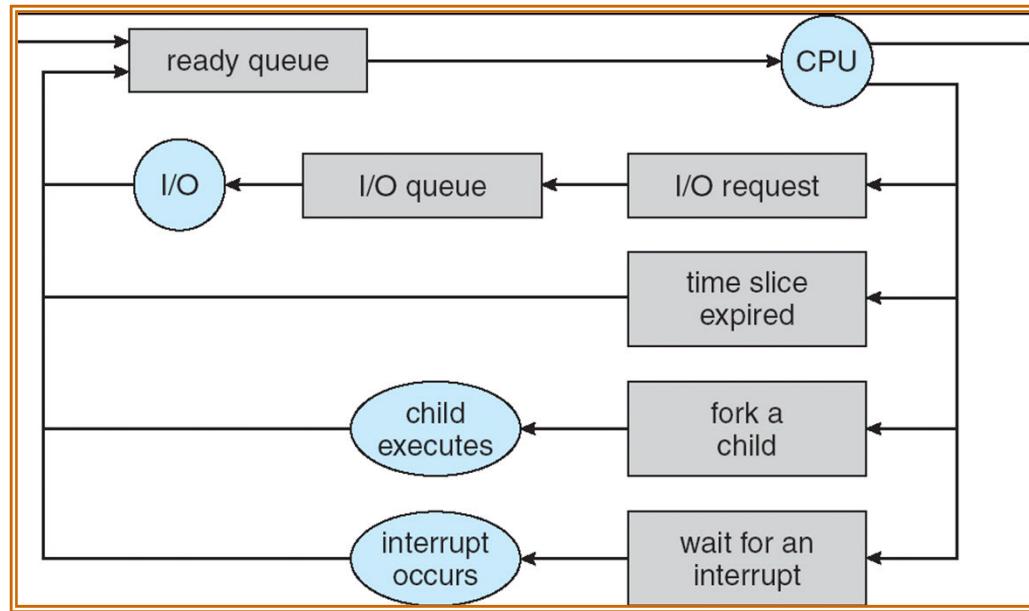
Objectives

- To introduce process scheduling, which is the basis for multiprogrammed operating systems
- To describe various process-scheduling algorithms





CPU Scheduling



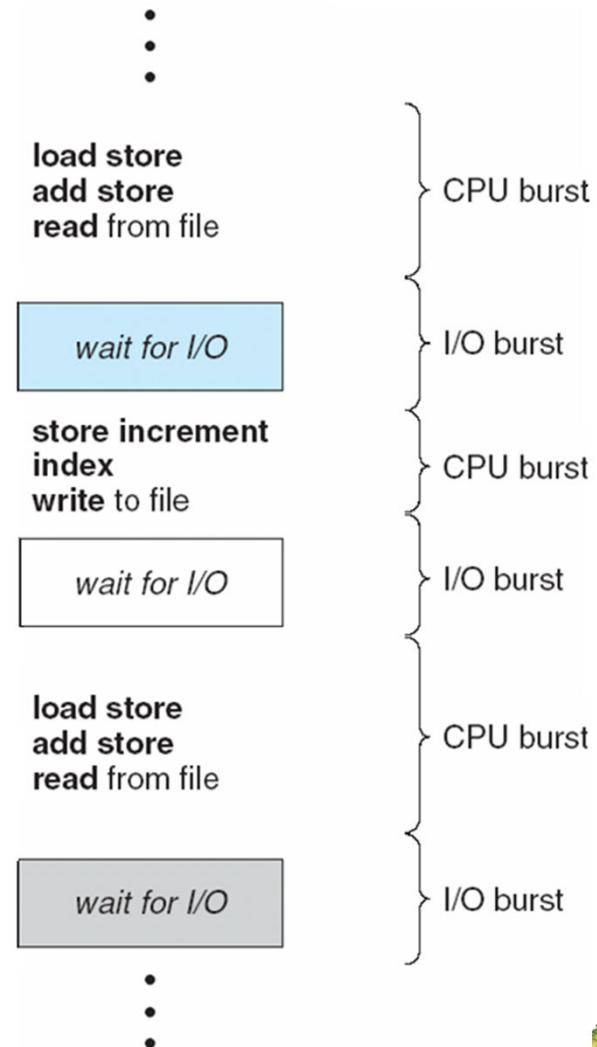
- Earlier, we talked about the life-cycle of a thread
 - Active threads work their way from Ready queue to Running to various waiting queues.
- Question: How is the OS to decide which of several tasks to take off a queue?
 - Obvious queue to worry about is ready queue
 - Others can be scheduled as well, however
- **Scheduling:** deciding which threads are given access to resources from moment to moment.





