Operating Systems 2023 Spring Term
Week 2

Dr. Emrah İnan (emrahinan@iyte.edu.tr)

Processes

March 16, 2023

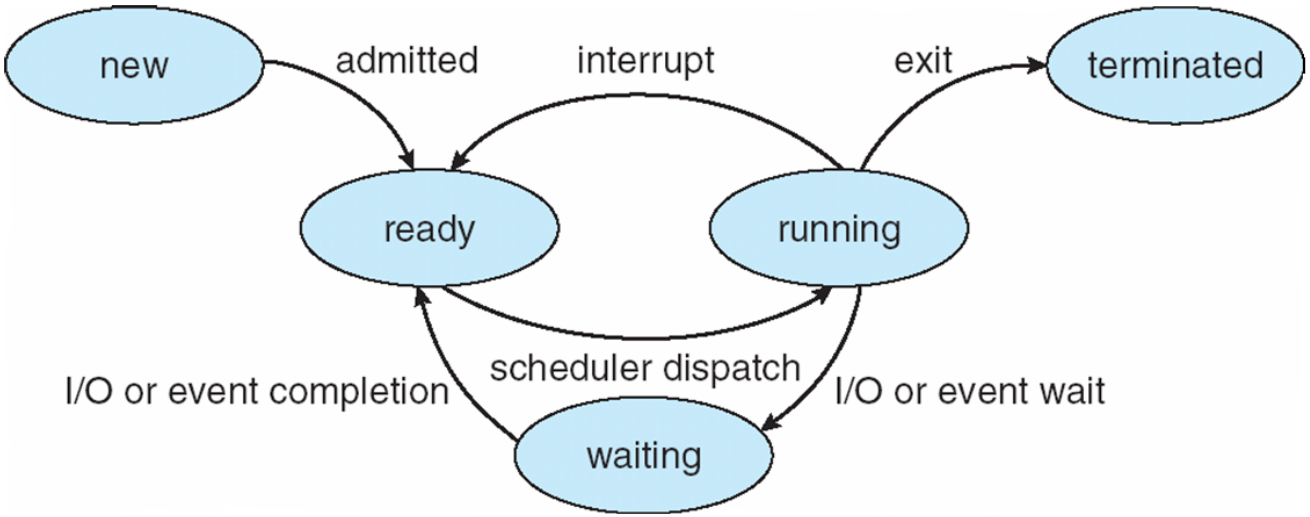
Week 1: Sample Glossary

- **Operating system:** a program that manages a computer's hardware, provides a basis for application programs, and acts as an intermediary between the computer user and the computer hardware (on Page 1260)
- **Kernel:** The operating system component running on the computer at all times after system boot (on Page 1254)
- **booting:** The procedure of starting of a computer by loading the kernel (on Page 1240)
- **Interrupt:** A hardware mechanism that enables a device to notify the CPU that it needs attention. (on Page 1253)

Process Concept

- An operating system executes a variety of programs: (1) Batch system – jobs (2) Time-shared systems – user programs or tasks
- **Process:** a program in execution; process execution must progress in sequential fashion
- Program is passive entity stored on disk (executable file), process is active -> Program becomes process when executable file loaded into memory (via GUI mouse clicks, command line entry of its name, etc)
- Multiple parts
  - The program code, also called text section
  - Current activity including program counter, processor registers
  - Stack containing temporary data
  - Function parameters, return addresses, local variables
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time

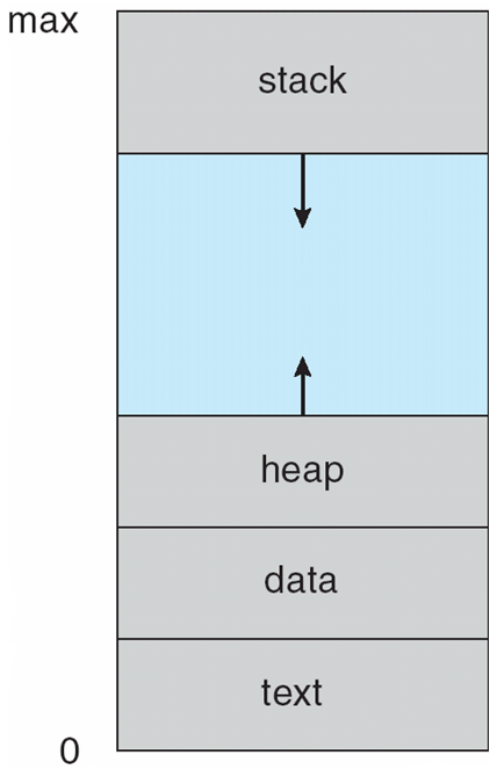
Diagram of Process State



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

Process in Memory



When would I want to use a heap?[1]

147

Use it whenever you need quick access to the largest (or smallest) item, because that item will always be the first element in the array or at the root of the tree.

