# Process

Heechul Yun

Disclaimer: some slides are adopted from the book authors' slides with permission

# Recap

- OS services
  – Resource (CPU, memory) allocation, filesystem, communication, protection, security, I/O operations
- OS interface
  – System-call interface
- OS structure
  – Monolithic , microkernel
  – Loadable module

# Roadmap

- Beginning of a series of **important** topics:
  - Process
  - Thread
  - Synchronization

- Today
  - Process concept
  - Context switching

# Process

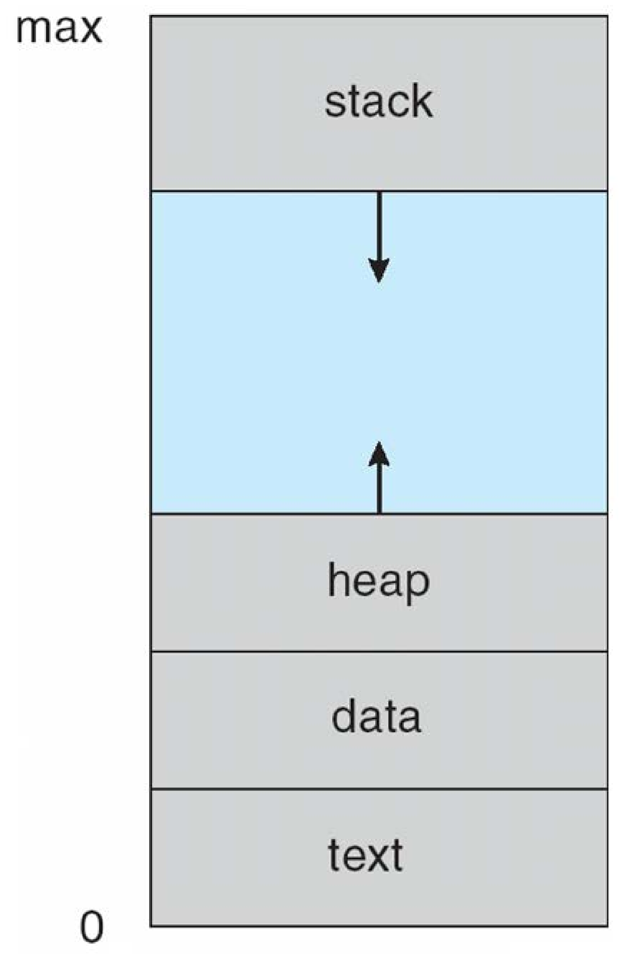
- Process
  - An OS **abstraction** represents a **running** application

- Three main components
  - **Address space**
    - The process's view of memory
    - Includes program code, global variables, dynamic memory, stack
  - **Processor state**
    - Program counter (PC), stack pointer, and other CPU registers
  - **OS resources**
    - Various OS resources that the process uses
    - E.g.) open files, sockets, accounting information

KU THE UNIVERSITY OF KANSAS

# Process Address Space

- Text
  - Program code
- Data
  - Global variables
- Heap
  - Dynamically allocated memory
    - i.e., Malloc()
- Stack
  - Temporary data
  - Grow at each function call