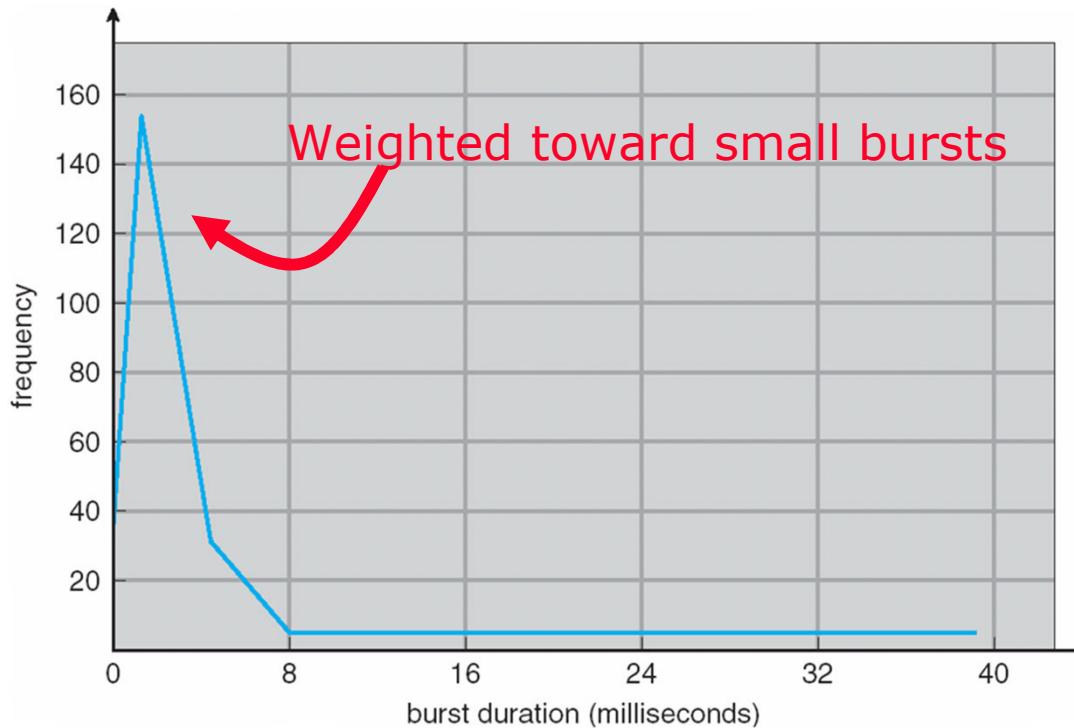
Assumption: CPU Bursts

- Scheduling is central to OS design: CPU and almost all computer resources are scheduled before use
- Execution model: programs alternate between bursts of CPU and I/O
 - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
 - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
 - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst





Histogram of CPU-burst Times



- CPU bursts, though different by process and computer, tend to have a frequency curve as shown above
 - Characterized by many short burst and few long ones
 - The distribution can help in the selection of an appropriate scheduling algorithm





Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time (= Completion Time)** – amount of time to execute a particular process
 - waiting to get into memory + waiting in the ready queue + executing on the CPU + I/O
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)





Scheduling Algorithm Optimization Criteria

- Max CPU utilization
 - Max throughput
 - Min turnaround time
 - Min waiting time
 - Min response time
-
- In most cases we optimize the average measure
 - In some circumstances we want to optimize the min or max values rather than the average – e.g., minimize the max response time so all users get good service





Chapter 5. Contents

- Basic Concepts. Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling. Multiple-Processor Scheduling
- Operating Systems Examples





CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
 1. Switches from running to waiting state
 2. Switches from running to ready state
 3. Switches from waiting to ready
 4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**:
 - Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or switching to a waiting state
 - Used by Windows 3.1 and Apple Macintosh
- All other scheduling is **preemptive**
 - Requires mechanisms to coordinate access to shared data
 - Sections of kernel code must be guarded from simultaneous use





Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
 - switching context
 - switching to user mode
 - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running





Scheduling Algorithms

Preemption	Non Preemptive	Preemptive
Scheduling		
Arrival order	First Come First Served	Round Robin
Running time	Shortest Job First	Shortest Remaining Time First
Priority	Non Preemptive with priority	Preemptive with priority





First-Come, First-Served (FCFS) Scheduling

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1, P_2, P_3
The Gantt Chart for the schedule is:



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
- Average completion time: $(24 + 27 + 30)/3 = 27$
- *Convoy effect*: short process behind long process

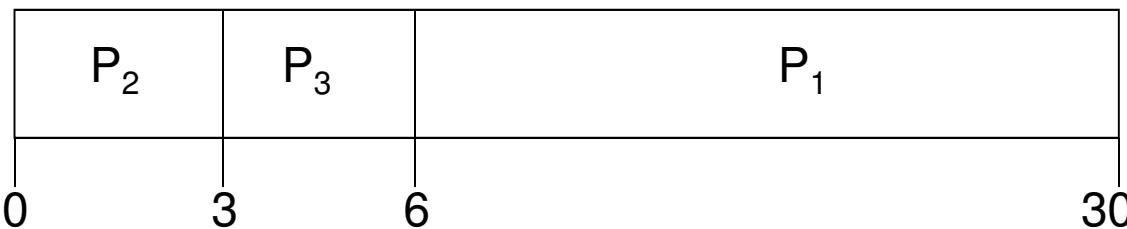




FCFS Scheduling (Cont)

Suppose that the processes arrive in the order: P_2, P_3, P_1

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Average completion time: $(3 + 6 + 30)/3 = 13$

- In second case:
 - average waiting time is much better (before it was 17)
 - Average completion time is better (before it was 27)





FCFS Scheduling (Cont)

- FCFS is non-preemptive
- FCFS is simple (+)
- Not good for time sharing systems where each user needs to get a share of the CPU at regular intervals
- ***Convoy effect*** short process (I/O bound) wait for one long CPU-bound process to complete a CPU burst before they get a turn
 - lowers CPU and device utilization

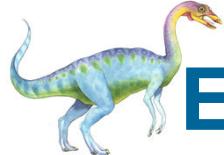




Round Robin (RR)

- FCFS Scheme: Potentially bad for short jobs!
 - Depends on submit order
- Round Robin Scheme
 - Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds.
 - After this time has elapsed, the process is preempted and added to the end of the ready queue.
 - If there are n processes in the ready queue and the time quantum is q :
 - ▶ each process gets $1/n$ of the CPU time
 - ▶ in chunks of at most q time units at once.
 - ▶ No process waits more than $(n-1)q$ time units.
- Performance
 - q large \Rightarrow FCFS
 - q small \Rightarrow Interleaved (q must be large with respect to context switch, otherwise overhead is too high)



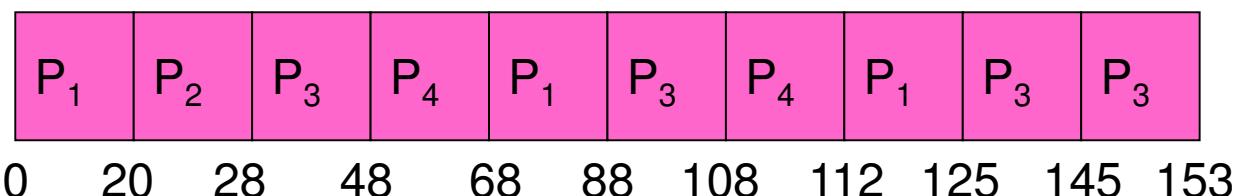


Example of RR with Time Quantum = 20

- Example:

<u>Process</u>	<u>Burst Time</u>
P_1	53
P_2	8
P_3	68
P_4	24

- The Gantt chart is:



- Waiting time for

$$P_1 = (68-20)+(112-88)=72$$

$$P_2 = (20-0)=20$$

$$P_3 = (28-0)+(88-48)+(125-108)=85$$

$$P_4 = (48-0)+(108-68)=88$$

- Average waiting time = $(72+20+85+88)/4=66\frac{1}{4}$
- Average completion time = $(125+28+153+112)/4 = 104\frac{1}{2}$

- Thus, Round-Robin Pros and Cons:

- Better for short jobs, Fair (+)
- Context-switching time adds up for long jobs (-)





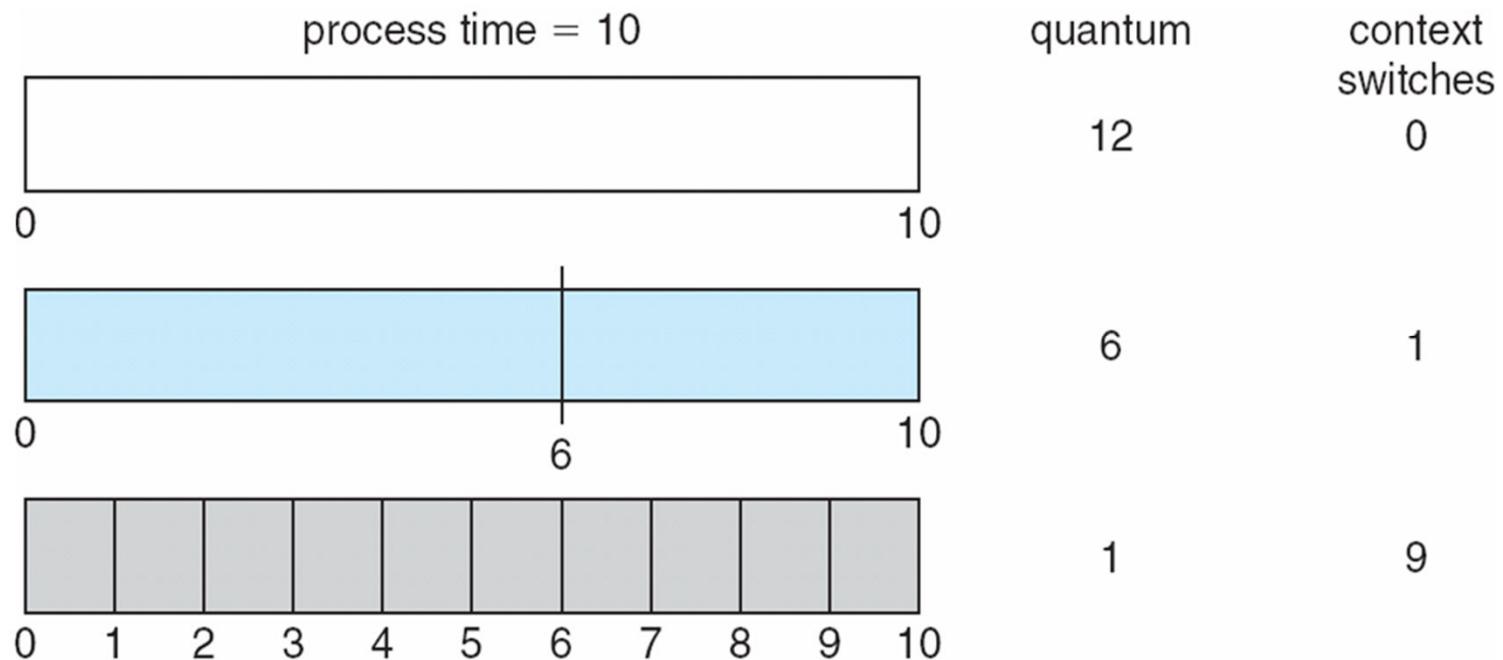
Round-Robin Discussion

- How do you choose time slice?
 - What if too big?
 - ▶ Response time suffers
 - What if infinite (∞)?
 - ▶ Get back FCFS
 - What if time slice too small?
 - ▶ Throughput suffers!
 - ▶ increases context switches
- Actual choices of timeslice:
 - Initially, UNIX timeslice one second:
 - ▶ Worked ok when UNIX was used by one or two people.
 - ▶ What if three compilations going on? 3 seconds to echo each keystroke!
 - In practice, need to balance short-job performance and long-job throughput:
 - ▶ Typical time slice today is between 10ms – 100ms
 - ▶ Typical context-switching overhead is 0.001ms – 0.01ms
 - ▶ Roughly 1% overhead due to context-switching





Time Quantum and Context Switch Time



A smaller time quantum increases context switches





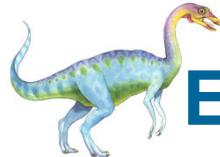
Comparisons: FCFS and Round Robin

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example:
 - 10 jobs, each take 100s of CPU time
 - RR scheduler quantum of 1s
 - All jobs start at the same time
- Completion Times:

Job #	FIFO	RR
1	100	991
2	200	992
...
9	900	999
10	1000	1000

- Both RR and FCFS finish at the same time
- Average response time is much worse under RR!
 - ▶ Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
 - Total time for RR longer even for zero-cost switch !