However, the remainder of the array is kept partially unsorted. Thus, instant access is only possible to the largest (smallest) item. Insertions are fast, so it's a good way to deal with incoming events or data and always have access to the earliest/biggest.

Useful for priority queues, schedulers (where the earliest item is desired), etc...

A heap is a tree where a parent node's value is larger than that of any of its descendant nodes.

If you think of a heap as a binary tree stored in linear order by depth, with the root node first (then the children of that node next, then the children of those nodes next); then the children of a node at index N are at 2N+1 and 2N+2. This property allows quick access-by-index. And since heaps are manipulated by swapping nodes, this allows for in-place sorting.

Share Improve this answer Follow

edited Jul 28, 2015 at 22:28
Community Bot
1 • 1

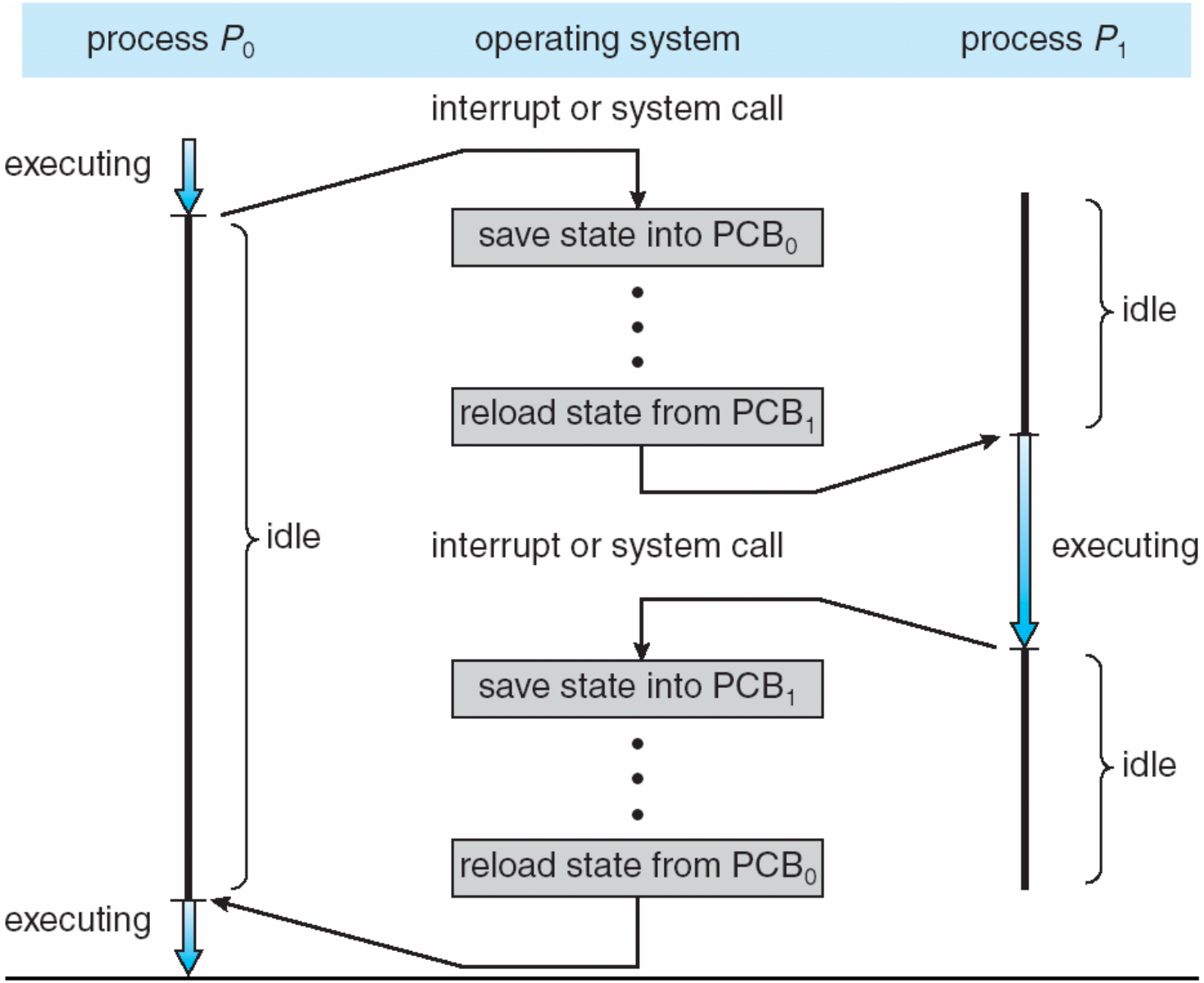
answered Apr 14, 2009 at 20:22
Joe Koberg
24.9k • 6 • 48 • 54