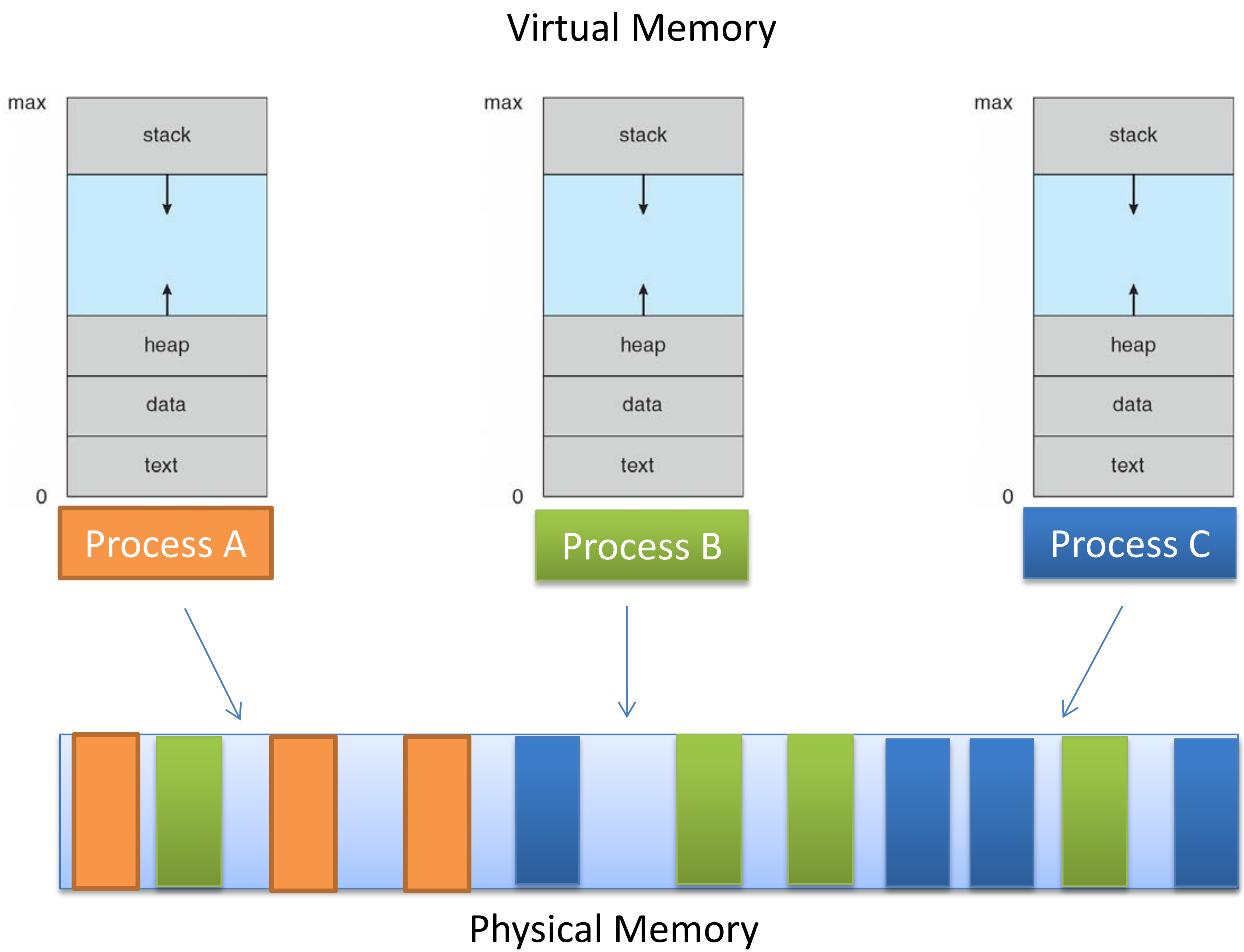


# Process Address Space

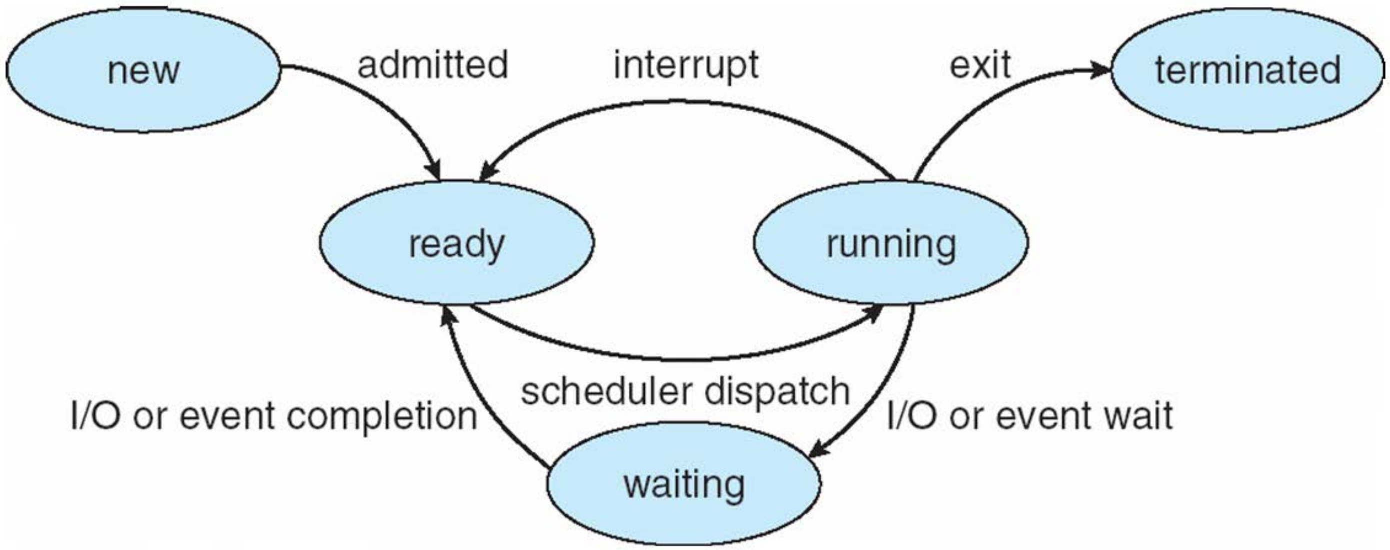
- Each process has its own **private** address space
  - $2^{32}$ (4GB) of **continuous memory** in a 32bit machine
  - Each has same address range (e.g., 0x0 ~ 0xffffffff)
  - How is this possible?
    - What if you have less than 4GB physical DRAM?
    - What if you have 100 processes to run?

- **Virtual memory**
  - An OS mechanism providing this **illusion**
  - We will study it in great detail later in the $2^{nd}$ half of the semester

# Virtual Memory vs. Physical Memory



# Process State




- **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

# Process Control Block (PCB)

- Information associated with each process
  - Process id