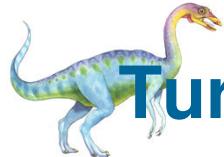
Earlier Example with Different Time Quantum

Best FCFS:

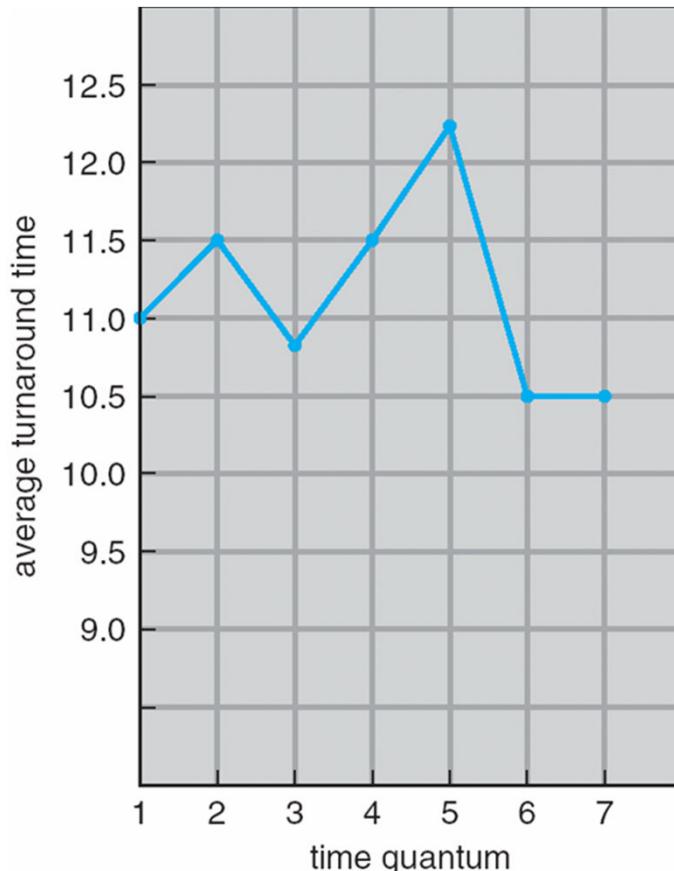
	P ₂ [8]	P ₄ [24]	P ₁ [53]	P ₃ [68]	
	0	8	32	85	153

	Quantum	P ₁	P ₂	P ₃	P ₄	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61¼
	Q = 8	80	8	85	56	57¼
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾





Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

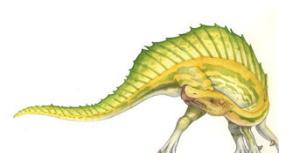
- The average turnaround time of a set of processes does not necessarily improve as the time quantum size increases
- In general, it can be improved if most processes finish their next CPU burst in a single time quantum





What if we Knew the Future?

- Could we always mirror best FCFS?
- Shortest Job First (SJF):
 - Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time (on a tie use FCFS)
 - ***nonpreemptive*** – once CPU given to the process it cannot be preempted until completes its CPU burst
- Shortest Remaining Time First (SRTF):
 - Preemptive version of SJF: if a new process arrives with CPU burst length less than remaining time of current executing process, preempt
- These can be applied either to a whole program or the current CPU burst of each program
 - Idea is to get short jobs out of the system
 - Big effect on short jobs, only small effect on long ones
 - Result is better average response time
- The difficulty is knowing the length of the next CPU request

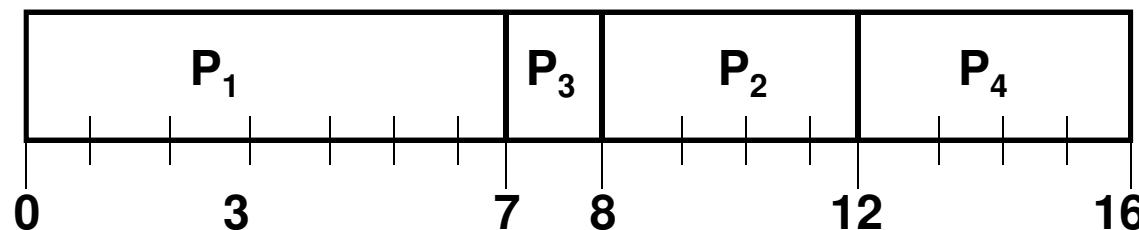




Example of SJF

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF scheduling chart



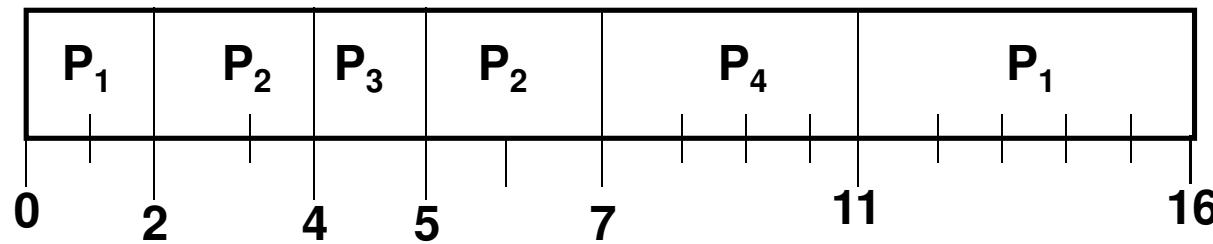
- Average waiting time = $(0 + 6 + 3 + 7) / 4 = 4$





