

# CSC 539: Operating Systems Structure and Design

Spring 2006

## Processes and threads

- process concept
- process scheduling: state, PCB, process queues, schedulers
- process operations: create, terminate, wait, ...
- cooperating processes: shared memory, message passing
- threads

1

## Process

an operating system executes a variety of programs:

batch system – *jobs*

time-shared systems – *user programs or tasks*

often, the terms *job* and *process* are used interchangeably

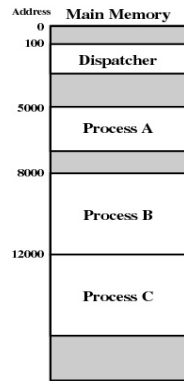
a *process* is a program in execution

a process (active entity) is both more and less than a program (passive entity)

- in addition to code, process involves program counter, registers, stack, data, ...
- same program can produce more than one process (e.g., users running pine)
- the OS interleaves the execution of several processes to maximize processor utilization
- the OS supports InterProcess Communication (IPC) and user creation of processes

2

## Tracing process execution



consider 3 processes in memory:

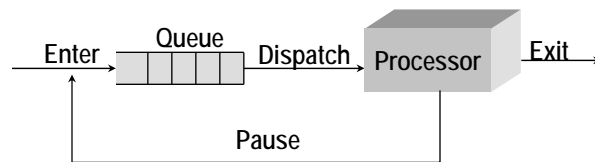
- OS must manage process scheduling
- Dispatcher swaps out active process if
  - (1) timeout occurs, or
  - (2) process requests I/O operation

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

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

## Simple queueing diagram



in this simple system, a single queue suffices to store processes

- a process is either active (executing) or inactive (waiting)
- the dispatcher is code that assigns the CPU to one process or another
  - it avoids wasted CPU cycles as the active process waits for I/O
  - it prevents a single process from monopolizing CPU time (time slicing)

note: this is the model from HW 2

## Process states

as a process executes, it changes *state*

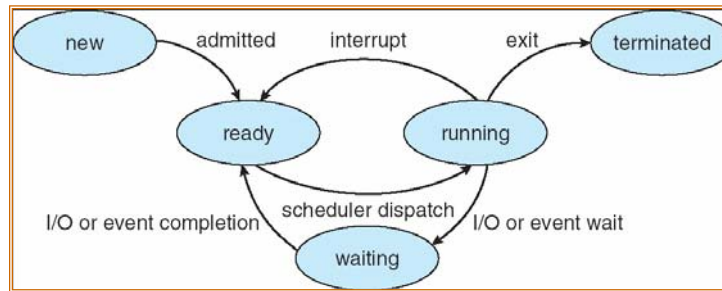
*New*: the process is being created

*Ready*: the process is waiting to be assigned to the CPU

*Running*: instructions are being executed

*Waiting or blocked*: the process is waiting for some event to occur (e.g., I/O operation)

*Terminated*: the process has finished execution

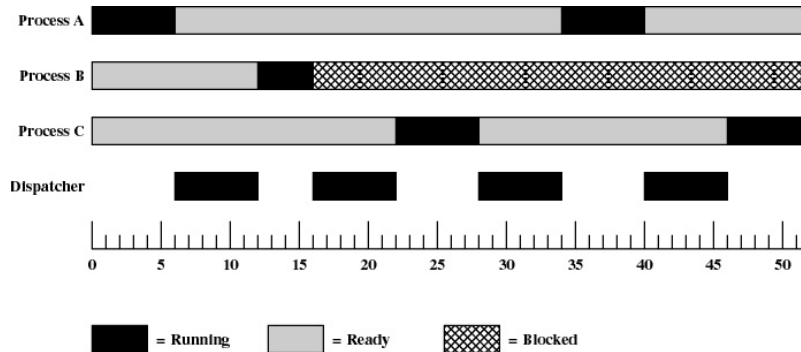


5

## Tracing process states

we can draw a timeline of CPU activity, identifying the states of the processes and Dispatcher