  - Process state
    - running, waiting, etc.
  - Saved CPU registers
    - Register values saved on the last preemption
  - 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
  - OS resources
    - Open files, sockets, etc.

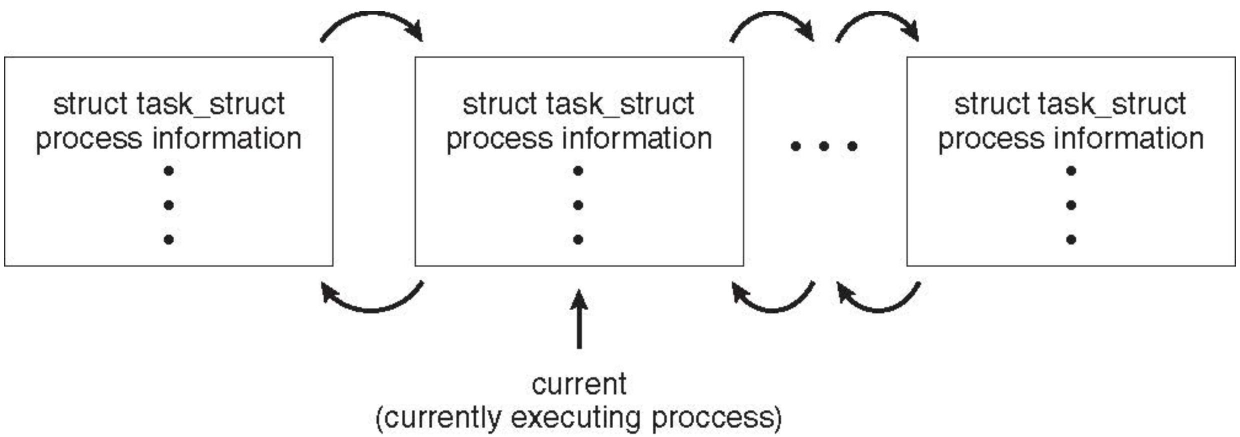
# Process in Linux

Represented by the C structure `task_struct (include/linux/sched.h)`

```
pid t_pid;                    /* process identifier */
long state;                   /* state of the process */
u64 vruntime;                 /* CFS scheduling information */
struct files_struct *files;/* list of open files */
struct mm_struct *mm;         /* address space of this process */
cputime_t utime, stime;       /* accounting information */
struct thread_struct thread; /* CPU states */
```