Example of SRTF

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



- Average waiting time = $(9 + 1 + 0 + 2)/4 = 3$





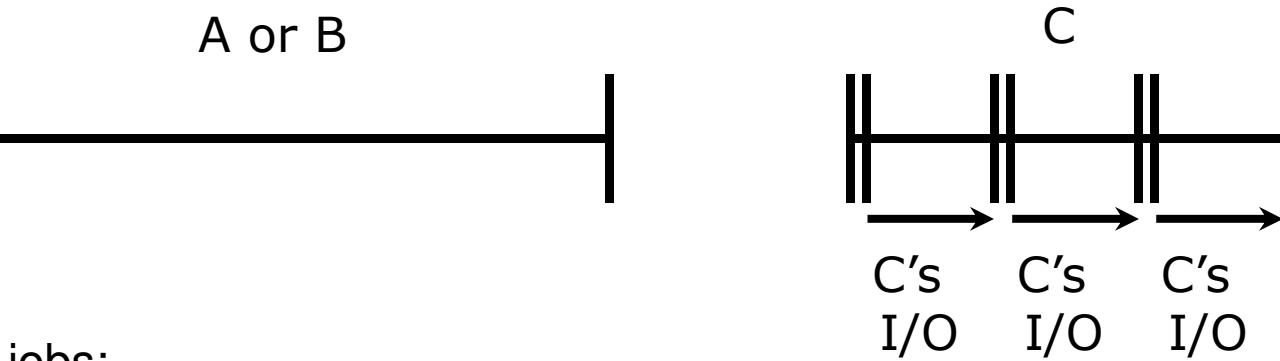
Discussion

- SJF/SRTF are the best you can do at minimizing average response time
 - Probably optimal (SJF among non-preemptive, SRTF among preemptive)
 - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS and RR
 - What if all jobs the same length?
 - ▶ SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
 - What if jobs have varying length?
 - ▶ SRTF (and RR): short jobs not stuck behind long ones





Example to illustrate benefits of SRTF

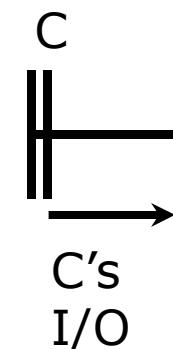


- Three jobs:
 - A,B: both CPU bound, run for week
 - C: I/O bound, loop 1ms CPU, 9ms disk I/O
 - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FIFO:
 - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
 - Easier to see with a timeline





SRTF Example continued:

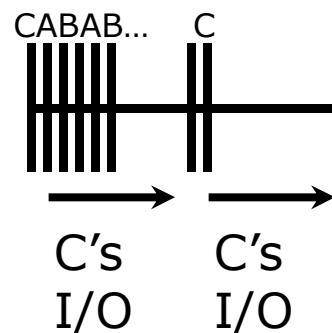


A

B

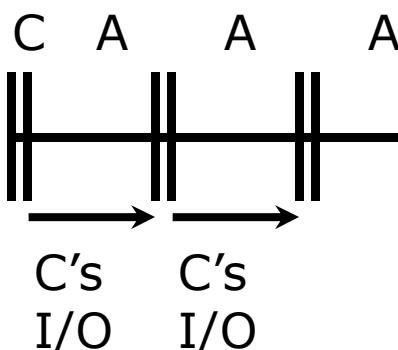
RR 100ms time slice

Disk Utilization:
9/201 ~ 4.5%



RR 1ms time slice

Disk Utilization:
~90% but lots of
wakeup!



SRTF

Disk Utilization:
90%





SRTF Further discussion

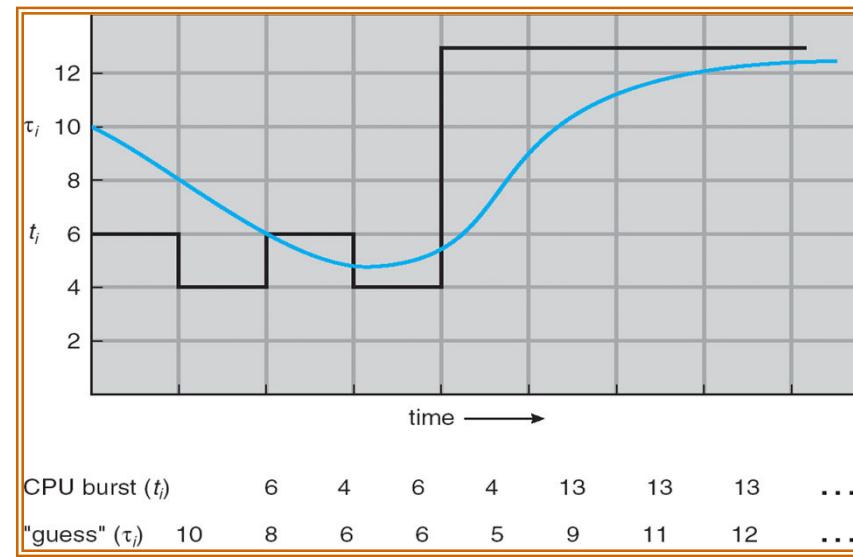
- Starvation
 - SRTF can lead to starvation if many small jobs!
 - Large jobs never get to run
- Somehow need to predict future
 - How can we do this?
 - Some systems ask the user
 - ▶ When you submit a job, have to say how long it will take
 - ▶ To stop cheating, system kills job if takes too long
 - But: Even non-malicious users have trouble predicting runtime of their jobs
- Bottom line, can't really know how long job will take
 - However, can use SRTF as a yardstick for measuring other policies
 - Optimal, so can't do any better
- SRTF Pros & Cons
 - Optimal (average response time) (+)
 - Hard to predict future (-)
 - Unfair (-)





Predicting the Length of the Next CPU Burst

- **Adaptive:** Changing policy based on past behavior
 - CPU scheduling, in virtual memory, in file systems, etc
 - Works because programs have predictable behavior
 - ▶ If program was I/O bound in past, likely in future
 - ▶ If computer behavior were random, wouldn't help
- Example: SRTF with estimated burst length
 - Use an estimator function on previous bursts:
Let t_{n-1} , t_{n-2} , t_{n-3} , etc. be previous CPU burst lengths. Estimate next burst $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \dots)$
 - Function f could be one of many different time series estimation schemes (Kalman filters, etc)
 - For instance,
exponential averaging
$$\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$$
with $(0 < \alpha \leq 1)$





Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer \equiv highest priority)
 - Preemptive (compares priority of process that has arrived at the ready queue with priority of currently running process)
 - Nonpreemptive (put at the head of the ready queue)
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem \equiv **Starvation** – low priority processes may never execute
- Solution \equiv **Aging** – as time progresses increase the priority of the process