1	2000	27	12004
2	2001	28	12005
3	2002		
4	2003	29	100
5	2004	30	101
6	2005	31	102
		32	103
7	100	33	104
8	101	34	105
9	102	35	2006
10	103	36	2007
11	104	37	2008
12	105	38	2009
13	2000	39	2010
14	2001	40	2011
15	2002		
16	2003	41	100
		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



6

## Process Control Block (PCB)

for each process, the OS must store all info needed to:

- execute the process
- save its execution state if interrupted
- restore its execution state and continue

relevant info is stored in a Process Control Block (PCB), including:

- process state (e.g., ready, running, waiting, ...)
- program counter (address of next instr to be executed)
- CPU registers (active data stored in registers)
- CPU scheduling info (process priority, ptrs to scheduling queues, ...)
- memory-management info (base & limit registers, page tables, ...)
- accounting info (CPU time used, time limits, ...)
- I/O status info (allocated I/O devices, open files, ...)



7

## Process scheduling queues

### job queue

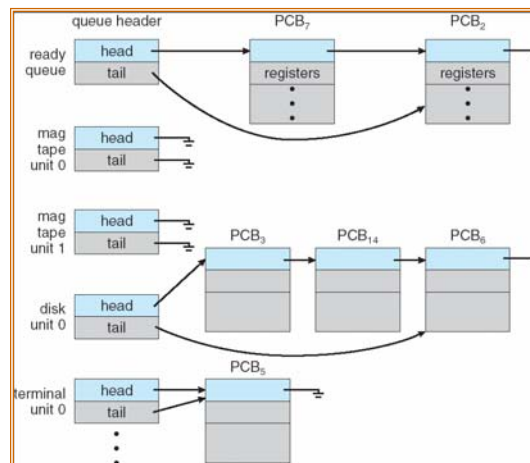
- stores PCBs of all processes in the system

### ready queue

- stores PCBs of processes in main memory, ready and waiting to execute

### device queues

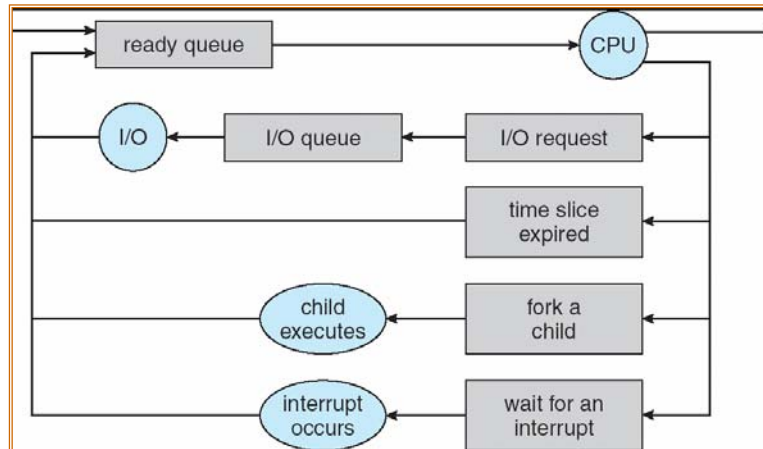
- stores PCBs of processes waiting for a particular I/O device



8

## Process queue flow

as the system operates, processes may flow from one queue to another



9

## Schedulers

### short-term scheduler (or CPU scheduler)

- selects which process should be executed next and allocates CPU.
- invoked frequently (milliseconds), so must be fast

### long-term scheduler (or job scheduler)

- selects which processes should be brought into the ready queue.
- controls the degree of *multiprogramming*
- invoked infrequently (seconds or minutes), so may be slow
- note: not all systems utilize a long-term scheduler  
e.g., in UNIX, assumption is that degrading performance will dissuade users

### processes can be described as either:

- *I/O-bound* – more I/O than computations, many short CPU bursts.
- *CPU-bound* – more computations than I/O; few long CPU bursts.