…

(very **big** structure: 5872 bytes in my desktop *)




(*) # cat /sys/kernel/slab/task_struct/object_size

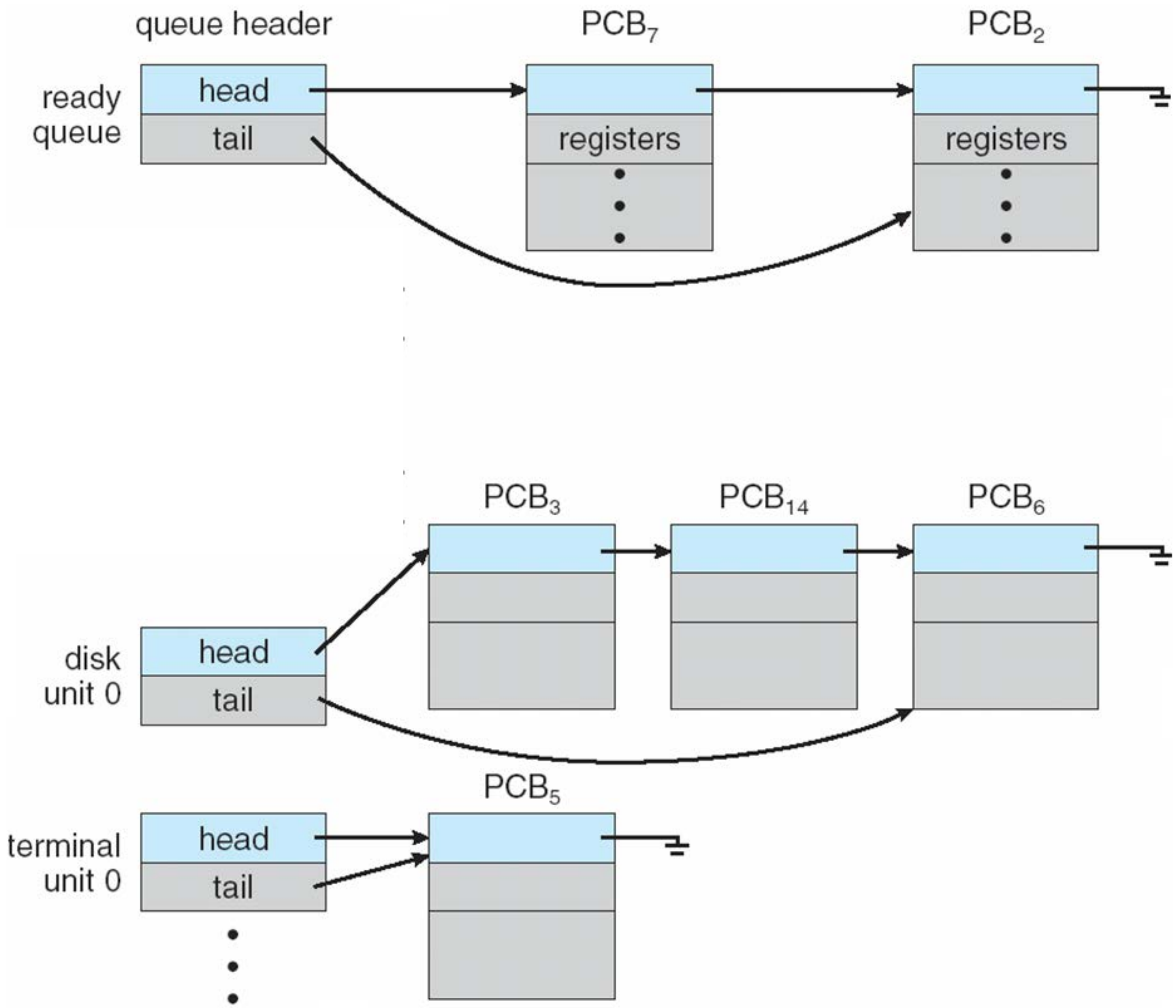
# Process Scheduling

- Decides