CS370 Operating Systems

Colorado State University
Yashwant K Malaiya

Fall 2016 Lecture 7

Slides based on

- **Text by Silberschatz, Galvin, Gagne**
- **Various sources**

CS370 OS Ch3 Processes

- Process Concept: a program in execution
- Process Scheduling
- Processes creation and termination
- Interprocess Communication using shared memory and message passing

FAQ

- Program control Block – What does it include and why? Where is it saved? another look
- Process Scheduling – Why? How? more soon
- How do multiple processes, that are part of a single program, interact. Inter-process communication details coming up.
- What happens to a process once it is finished? Resources deallocated, but only after ..
- If a process forks a child, what happens to the parent? It continues.
- When the CPU is running user-processes how does the OS run? Is the kernel one process? Where does it reside?
- Can a process induce its own context switch? Yes, we'll see how.

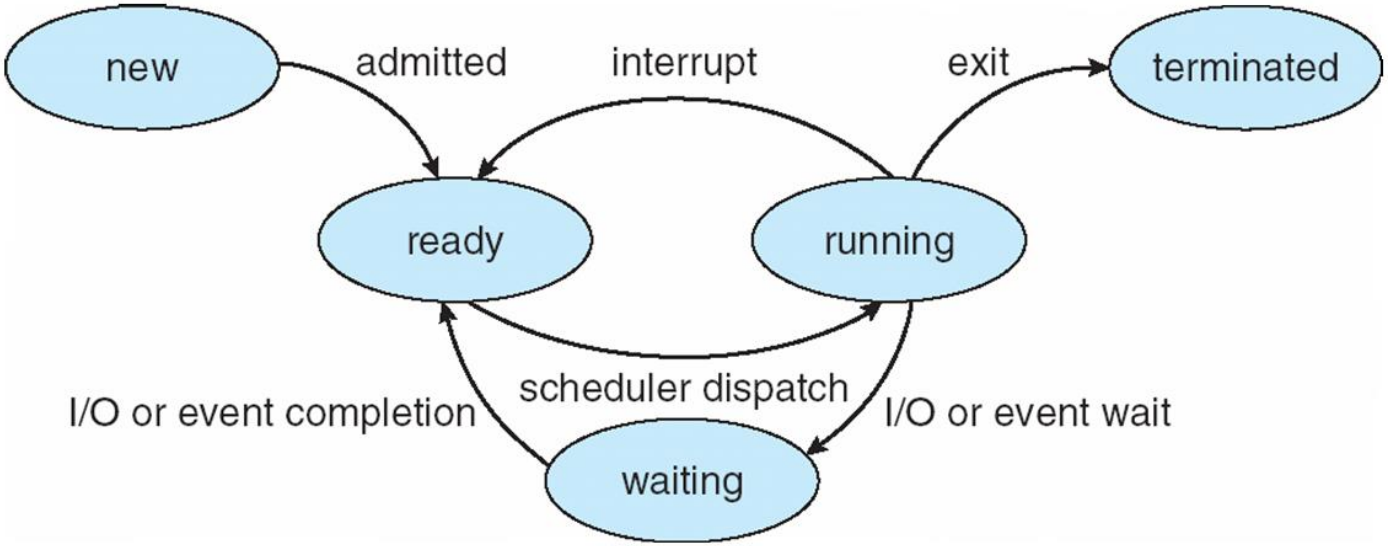
Notes

- Please register your iClicker: Canvas > iClicker menu item
  - Bring to class
- Introduce yourself on Canvas
- Canvas Discussion board available for discussions
  - usual rules apply
- The TA office hours and photos are on cs270 home page

More FAQ

- Q3.1 Registers are managed by the (a) Compiler
- Q2.b: Multiprogramming requires presence of multiple processors. False
- Q4.b POSIX API are used in a high level language. True