[1] https://stackoverflow.com/questions/749199/when-would-i-want-to-use-a-heap

Process Control Block (PCB)

- Information associated with each process (also called task control block)
- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

CPU Switch From Process to Process



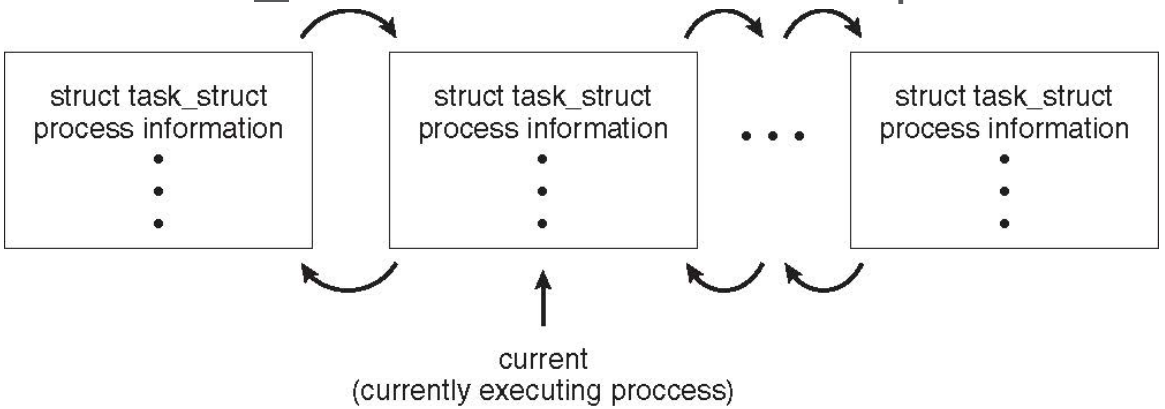
Threads

- Process has a single thread of execution
- Consider having multiple program counters per process -> Multiple locations can execute at once (Multiple threads of control -> threads)
- Must then have storage for thread details, multiple program counters in PCB (next chapter in detail)

