Processes

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

UNIVERSITY

Today's Topics

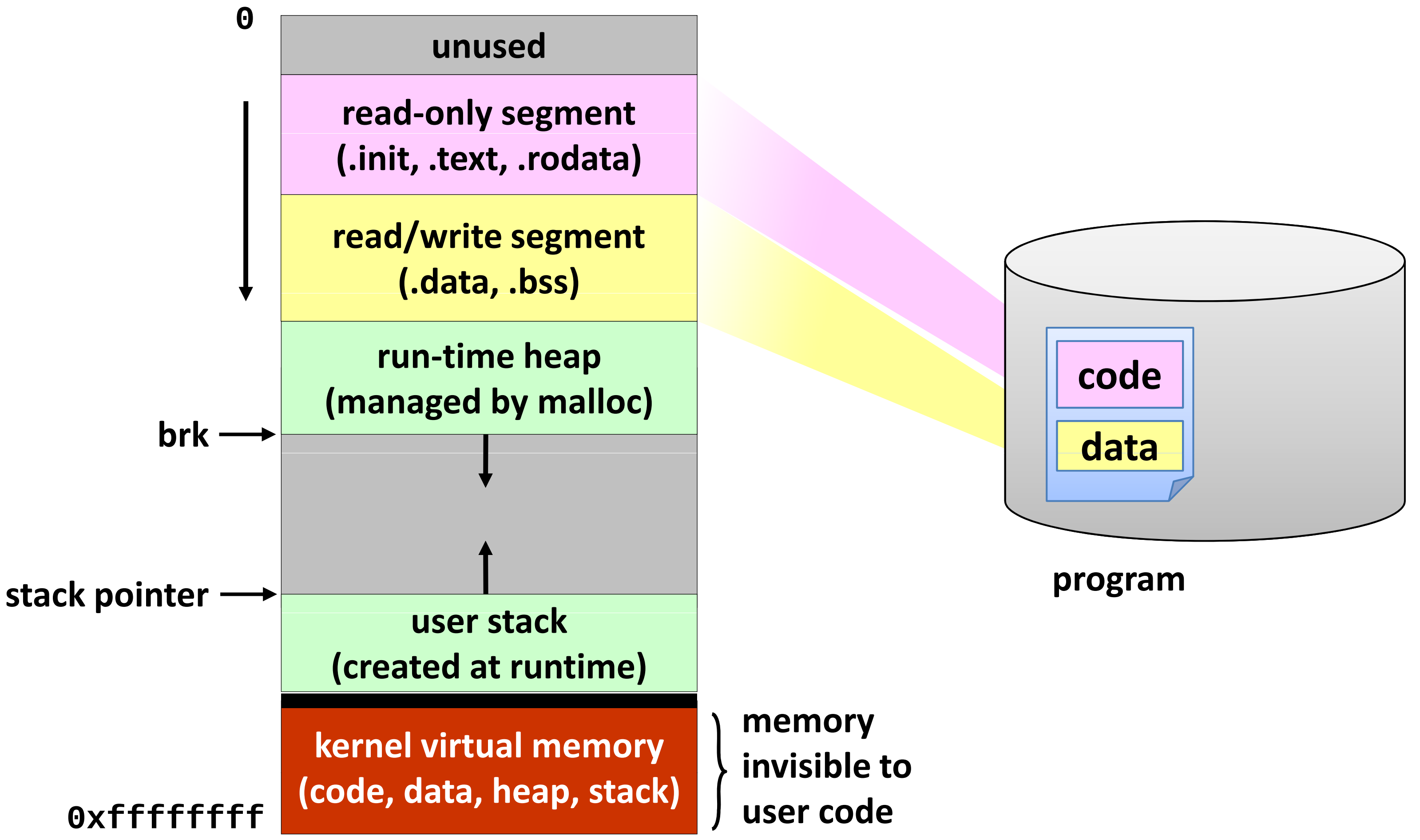
- What is the process?
- How to implement processes?
- Inter-Process Communication (IPC)

Process Concept (1)

- **What is the process?**
  - An instance of a program in execution.
  - An encapsulation of the flow of control in a program.
  - A dynamic and active entity.
  - The basic unit of execution and scheduling.
  - A process is named using its process ID (PID).
  - Job, task, or sequential process
  - A process includes:
    - CPU contexts (registers)
    - OS resources (memory, open files, etc.)
    - Other information (PID, state, owner, etc.)