Diagram of Process State

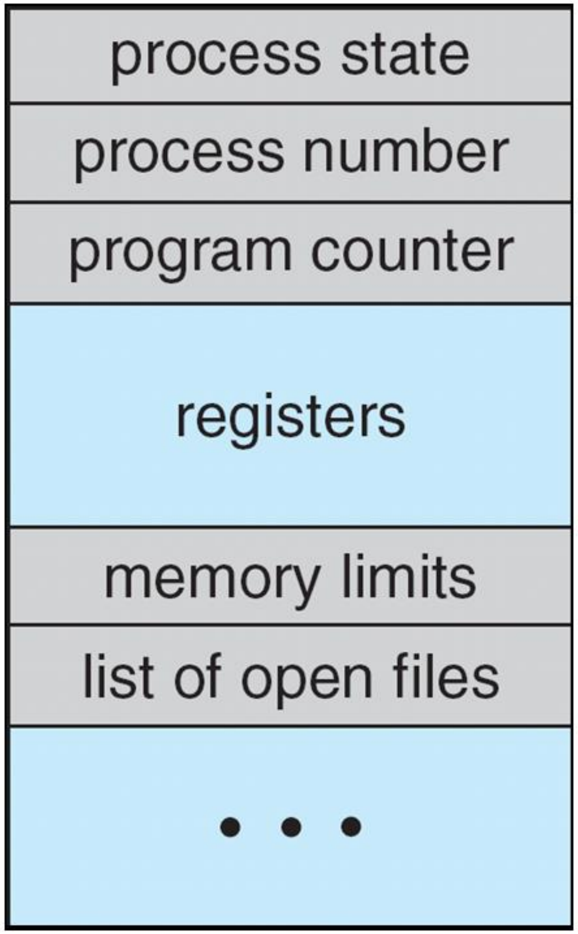


Ready to Running: scheduled by scheduler
Running to Ready: scheduler picks another process, back in ready queue
Running to Waiting (Blocked) : process blocks for input/output
Waiting to Ready: Input available

Process Control Block (PCB)

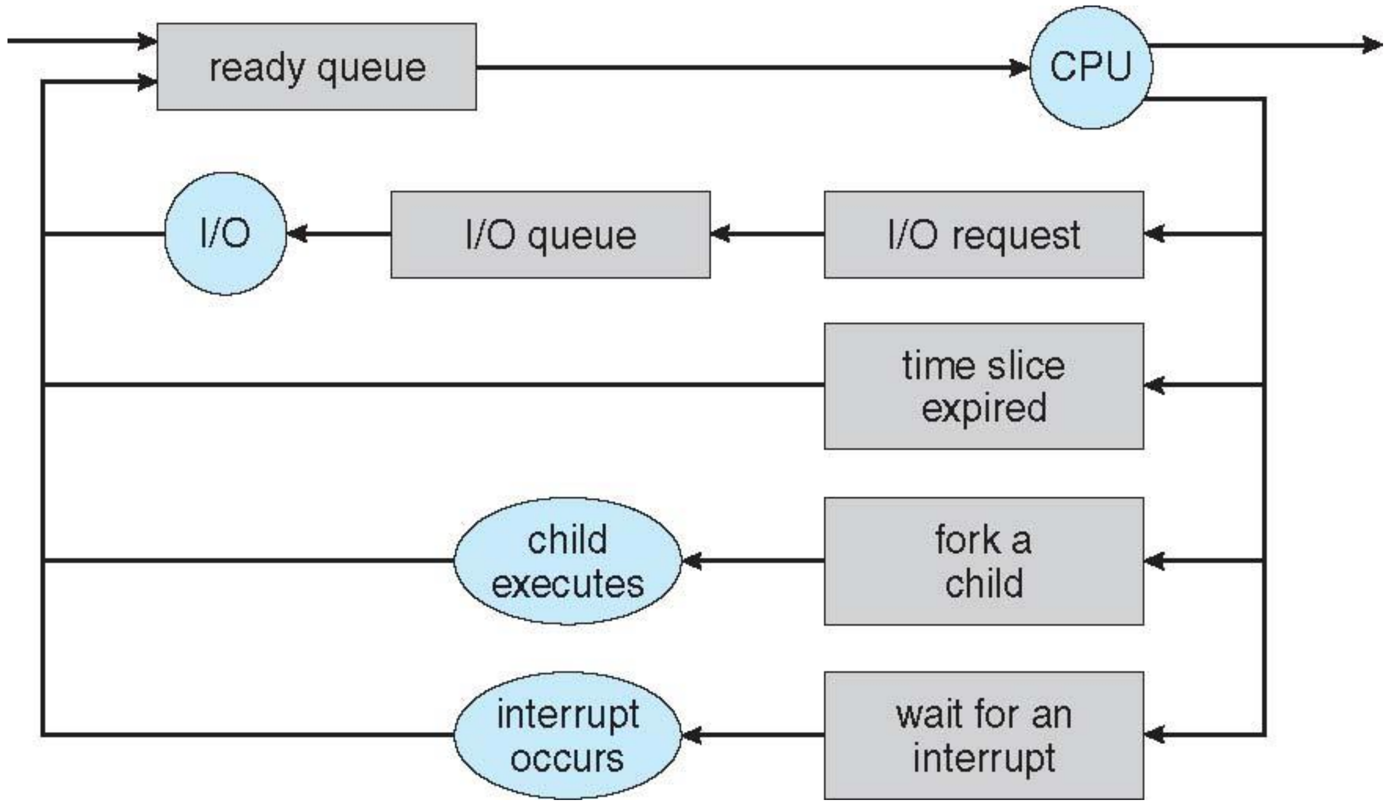
Information associated with each process (also called **task control block**)