Process Representation in Linux

Represented by the C structure task_struct

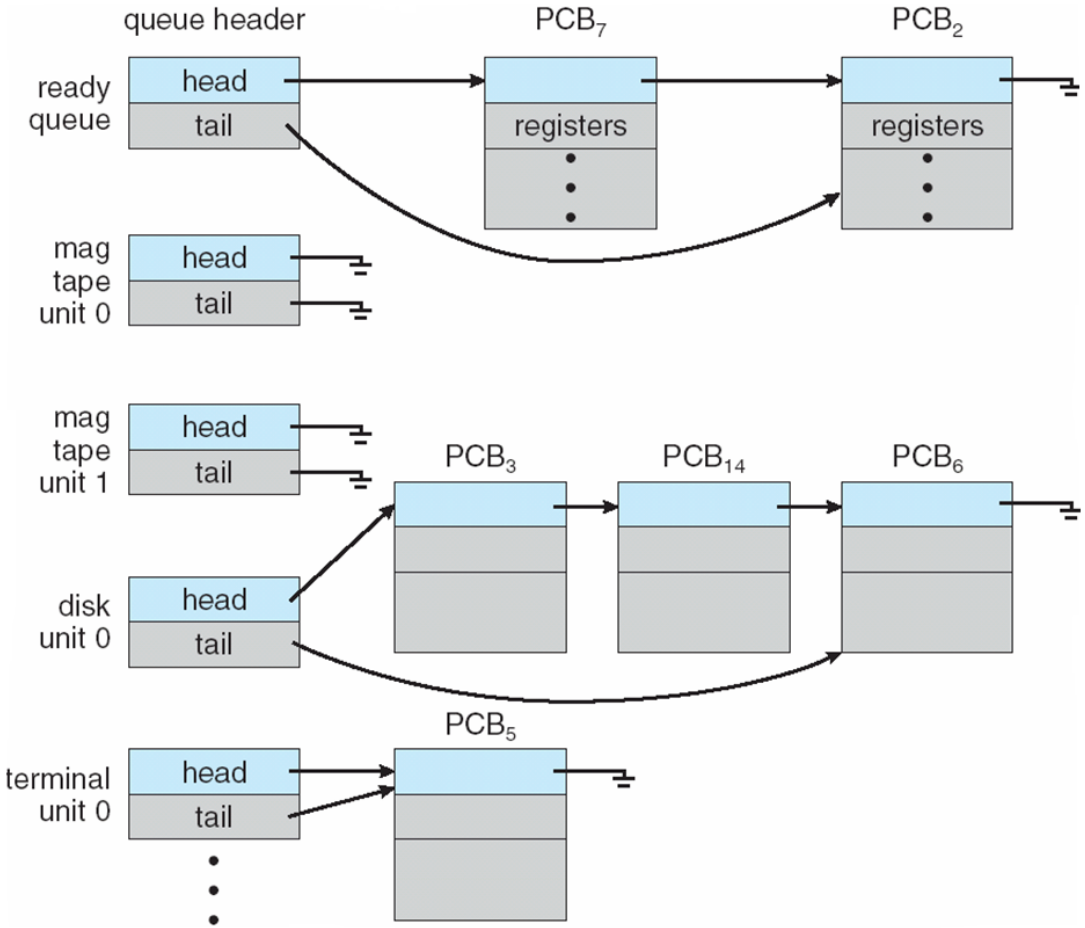
```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



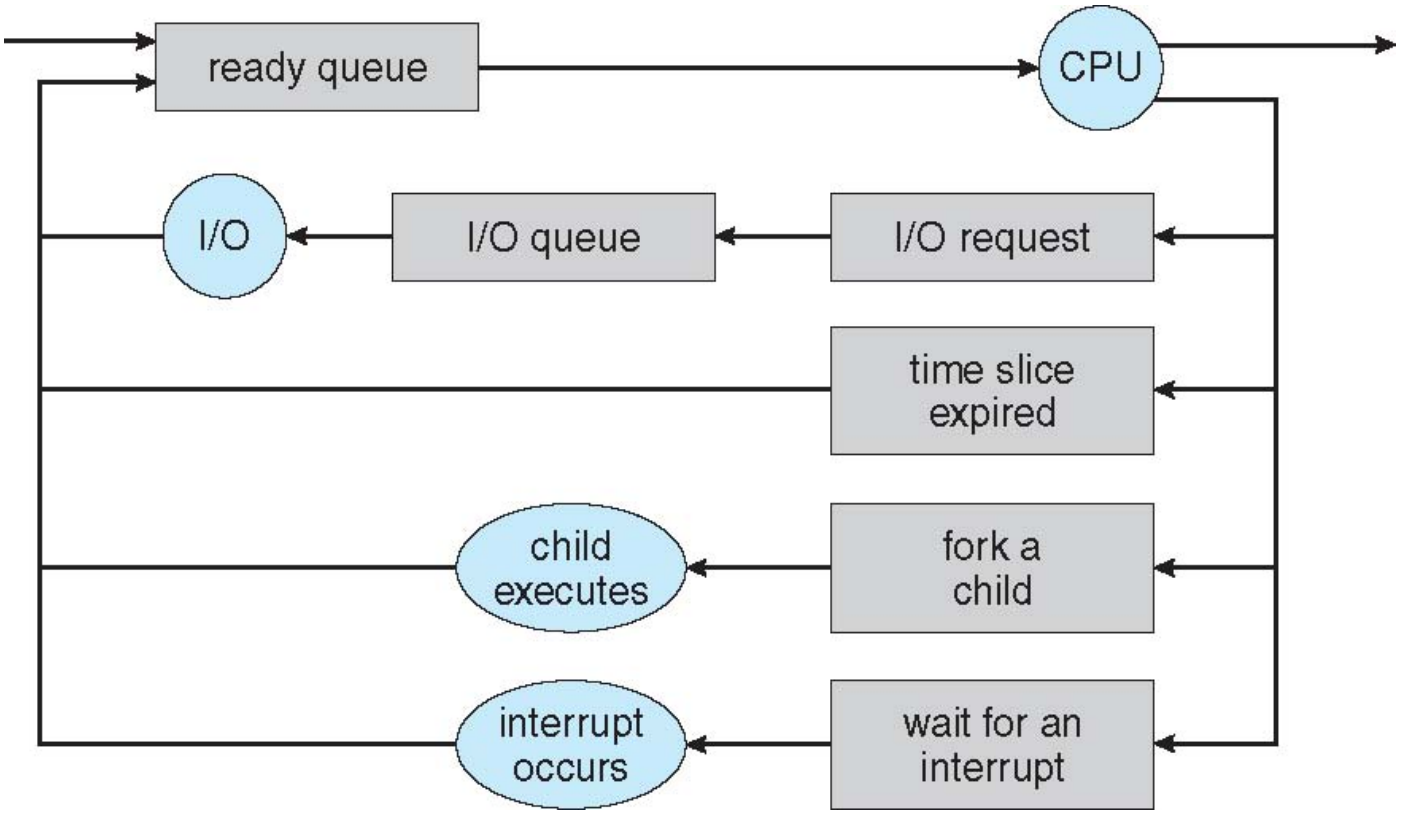
Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- Process scheduler selects among available processes for next execution on CPU
- Maintains scheduling queues of processes
  - 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

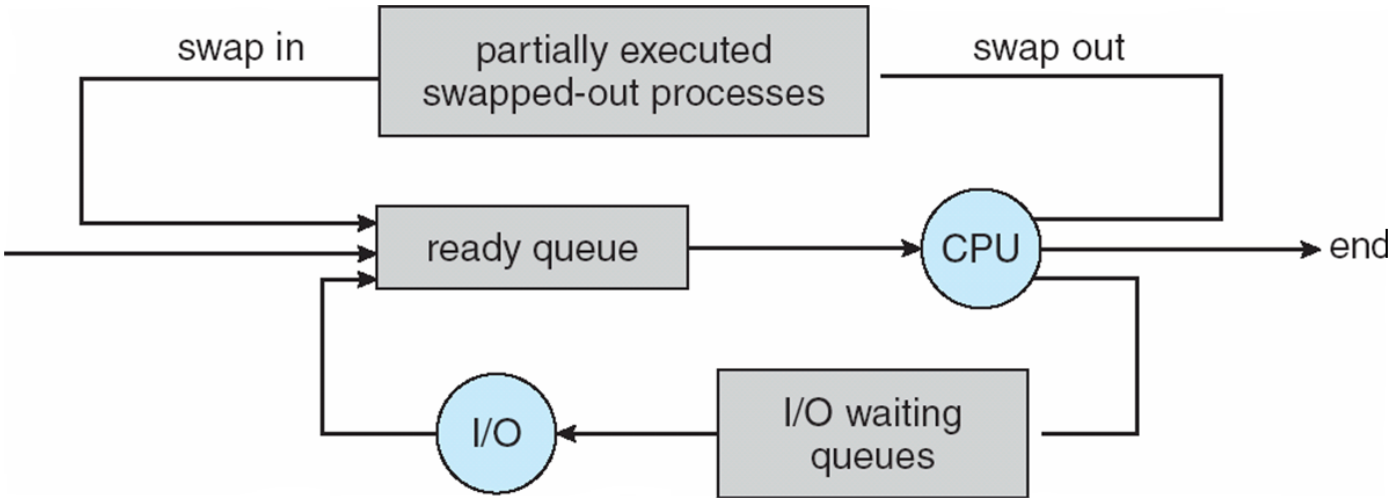


Schedulers

- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) -> (must be fast)
- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) -> (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) or CPU-bound process (spends more time doing computations)