Priority Scheduling

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

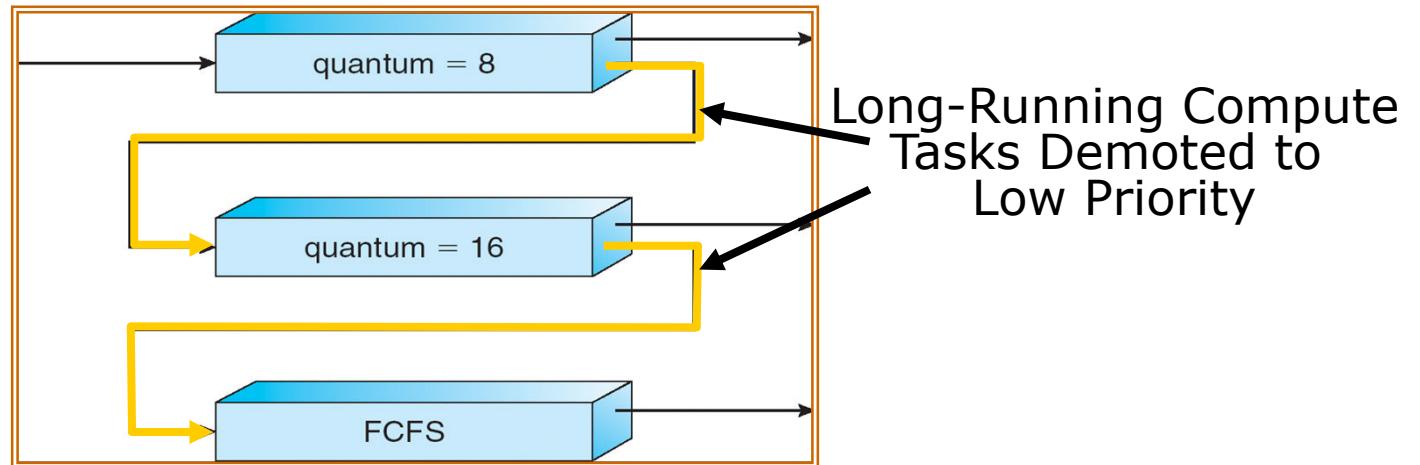


Average waiting time: 8.2





Multi-Level Feedback Scheduling



- Another method for exploiting past behavior
 - First used in CTSS
 - **Multiple queues, each with different priority**
 - ▶ Higher priority queues often considered “foreground” tasks
 - **Each queue has its own scheduling algorithm**
 - ▶ e.g. foreground – RR, background – FCFS
 - ▶ Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next:2ms, next: 4ms, etc)
- Adjust each job’s priority as follows (details vary)
 - Job starts in highest priority queue
 - If timeout expires, drop one level
 - If timeout doesn’t expire, push up one level (or to top)





Multilevel Feedback Scheduling

- Result approximates SRTF:
 - CPU bound jobs drop like a rock
 - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
 - **Fixed priority scheduling:**
 - ▶ serve all from highest priority, then next priority, etc.
 - ▶ Possibility of starvation: long running jobs may never get CPU
 - In Multics, shut down machine, found 10-year-old job
 - ▶ Solution: could increase priority of jobs that don't get service
 - **Time slice:**
 - ▶ each queue gets a certain amount of CPU time
 - ▶ e.g., 70% to highest, 20% next, 10% lowest
- **Countermeasure:** user action that can foil intent of the OS designer
 - For multilevel feedback, put in a bunch of meaningless I/O to keep job's priority high
 - ▶ Put in printf's, ran much faster!
 - Of course, if everyone did this, wouldn't work!





Multilevel Feedback Parameters

- Multilevel-feedback-queue scheduler defined by the following parameters:
 - number of queues
 - scheduling algorithms for each queue
 - method used to determine when to upgrade a process
 - method used to determine when to demote a process
 - method used to determine which queue a process will enter when that process needs service
- Example:
 - Job starts in highest priority queue
 - If timeout expires, drop one level
 - If timeout doesn't expire, push up one level (or to top)





Chapter 5. Contents

- Basic Concepts. Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling. Multiple-Processor Scheduling
- Operating Systems Examples





Thread Scheduling

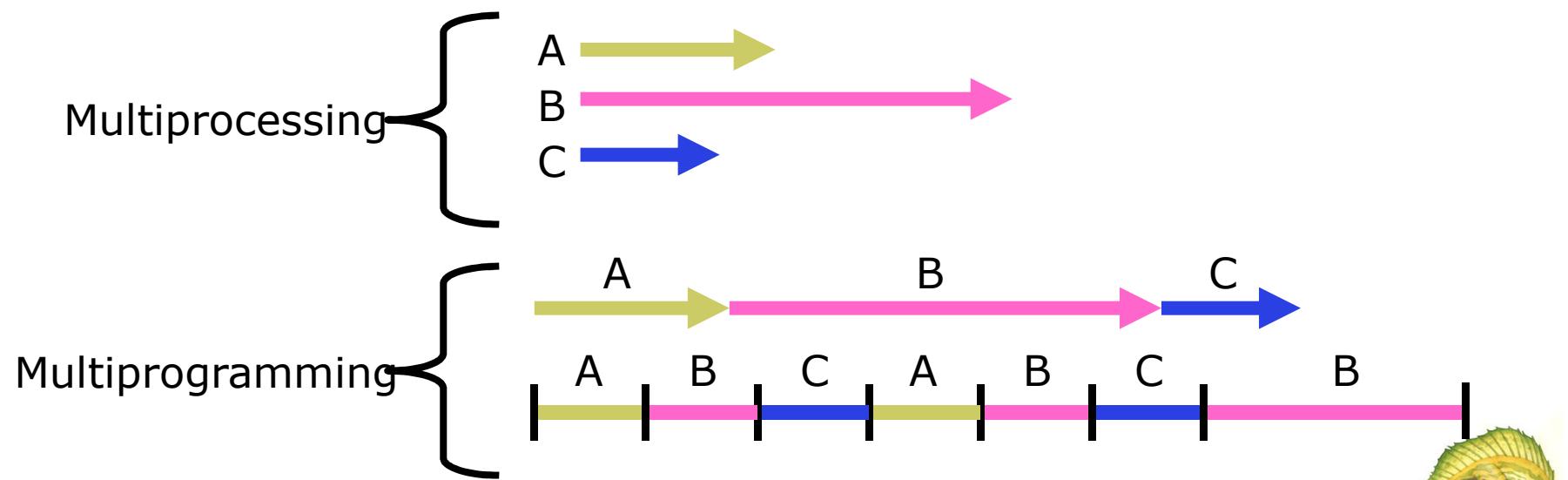
- Distinction between user-level and kernel-level threads:
 - Kernel-level threads (not processes) are scheduled by the O.S.
 - User-level threads are managed by the thread library
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
 - Known as **process-contention scope (PCS)** since scheduling competition takes place among threads within the process
- Systems using the one-to-one model (Windows XP, Solaris, Linux) schedule threads using the **system-contention scope (SCS)** – competition for the CPU takes place among all threads in system
- Pthread API allows specifying either PCS or SCS during thread creation:
 - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
 - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling. The only one possible for Linux and Mac OS X systems