Process Concept (2)

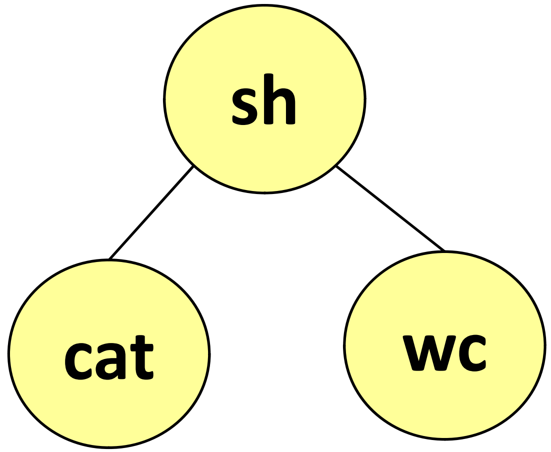
- Process in memory



Process Creation (1)

- **Process hierarchy**
  - One process can create another process: parent-child relationship
  - UNIX calls the hierarchy a "process group"
  - Windows has no concept of process hierarchy.
  - Browsing a list of processes:
    - ps in UNIX
    - taskmgr (Task Manager) in Windows

```
$ cat file1 | wc
```



Process Creation (2)

- **Process creation events**
  - Calling a system call
    - fork() in POSIX, CreateProcess() in Win32
    - Shells or GUIs use this system call internally.
  - System initialization
    - init process
- **Background processes**
  - Do not interact with users
  - Daemons

Process Creation (3)

- **Resource sharing**
  - Parent may inherit all or a part of resources and privileges for its children
    - UNIX: User ID, open files, etc.
- **Execution**
  - Parent may either wait for it to finish, or it may continue in parallel.
- **Address space**
  - Child duplicates the parent's address space or has a program loaded into it.

Process Termination

- **Process termination events**
  - Normal exit (voluntary)
  - Error exit (voluntary)
  - Fatal error (involuntary)
    - Exceed allocated resources
    - Segmentation fault
    - Protection fault, etc.
  - Killed by another process (involuntary)
    - By receiving a signal

fork()

```
#include <sys/types.h>
#include <unistd.h>

int main()
{
     int pid;

     if ((pid = fork()) == 0)
        /* child */
        printf (“Child of %d is %d\n”,
                getppid(), getpid());
     else
        /* parent */
        printf (“I am %d. My child is %d\n”,
                getpid(), pid);
}
```

fork(): Example Output

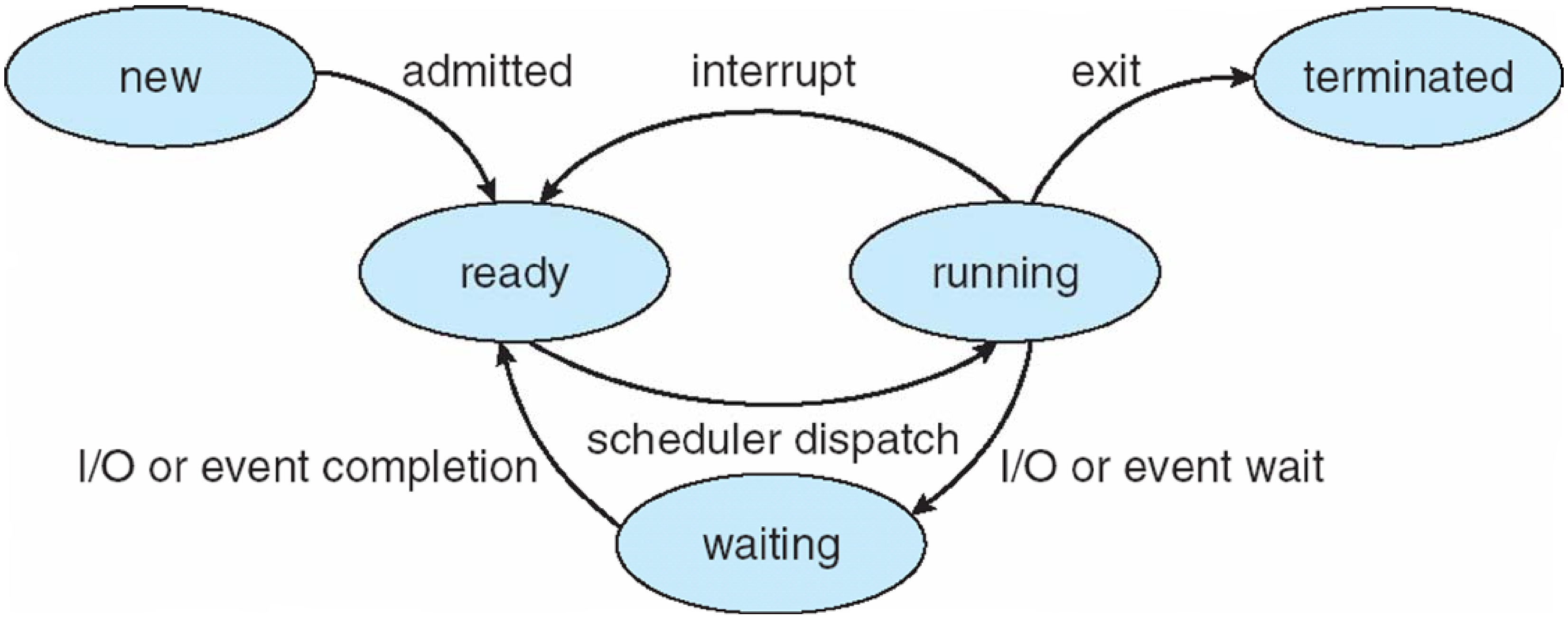
```
% ./a.out
I am 31098. My child is 31099.
Child of 31098 is 31099.

% ./a.out
Child of 31100 is 31101.
I am 31100. My child is 31101.
```