- 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



Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows



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 (10-100 milliseconds) $\Rightarrow$ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue. Batch systems.
  - invoked infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
  - controls the **degree of multiprogramming**
- Processes can be described as either:
  - **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
- Long-term scheduler strives for good ***process mix***

Scheduling

- **UNIX and Windows systems often have no long-term scheduler**
- **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**
  - Handles strains on memory etc.

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
  - iPad with IOS9: Split View multitasking
- 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


ARE YOU SURE YOU WANT TO SIT AT THE KIDS TABLE?
Window Phone 7
the ENTERPRISE
Mobile OSs
www.jklossner.com
klossner

Context Switch

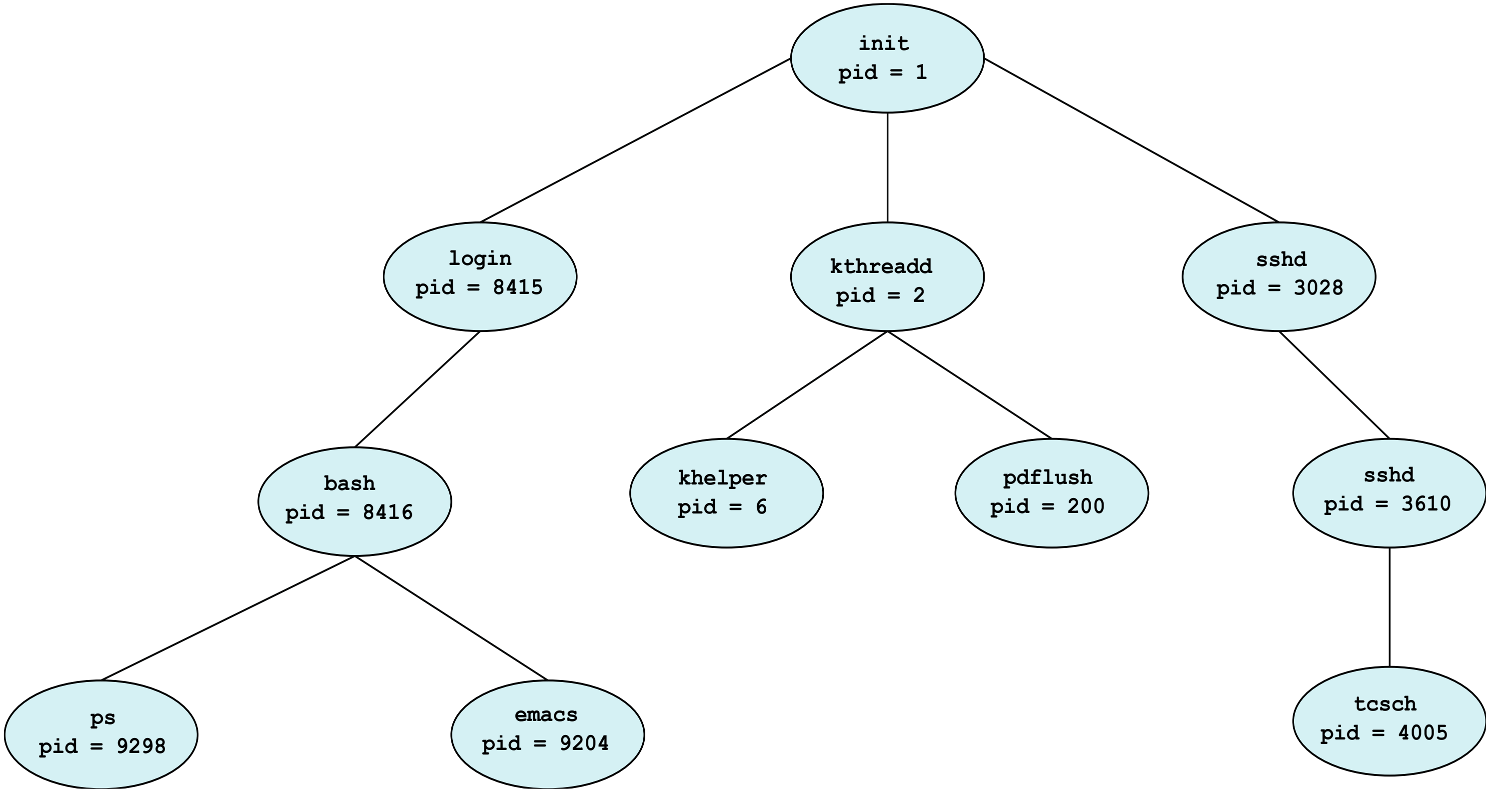
- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
  - CPU registers, process state, memory management info etc.
- Context-switch time (typically microsecs) 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