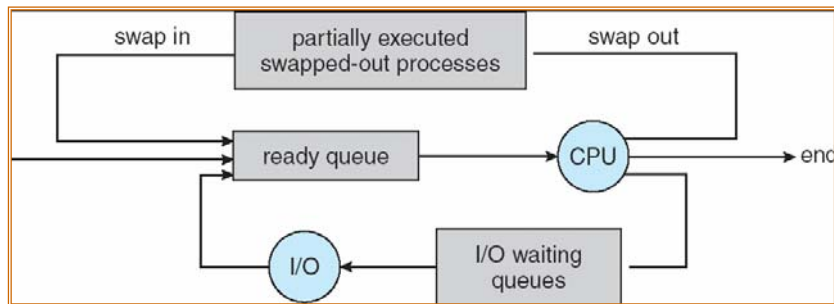
ideal: provide a mix of processes to fully utilize CPU and I/O devices

10

## Medium-term scheduler

some systems may introduce a medium-term scheduler

- temporarily swaps out processes from ready queue, restore at later time
- reduces degree of multiprogramming
- with no long-term scheduler, UNIX & Windows rely on a medium-term scheduler if the degree of multiprogramming exceeds main memory
- may also be utilized to improve the process mix



11

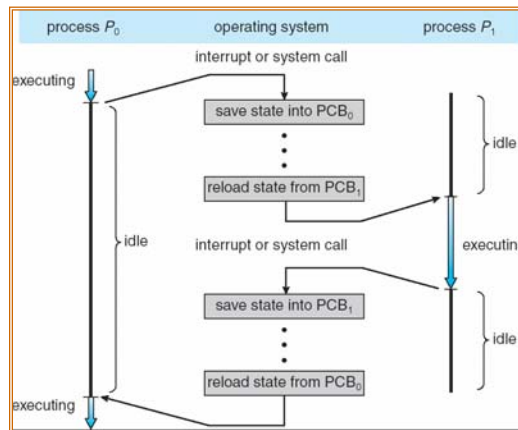
## CPU context switching

when switch to another process

- must save the state of old process and load saved state for new process

context-switch time is overhead

- the system does no useful work while switching
- time required for context-switch is dependent on hardware support.



12

## Operations on processes

### creation:

- parent process create children processes, which, in turn create other processes, forming a tree of processes
- parent and children may share resources or not
- parent and children may execute concurrently, or parent may wait on child
- child may inherit the address space of the parent, or have a new program loaded

### termination:

- process executes last statement and asks the operating system to delete it (exit)  
may output data from child to parent (via wait).  
process' resources are deallocated by operating system.
- parent may terminate execution of children processes (abort).
  - e.g., child has exceeded allocated resources, task is no longer required, parent is exiting (note: OS does not allow child to continue without parent)

13

## UNIX system calls

`fork()` creates a new process whose address space is a copy of the parent; child has process ID (PID) of 0 while parent PID is nonzero

`exec()` replaces the process' memory space with a new program

`wait()` suspends the parent process until the child terminates

### pseudocode for UNIX shell:

```
while(!EOF) {
  read input (PROGRAM + argc + argv)
  handle regular expressions
  int pid = fork();           // create a child
  if(pid == 0) {             // child continues here
    exec(PROGRAM, argc, argv0, argv1, ...);
  }
  else {                      // parent continues here
    ...
  }
}
```

14

## Command interface may provide ways to control processes

### UNIX:

`ps -a` lists all active processes on the machine  
`ps -au daverreed` lists all active processes for user `daverreed`  
  
`command &` runs command in background mode  
  
`kill -9 PID` terminates a process

### Windows:

`CTRL-ALT-DEL` displays all active processes in a window  
can view whether process is responsive  
can select process and terminate

15

## Cooperating processes

### *cooperating processes can affect each other's execution*

advantages include:

- information sharing (e.g., a shared file)
- computation speed-up (e.g., multiprocessor execution)
- modularity (e.g., split task into distinct parts, develop/test independently)
- convenience (e.g., user juggling many tasks, such as compile + edit + print)

### *cooperating processes can communicate via*

- shared memory  
consumer/producer model: one process writes to shared buffer, other reads from it
- message passing  
InterProcess Communication (IPC) facility must provide at least 2 operations:  
`send(message)` `receive(message)`  
can be direct (by name) or indirect (by ports),  
blocking (wait for delivery) or nonblocking (resume after sending)