Simplified UNIX Shell

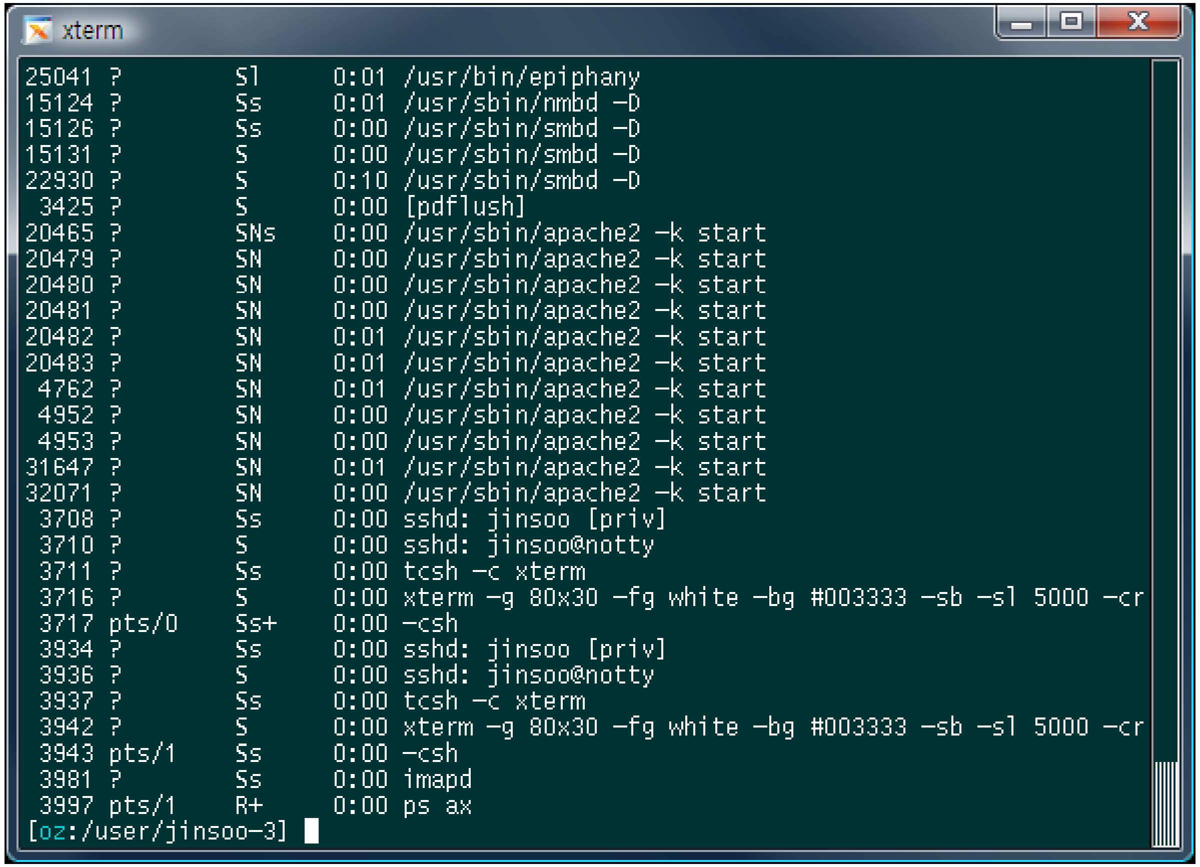
```
int main()
{
     while (1) {
          char *cmd = read_command();
          int pid;
          if ((pid = fork()) == 0) {
               /* Manipulate stdin/stdout/stderr for
                   pipes and redirections, etc. */
               exec(cmd);
               panic("exec failed!");
          } else {
               wait (pid);
          }
     }
}
```