Review: Multiprocessing vs Multiprogramming

- Remember Definitions:
 - Multiprocessing ≡ Multiple CPUs
 - Multiprogramming ≡ Multiple Jobs or Processes
 - Multithreading ≡ Multiple threads per Process
- What does it mean to run two threads “concurrently”?
 - Scheduler is free to run threads in any order and interleaving: FIFO, Random, ...
 - Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks





Multiple-Processor Scheduling

- When multiple CPUs are available **load sharing** becomes possible but CPU scheduling more complex
- We concentrate on **homogeneous multiprocessing**: all the processors are identical in terms of their functionality. Approaches:
 - **Asymmetric multiprocessing** – only one processor has all scheduling decisions, I/O processing and other system activities (= only one accesses the system data structures, alleviating the need for data sharing)
 - **Symmetric multiprocessing (SMP)** – each processor is self-scheduling:
 - ▶ All processes in a common ready queue
 - ▶ Each has its own private queue of ready processes
 - All modern O.S. support SMP: Windows XP, Windows 2000, Solaris, Linux, Mac OS X





Multiple-Processor Scheduling

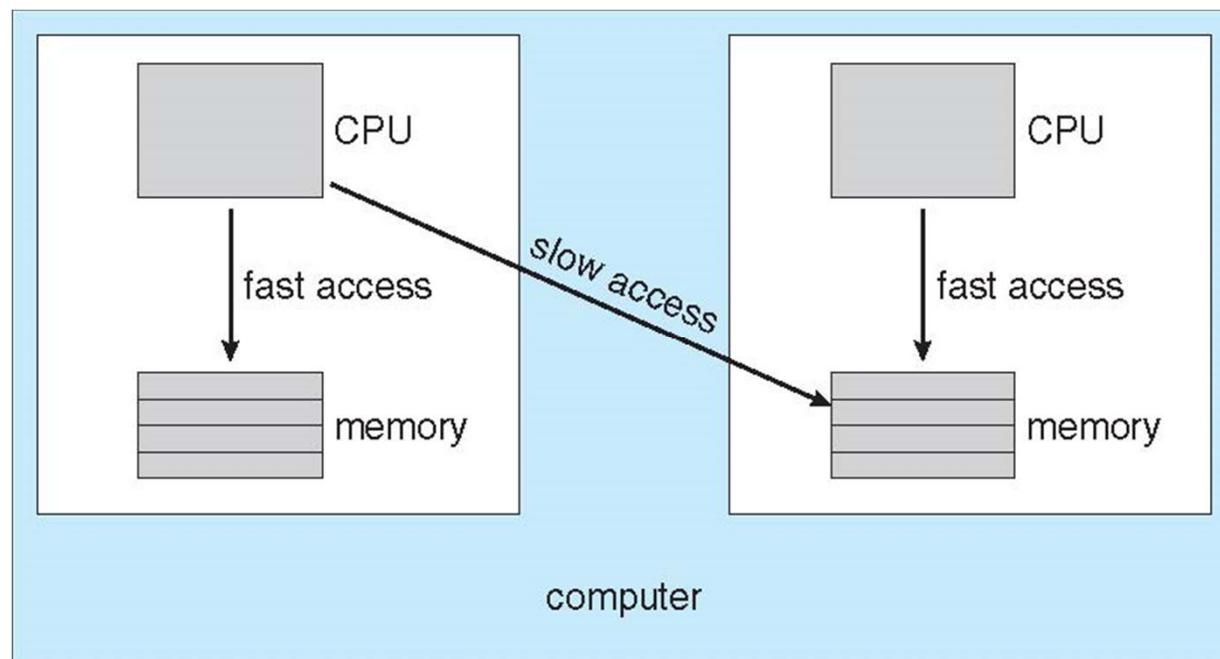
- **Local cache accesses.** When a process remains executing on the same processor: data most recently accessed populates the cache for that processor. If the process migrates to another processor:
 1. the contents of the cache must be invalidated for the first processor,
 2. and the cache for the second processor must be repopulated.
- **Processor affinity** – process has affinity for processor on which it is currently running
 - **soft affinity**- OS attempts to keep a process running on the same processor (but not guaranteeing that it will do so)
 - **hard affinity** (OS provides system calls allowing a process to specify that it is not to migrate to other processors)





NUMA and CPU Scheduling

- Affinity issues are affected by the main memory architecture of a system.
- NUMA = Non Uniform Memory Access. A CPU has faster access to some parts of main memory than to other parts (systems containing combined CPU and memory boards)
- System's CPU scheduler and memory placement algorithms should work together: a process that is assigned affinity to a particular CPU can be allocated memory on the board where that CPU resides.