which process to run **next**
  - Among **ready** processes

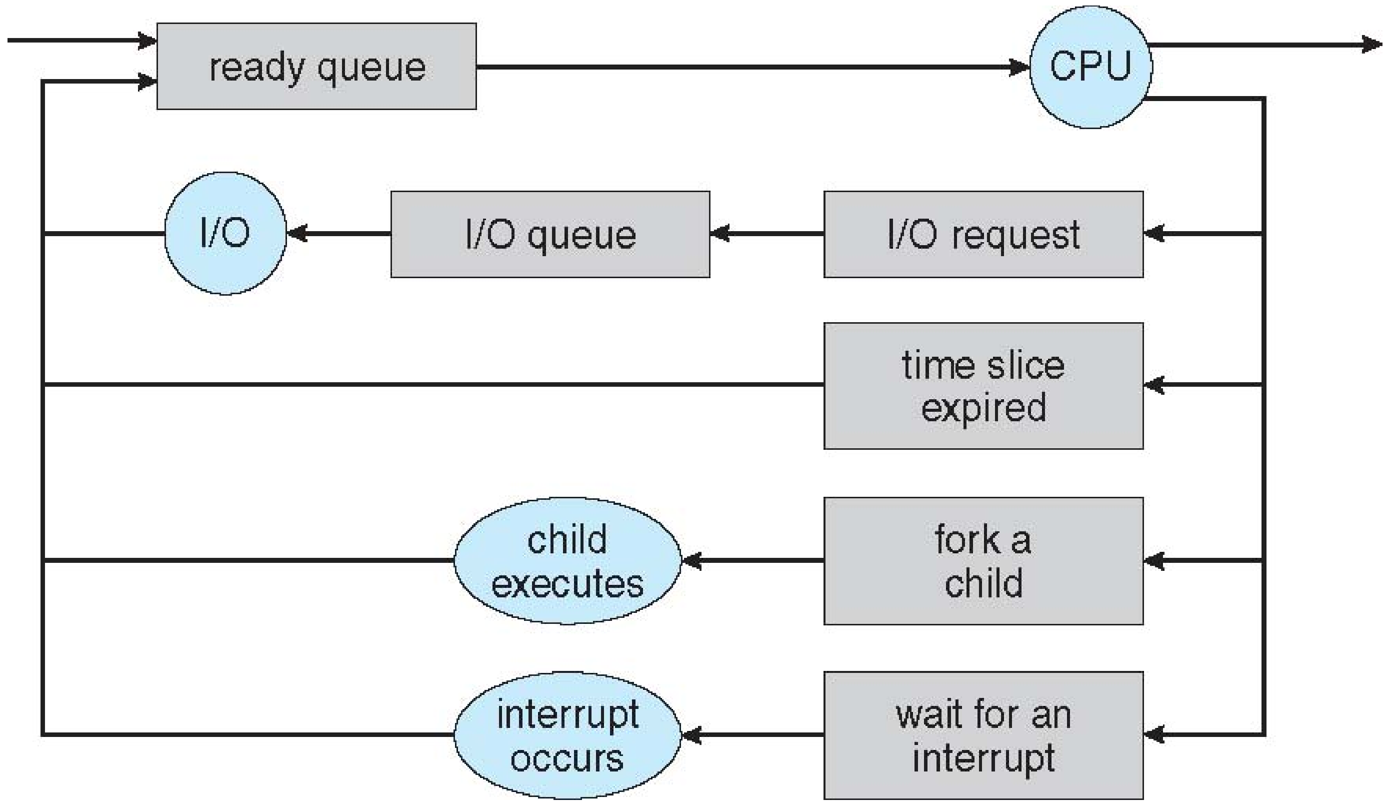
- We cover in much more detail later in the class
  - but let's get some **basics**