Process State Transition (1)



Process State Transition (2)

- **Linux example**



R: Runnable
S: Sleeping
T: Traced or Stopped
D: Uninterruptible Sleep
Z: Zombie
<: High-priority task
N: Low-priority task
s: Session leader
+: In the foreground process group
l: Multi-threaded

Process Data Structures

- **PCB (Process Control Block)**
  - Each PCB represents a process.
  - Contains all of the information about a process
    - Process state
    - Program counter
    - CPU registers
    - CPU scheduling information
    - Memory management information
    - Accounting information
    - I/O status information, etc.
  - task_struct in Linux
    - 1456 bytes as of Linux 2.4.18

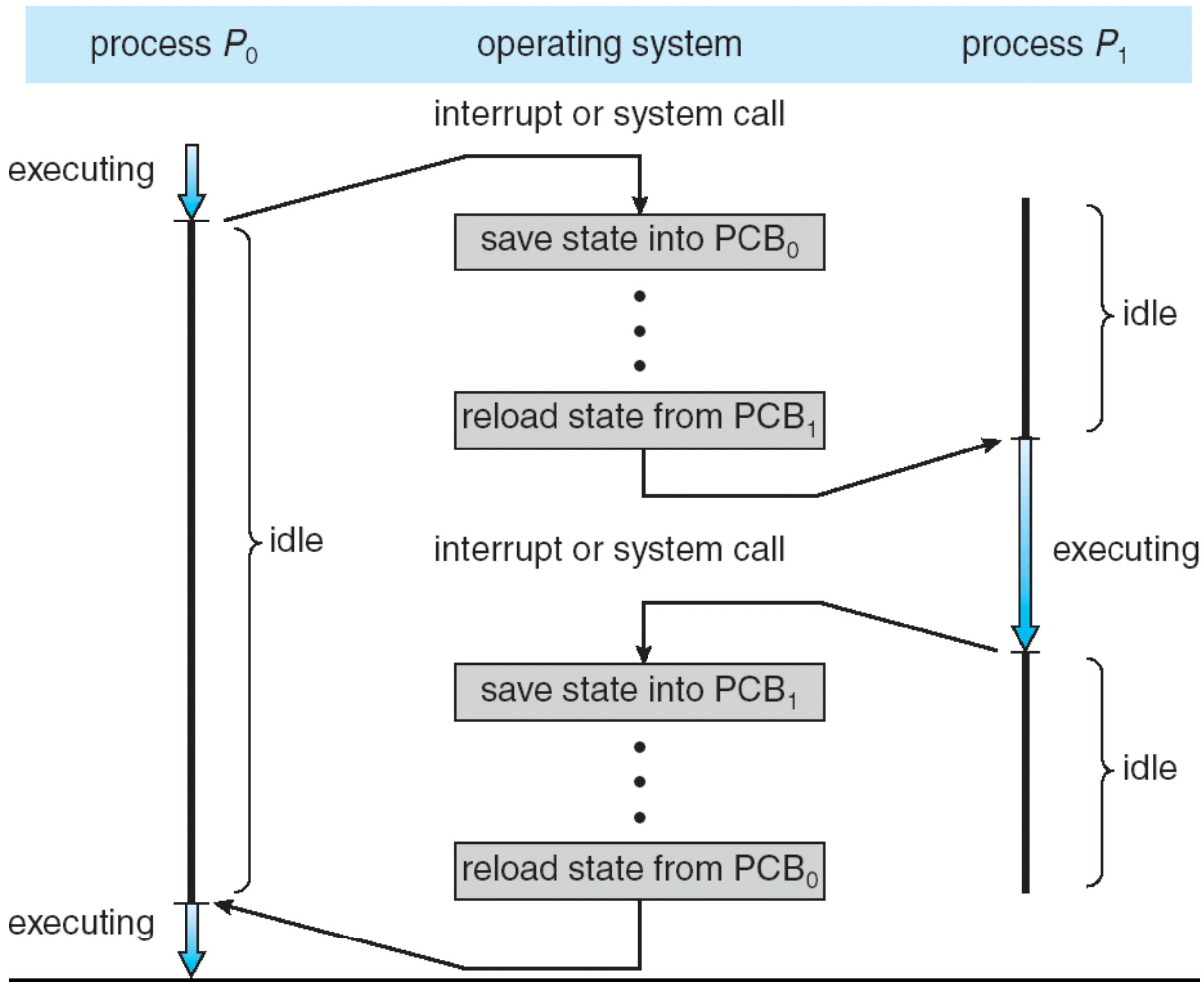
Process Control Block (PCB)