Processes creation & termination

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**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

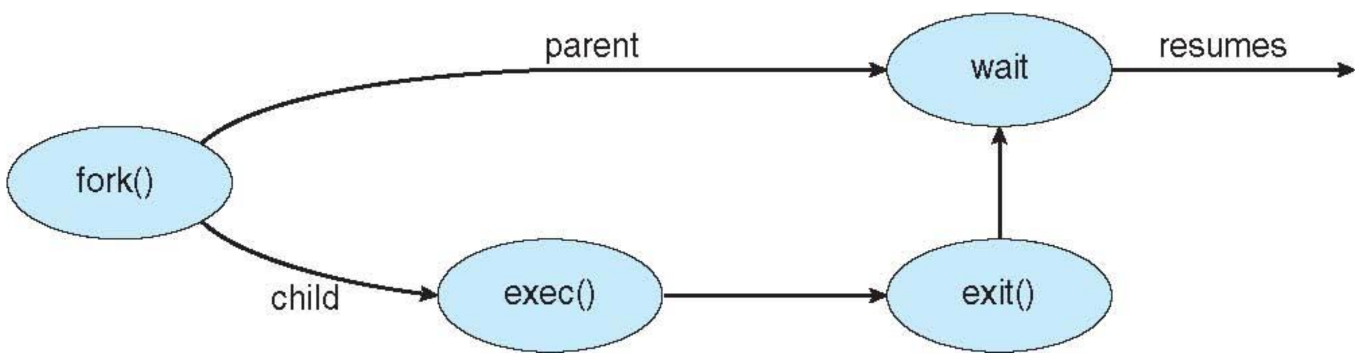
A Tree of Processes in Linux



Colorado State University

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - 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



Colorado State University

Fork ( ) to create a child process

- Fork creates a copy of process
- Return value from fork (): integer
  - When > 0:
    - Running in (original) Parent process
    - return value is pid of new child
  - When = 0:
    - Running in new Child process
  - When < 0:
    - Error! Perhaps exceeds resource constraints. sets errno (a global variable in errno.h)
    - Running in original process
- All of the state of original process duplicated in both Parent and Child!
  - Memory, File Descriptors (next topic), etc…