- OS maintains multiple **scheduling queues**
  - **Ready queue**
    - ready to be executed processes
  - **Device queues**
    - processes waiting for an I/O device
  - Processes **migrate** among the various queues

# Ready Queue and I/O Device Queues

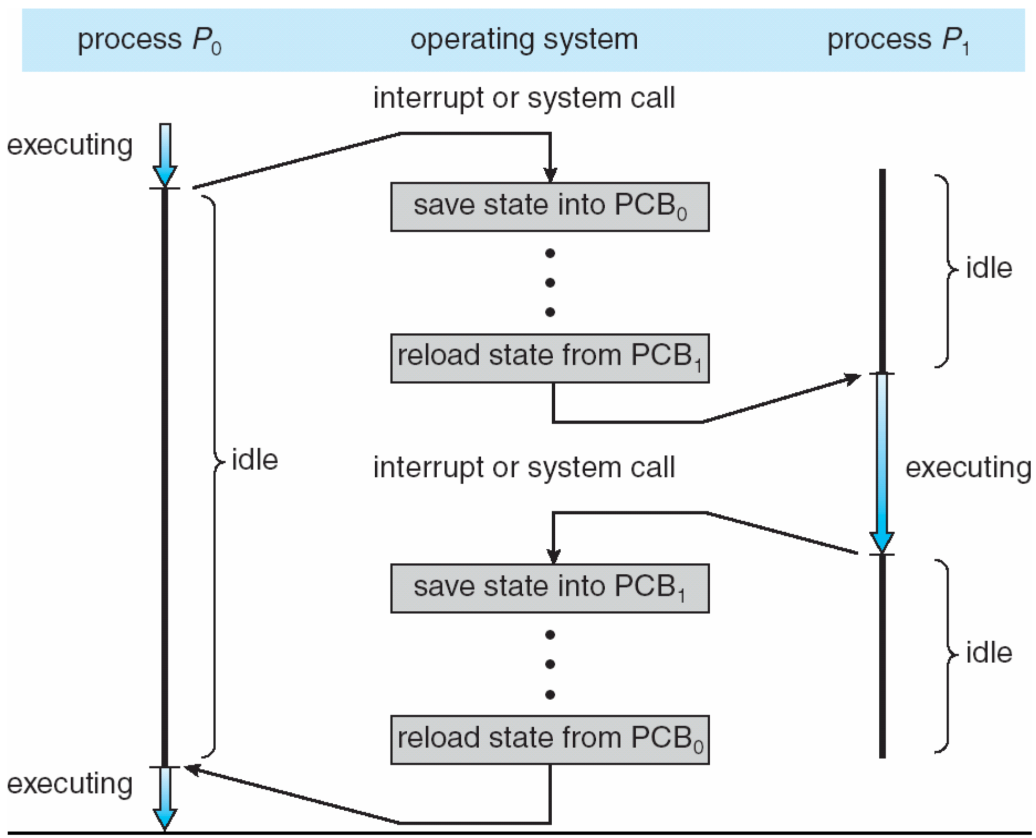


# Process Scheduling: Queuing Representation



# Context Switching

- Suspend the current process and resume a next one from its last suspended state