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

PCBs and Hardware State

- **When a process is running:**
  - Its hardware state is inside the CPU:
    PC, SP, registers
- **When the OS stops running a process:**
  - It saves the registers' values in the PCB.
- **When the OS puts the process in the running state:**
  - It loads the hardware registers from the values in that process' PCB.

Context Switch (1)

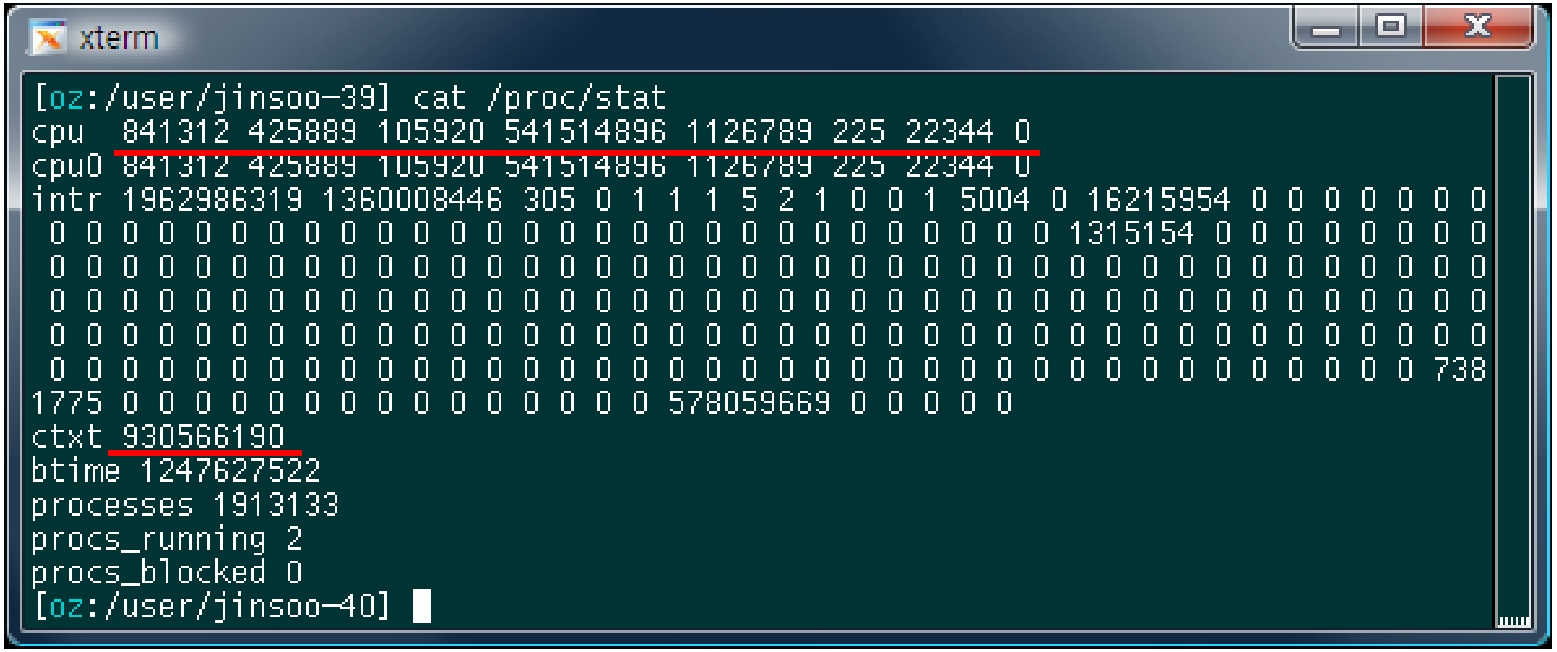


Context Switch (2)