Process Management System Calls

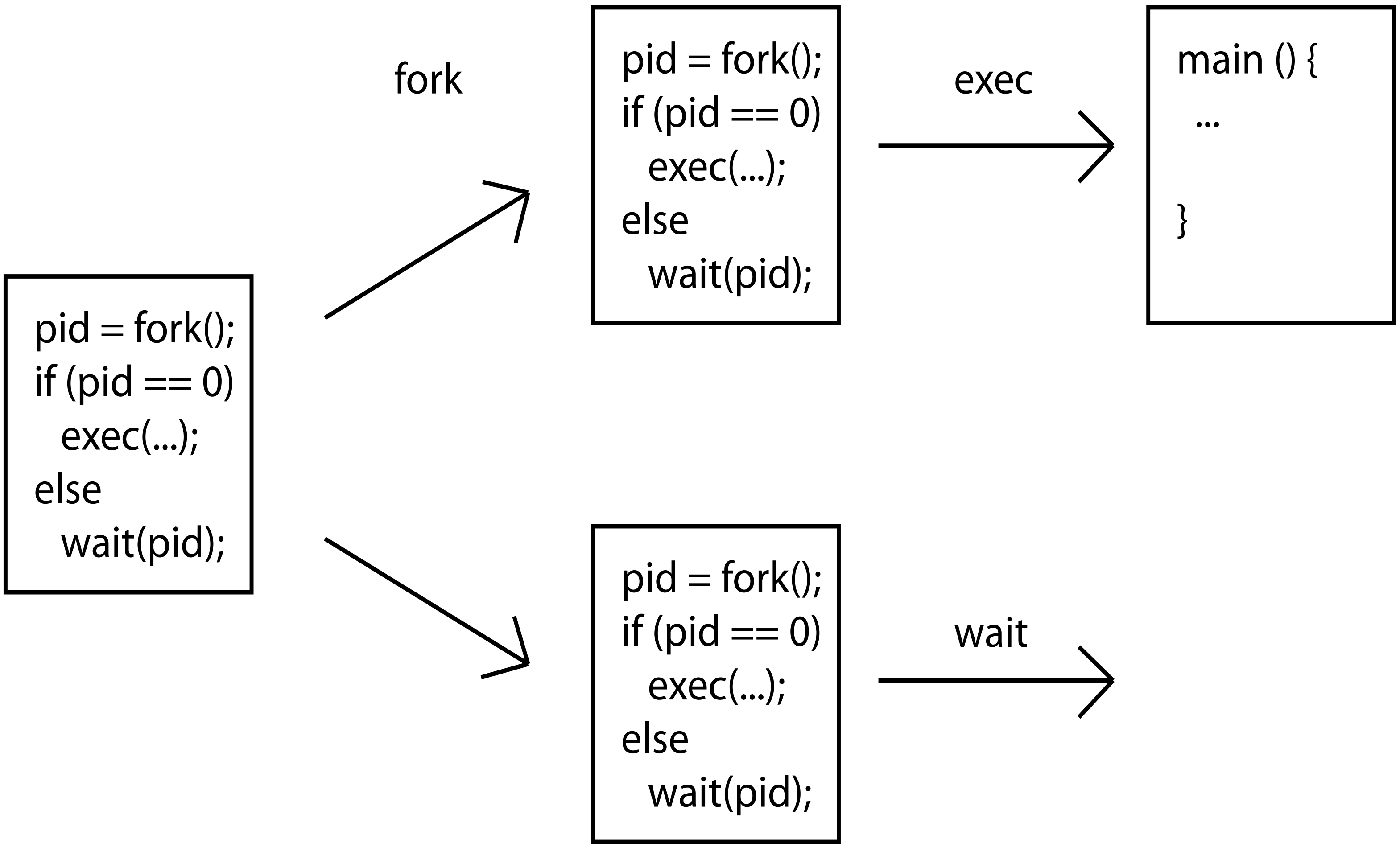
- UNIX fork – system call to create a copy of the current process, and start it running
  - No arguments!
- UNIX exec – system call to *change the program* being run by the current process. Several variations.
- UNIX wait – system call to wait for a process to finish
- Details: see man pages

Notes:

```
pid_t pid = getpid();   /* get current processes PID */;
waitpid(cid, 0, 0);   /* Wait for my child to terminate. */
exit (0);   /* Quit*/
kill(cid, SIGKILL);   /* Kill child*/
```


Colorado State University

UNIX Process Management



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;
}
```

execlp(3) - Linux man page
http://linux.die.net/man/3/execlp

Forking PIDs

```
#int main()
{
pid_t cid;

        /* fork a child process */
        cid = fork();
        if (cid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed\n");
           return 1;
        }
        else if (cid == 0) { /* child process */
            printf("I am the child %d, my PID is %d\n", cid, getpid());
             execlp("/bin/ls","ls",NULL);
        }
        else { /* parent process */
             /* parent will wait for the child to complete */
             printf("I am the parent with PID %d, my parent is %d, my child is %d\n",getpid(), getppid(), cid);
             wait(NULL);

             printf("Child Complete\n");
        }

  return 0;
}
```

```
Ys-MacBook-Air:ch3 ymalaiya$ ./newproc-posix_m
I am the parent with PID 494, my parent is 485, my child is 496
I am the child 0, my PID is 496
DateClient.java                         newproc-posix_m