Recent trends

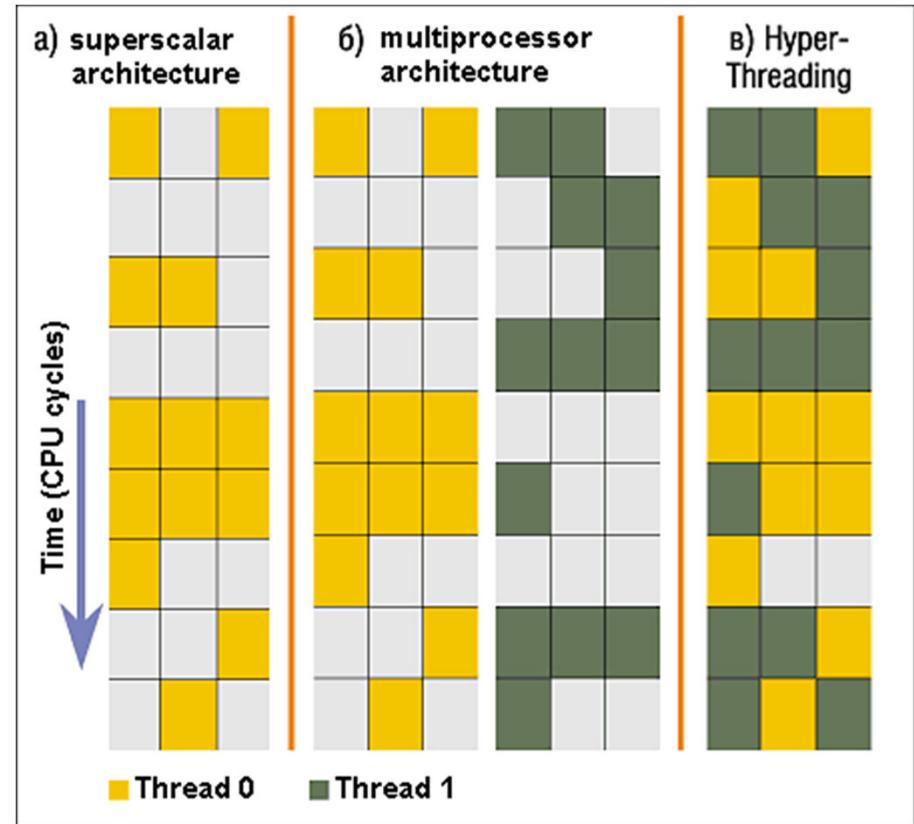
- **Multicore Processors:** Recent trend to place multiple processor cores on same physical chip
 - Faster and consume less power
- Multiple threads per core also growing: **SMT/Hyperthreading**
 - Takes advantage of memory stall to make progress on another thread while memory retrieve happens





SMT/Hyperthreading

- Hardware technique
 - Exploit natural properties of superscalar processors to provide illusion of multiple processors
 - Higher utilization of processor resources
- **Can schedule each thread as if were separate CPU**
 - However, not linear speedup!
 - If have multiprocessor, should schedule each processor first
- Original technique called “Simultaneous Multithreading”
 - See <http://www.cs.washington.edu/research/smt/>
 - Alpha, SPARC, Pentium 4 (“Hyperthreading”), Power 5





Chapter 5. Contents

- Basic Concepts. Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling. Multiple-Processor Scheduling
- Operating Systems Examples





Operating System Examples

- Windows XP scheduling
- Linux scheduling





Windows XP Priorities

- Priority based, preemptive scheduling algorithm; highest priority always runs
- Priorities are divided into two classes:
 - Variable class – 1 through 15 and
 - Real time class – 16 through 31
- Priority is based on priority class it belongs to (realtime, high etc) and the relative priority within each class
 - When a threads quantum runs out the thread is interrupted. If it is in the variable priority class its priority is lowered (never below the base priority for that class)
 - Gives good response to interactive threads (mouse, windows) and enables I/O bound threads to keep the I/O devices busy while compute bound use spare CPU cycles in the background

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

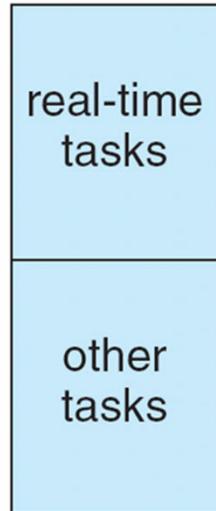




Linux Scheduling

- Preemptive, priority based with two separated priority ranges: time-sharing and real-time
- Higher priority tasks have longer time quanta
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140

numeric priority	relative priority	time quantum
0	highest	200 ms
•		
•		
•		
99		
100		
•		
•		
•		
140	lowest	10 ms

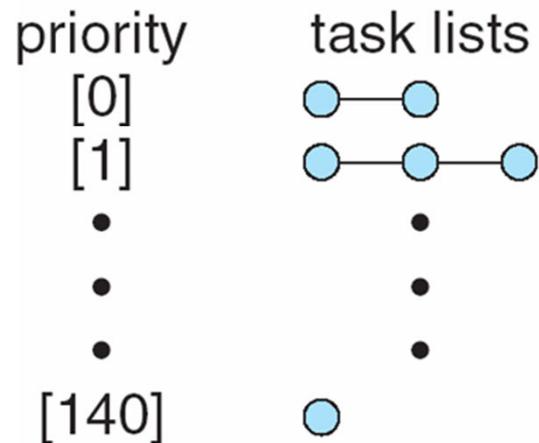




List of Tasks Indexed According to Priorities

- Constant order $O(1)$ scheduling time
- Kernel maintains list of all runnable tasks (task that has time remaining on its time slice) in a runqueue data structure
 - Each **runqueue** contains two priority arrays - **active** and **expired**
 - When all active tasks have exhausted their time slices, the two priority arrays are exchanged (active \leftrightarrow expired)

**active
array**



**expired
array**