Medium Term Scheduling

- Medium-term scheduler can be added if degree of multiple programming needs to decrease
- Remove process from memory, store on disk, bring back in from disk to continue execution: swapping



Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single foreground process- controlled via user interface
  - Multiple background processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a service to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

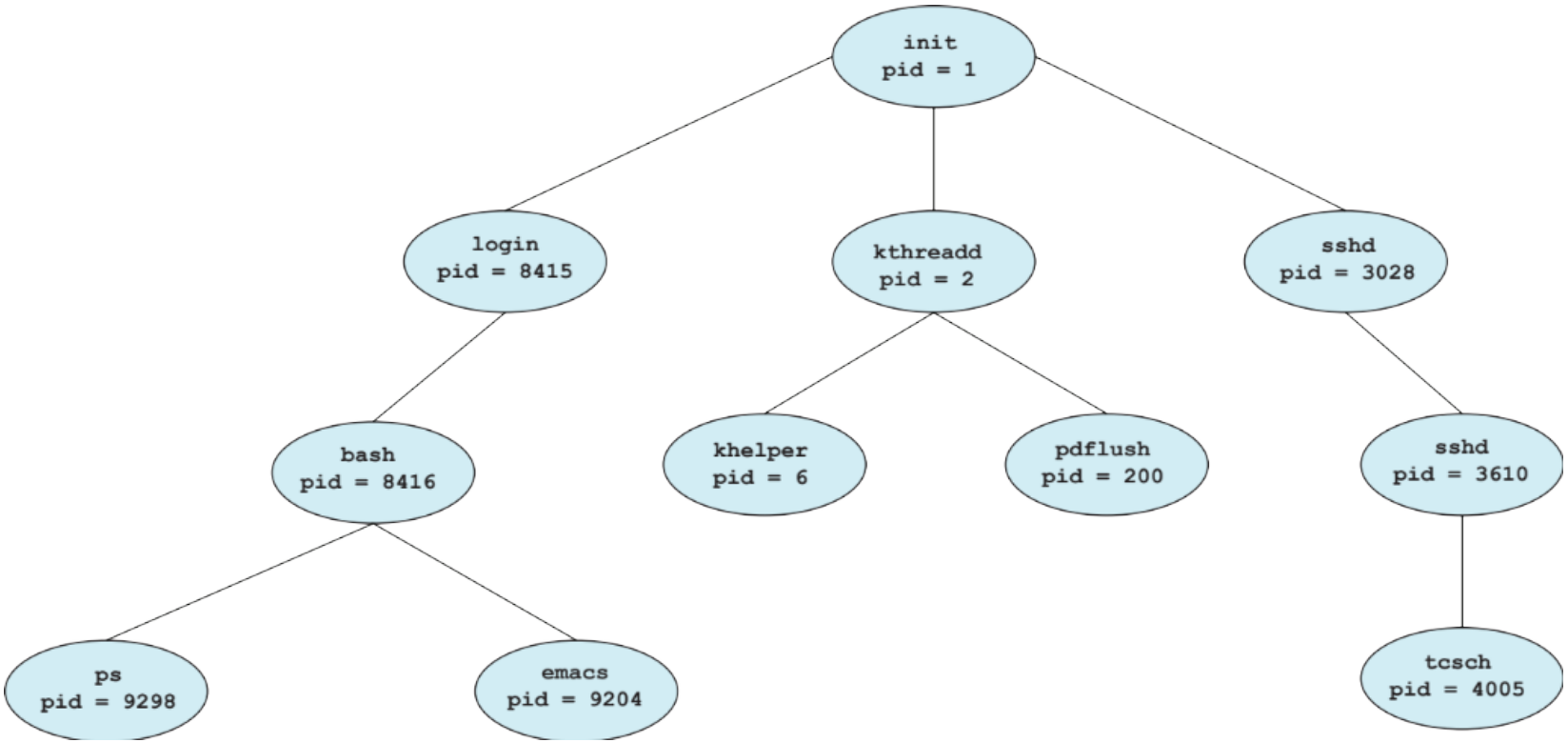
Context Switch

- 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 -> The more complex the OS and the PCB -> the longer the context switch