Child Complete
Ys-MacBook-Air:ch3 ymalaiya$
```

wait/waitpid

- Wait/waitpid ( ) allows caller to suspend execution until child's status is available
- Process status availability
  - Generally after termination
  - Or if process is stopped
- pid_t waitpid(pid_t pid, int *status, int options);
- The value of pid can be:
  - 0   wait for any child process with same *process group ID* (perhaps inherited)
  - > 0   wait for child whose process group ID is equal to the value of pid
  - Others
- Status: where status info needs to be saved

Process Termination

- 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

- 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.
- The 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**

Multiprocess Architecture – Chrome Browser

- Early web browsers ran as single process
  - 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

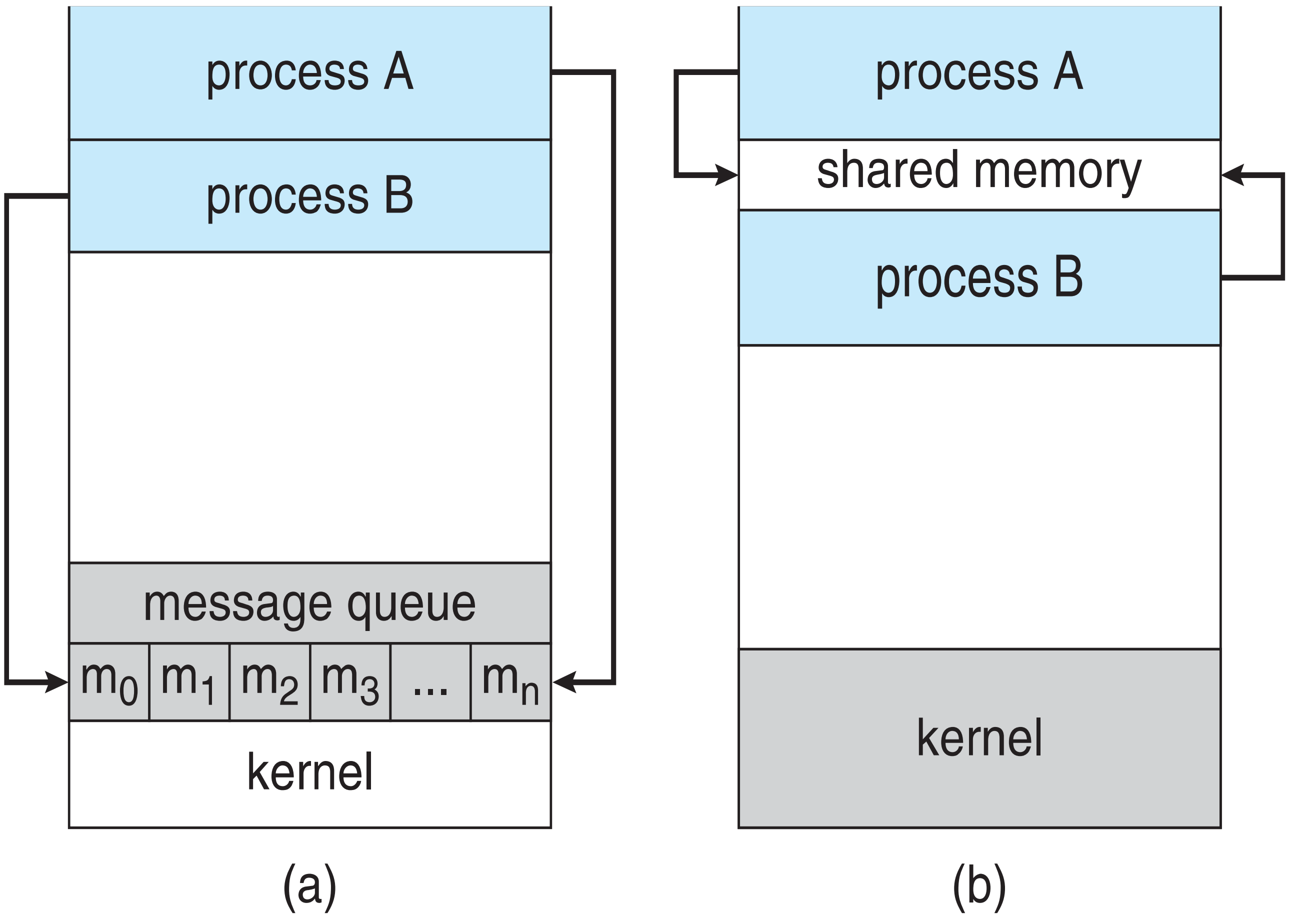
Each tab represents a separate process

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Message passing. (b) shared memory.



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
  - Modularity
  - Convenience