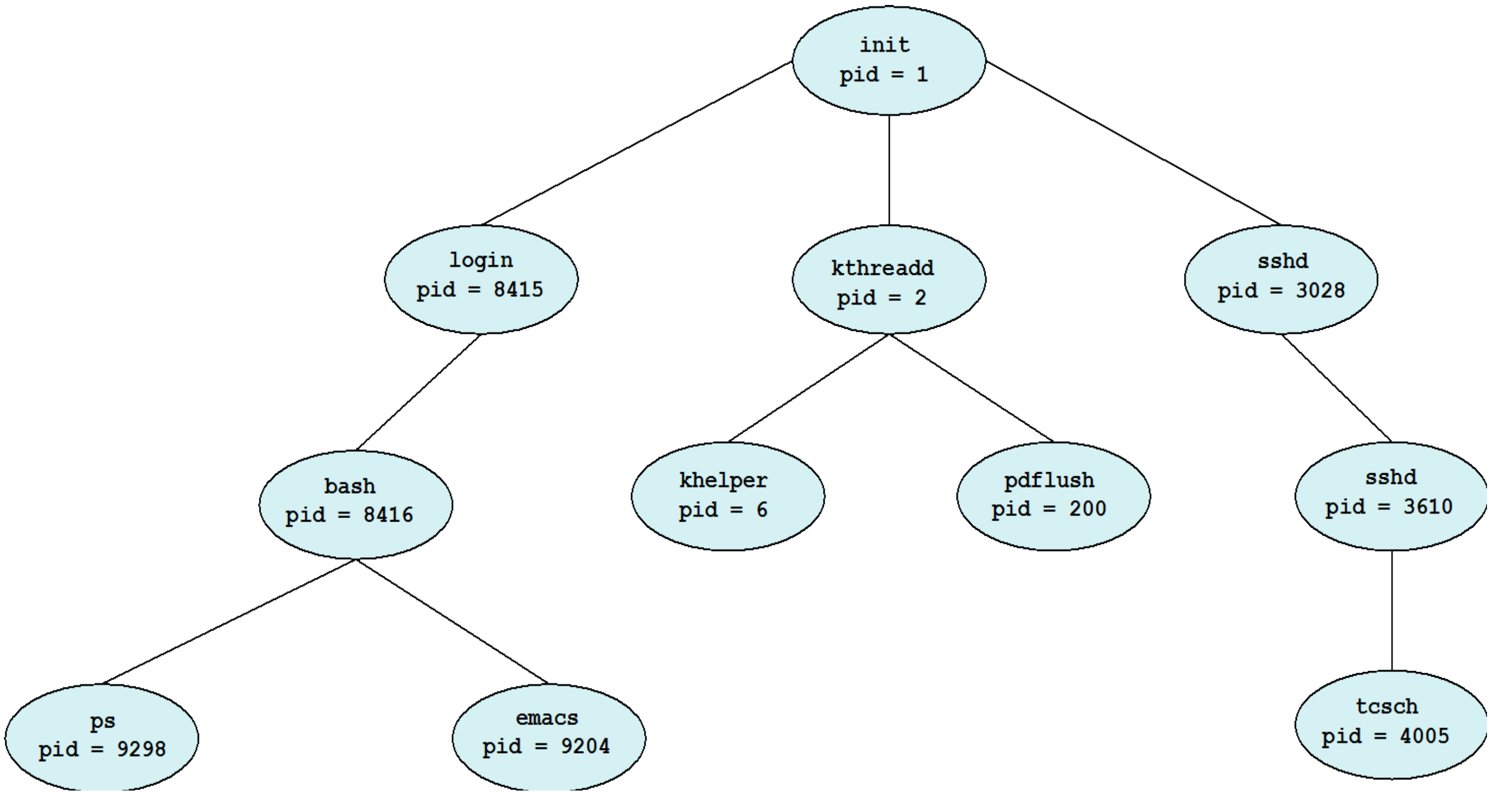
# Context Switching

- Overhead
  - Save and restore CPU states
  - Warm up instruction and data cache
    - Cache data of previous process is not useful for new process
- In Linux 3.6.0 on an Intel Xeon 2.8Ghz
  - About 1.8 us
  - ~ 5040 CPU cycles
  - ~ thousands of instructions

# Process Creation

- **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**)