- **Context switch (or process switch)**
  - The act of switching the CPU from one process to another.
  - Administrative overhead
    - saving and loading registers and memory maps
    - flushing and reloading the memory cache
    - updating various tables and lists, etc.
  - Context switch overhead is dependent on hardware support.
    - Multiple register sets in UltraSPARC.
    - Advanced memory management techniques may require extra data to be switched with each context.
  - 100s or 1000s of switches/s typically.

Context Switch (3)

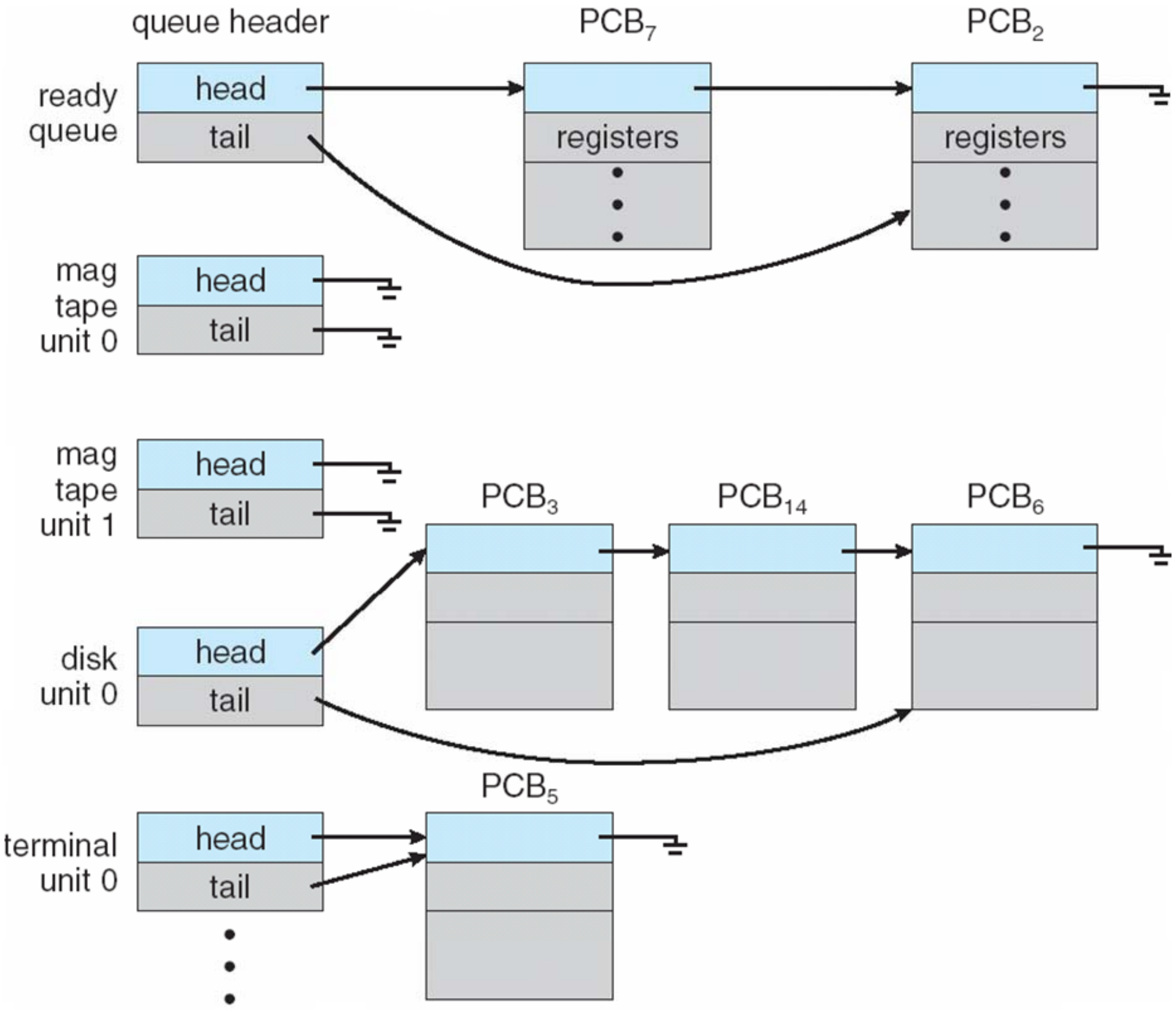
- **Linux example**
  - Total 544,037,375 user ticks = 1511 hours = 63.0 days
  - Total 930,566,190 context switches
  - Roughly 86 context switches / sec (per CPU)