16

## Examples of IPC

POSIX (int'l standard for OS interfaces, supported by Windows & UNIX) defines a mechanism for shared memory

- process must first create a shared memory segment using `shmget()` system call
- processes that wish to access the segment must attach it to their address space using `shmat()` system call

Mach (UNIX-based OS that underlies Mac OS X) also supports message passing to ports

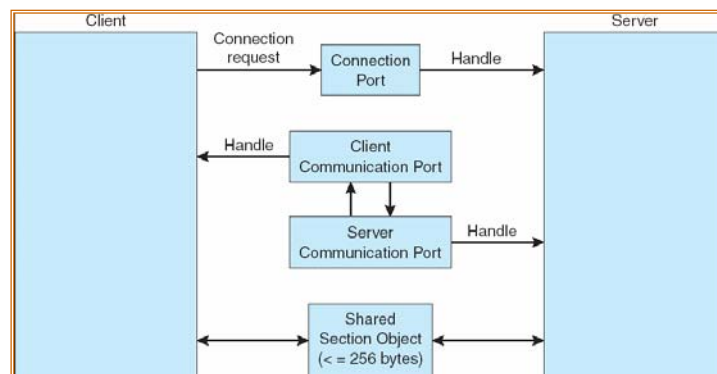
- when a process is created, two special ports (mailboxes) are created
  1. Kernel port is used by kernel to communicate with process
  2. Notification port is used to notify the process that an event has occurred
- additional ports can be created via `port_allocate()` system call
- messages are sent/received via `msg_send()` and `msg_receive()` system calls

17

## Examples of IPC (cont.)

Windows XP message passing is called Local Procedure-Call (LPC)

- for small messages,
  1. client sends connection request to server
  2. server sets up two private communication ports and notifies client
  3. messages are sent back and forth via the private ports
- for large messages,
  1. a shared section object is set up (shared memory)
  2. client & server can communicate by writing/reading shared memory



18

## Threads

### a thread is a lightweight process

- each thread has its own program counter, register values, and stack
- threads can share code, data and resources with other threads from same process

### a process can be divided into many threads

- some systems consider threads to be the fundamental execution unit  
e.g., Windows 2000, XP

### motivation

- an application might need to perform several different tasks  
e.g., Web browser: display images/text, download page, ...  
word processor: display text, read keystrokes, spell check, ...  
can associate a thread with each task, execute concurrently
- an application might need to perform the same task repeatedly  
e.g., Web server: receives many requests for pages, images, ...  
can associate a thread with each request, execute concurrently

19

## Advantages of threads

### responsiveness

- multithreading means that part of the process can be executing even when others are blocked/busy

### resource sharing

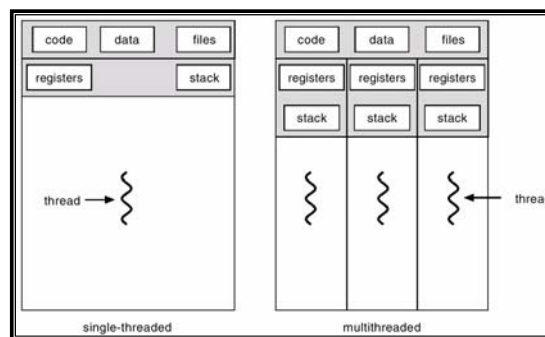
- can save unnecessary duplication of data, files, and other resources  
(no OS support needed for shared data/communication)

### economy

- since a thread shares code, data, and files, much less overhead than creating processes

### concurrency

- if multiple processors, can split a process into threads and execute in parallel



20

## Issues concerning threads

- context switching between threads of different processes is as time consuming as any process context switch, since no sharing
- threads can be implemented within the OS (kernel implementation) or as a separate set of functions (user implementation)
- kernel implementation of threads implies that the OS schedules threads, which can lead to uneven CPU access
- thread programming is more difficult, since the programmer must think about concurrent operations
- sharing amongst threads introduces some security issues