- Time dependent on hardware support Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once

Process Creation I

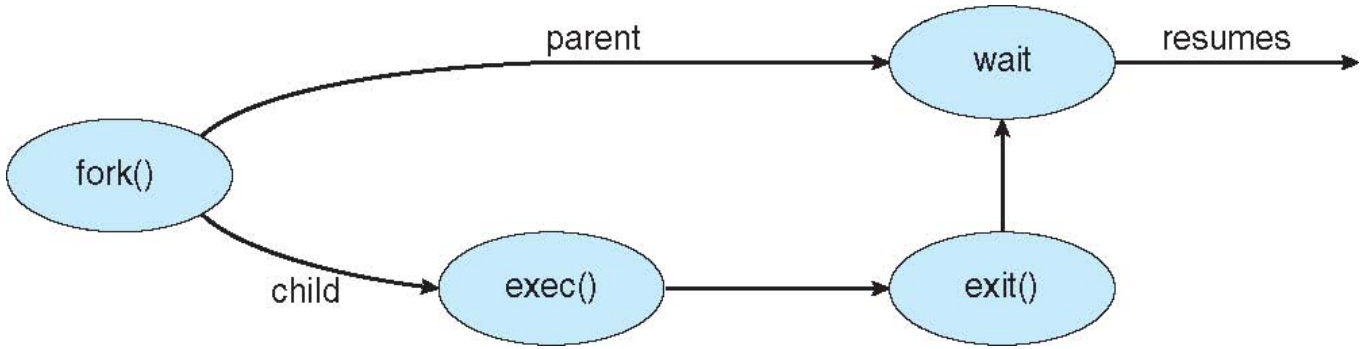
- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options: (i) Parent and children execute concurrently (ii) Parent waits until children terminate

A Tree of Processes in Linux



Process Creation II

- Address space: (i) Child duplicate of parent (iİ) 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



C Program Forking Separate Process

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

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 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");
    }

    return 0;
}
```