Process State Queues (1)

- **State queues**
  - The OS maintains a collection of queues that represent the state of all processes in the system
    - Job queue
    - Ready queue
    - Wait queue(s): there may be many wait queues, one for each type of wait (device, timer, message, …)
  - Each PCB is queued onto a state queue according to its current state.
  - As a process changes state, its PCB is migrated between the various queues.

Process State Queues (2)



Process State Queues (3)

- **PCBs and state queues**
  - PCBs are data structures
    - dynamically allocated inside OS memory
  - When a process is created:
    - OS allocates a PCB for it
    - OS initializes PCB
    - OS puts PCB on the correct queue
  - As a process computes:
    - OS moves its PCB from queue to queue
  - When a process is terminated:
    - OS deallocates its PCB

Process Creation: UNIX (1)

```
int fork()
```

- **fork()**
  - Creates and initializes a new PCB
  - Creates and initializes a new address space
  - Initializes the address space with a copy of the entire contents of the address space of the parent.
  - Initializes the kernel resources to point to the resources used by parent (e.g., open files)
  - Places the PCB on the ready queue.
  - Returns the child's PID to the parent, and zero to the child.

Process Creation: UNIX (2)

```
int exec (char *prog, char *argv[])
```

- **exec()**
  - Stops the current process
  - Loads the program "prog" into the process' address space.
  - Initializes hardware context and args for the new program.
  - Places the PCB on the ready queue.
    - Note: exec() does not create a new process.
  - What does it mean for exec() to return?

Process Creation: NT

```
BOOL CreateProcess (char *prog, char *args, …)
```

- **CreateProcess()**
  - Creates and initializes a new PCB
  - Creates and initializes a new address space
  - Loads the program specified by "prog" into the address space
  - Copies "args" into memory allocated in address space
  - Initializes the hardware context to start execution at main
  - Places the PCB on the ready queue

Why fork()?

- **Very useful when the child…**
  - is cooperating with the parent.
  - relies upon the parent's data to accomplish its task.
  - Example: Web server

```
While (1) {
      int sock = accept();
      if ((pid = fork()) == 0) {
            /* Handle client request */
      } else {
            /* Close socket */
      }
}
```

Inter-Process Communications

- **Inside a machine**
  - Pipe
  - FIFO
  - Shared memory
  - Sockets

- **Across machines**
  - Sockets
  - RPCs (Remote Procedure Calls)
  - Java RMI (Remote Method Invocation)

Projects (1)

- **Plan**
  - We will use the Pintos educational operating system
    - Developed by Stanford University (Some source files are derived from code used in the MIT OS course)
    - A real, bootable OS for 80x86 architecture
    - The original structure was inspired by the Nachos educational OS (Java-based)
    - Written in C language (with minimal assembly code)
  - Platform: Linux + PC emulators (bochs or qemu)
  - Group projects: in teams of 3 students
  - More on Pintos will be coming up soon

Projects (2)

- **Action items**
  - Form a project team
    - We have 27 enrolled students, so there will be 9 teams
  - Send me an e-mail by the next class including
    - The name of your team
    - The list of team members (name & e-mail address)
  - Each student will be invited by the mailing list: skku-pintos-project@googlegroups.com
    - Project-related discussions will be done via this mailing list
    - Once you accept the invitation, you can send e-mail to everyone on the list
    - http://groups.google.com/group/skku-pintos-project
  - Prepare your own Linux platform

Projects (3)

- **Lab session**
  - For project assignments, discussions, & demos

	Mon	Tue	Wed	Thu	Fri
12:00 – 13:00					
13:00 – 14:00					
14:00 – 15:00					
15:00 – 16:00					
16:00 – 17:00					
17:00 – 18:00					
18:00 – 19:00					
19:00 – 20:00				Lab	
20:00 – 21:00				Session	
21:00 – 22:00					