# A Process Tree in Linux

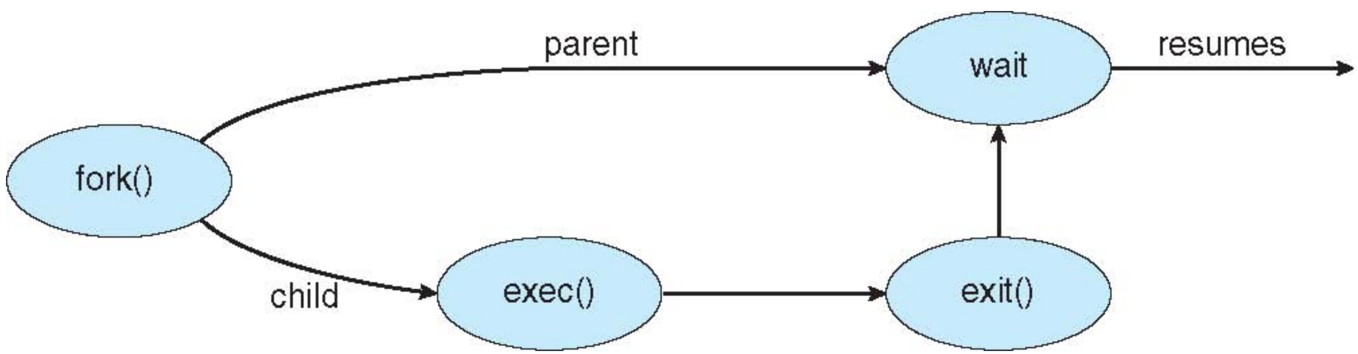


```
terminal
root@icecream:/ssd/Dropbox/Teaching/2014-Fall/EECS678# pstree
init─┬─NetworkManager─┬─dhclient
     │                ├─dnsmasq
     │                └─2*[{NetworkManager}]
     ├─accounts-daemon───{accounts-daemon}
     ├─acpid
     ├─atd
     ├─automount───3*[{automount}]
     ├─bluetoothd
     ├─colord───2*[{colord}]
     ├─console-kit-dae───64*[{console-kit-dae}]
     ├─cron
     ├─cupsd───dbus
     ├─dbus-daemon
     ├─dropbox───29*[{dropbox}]
     ├─5*[getty]
     ├─irqbalance
     ├─login───bash
     ├─login───bash───nmon
     ├─memcached───5*[{memcached}]
     ├─modem-manager
     ├─nscd───11*[{nscd}]
     ├─nslcd───5*[{nslcd}]
     ├─php5-fpm───6*[php5-fpm]
     ├─polkitd───{polkitd}
     ├─rpc.idmapd
     ├─rpc.statd
     ├─rpcbind
     ├─rsyslogd───3*[{rsyslogd}]
     ├─rtkit-daemon───2*[{rtkit-daemon}]
     ├─sh───initctl
     ├─sshd─┬─sshd───sshd
     │      └─sshd───sshd───bash─┬─emacs───bash
     │                           └─sudo───bash───pstree
     ├─udevd───2*[udevd]
     ├─udisks-daemon─┬─udisks-daemon
     │               └─2*[{udisks-daemon}]
     ├─upowerd───2*[{upowerd}]
     ├─upstart-socket-
     ├─upstart-udev-br
     ├─whoopsie───{whoopsie}
     ├─xrdp
     └─xrdp-sesman
root@icecream:/ssd/Dropbox/Teaching/2014-Fall/EECS678#
```

'pstree' output

# Process Creation

- 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



# Example: Forking a Process in UNIX

```
#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;
}
```

Child

Parent

# Example: Forking a Process in Windows

```
#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

- Normal termination via **`exit()`** system call.
  - Exit by itself.
  - Returns status data from child to parent (via **`wait()`**)
  - Process's resources are deallocated by operating system
- Forced termination via **`kill()`** system call
  - Kill someone else (child)
- **Zombie** process
  - If no parent waiting (did not invoke **`wait()`**)
- **Orphan** process
  - If parent terminated without invoking **`wait`**
  - **Q: who will be the parent of a orphan process?**
  - **A: Init process**