Creating a Separate Process via Windows API

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Process Termination I

- Process executes last statement and then asks the operating system to delete it using the exit() system call.
  - Returns status data from child to parent (via wait())
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the abort() system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination II

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - cascading termination. All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- he parent process may wait for termination of a child process by using the wait()system call. The call returns status information and the pid of the terminated process (pid = wait(status);)
- If no parent waiting (did not invoke wait()) process is a zombie
- If parent terminated without invoking wait , process is an orphan

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

Examples of Windows and Unix System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

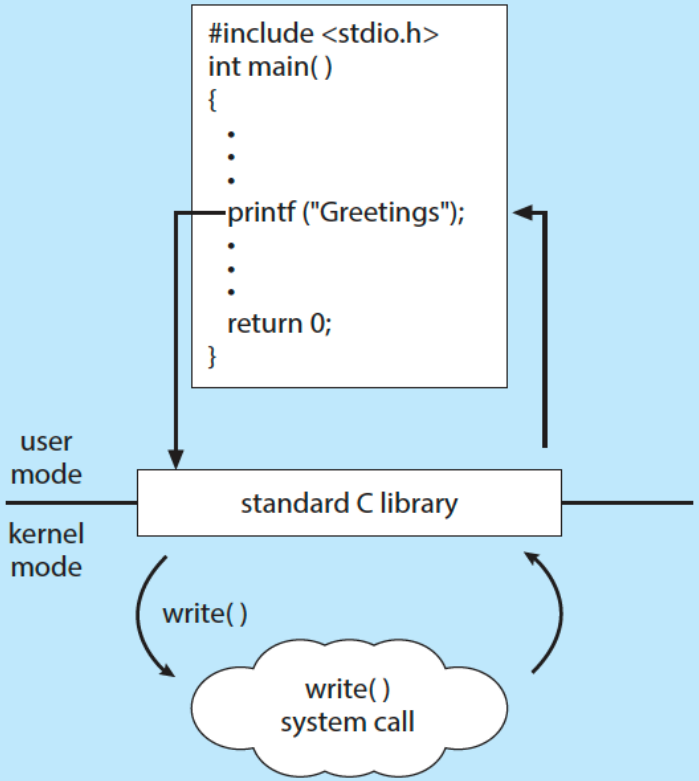
The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File management	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device management	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	`write()`
Information maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communications	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shm_open()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitlializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

Standard C Library Example

THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do) -> If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - Browser process manages user interface, disk and network I/O
  - Renderer process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened -> Runs in sandbox restricting disk and network I/O, minimizing effect of security exploits
  - Plug-in -> process for each type of plug-in