CS370 Operating Systems

Colorado State University
Yashwant K Malaiya

Fall 24 Lecture 6
OS Structures/Processes

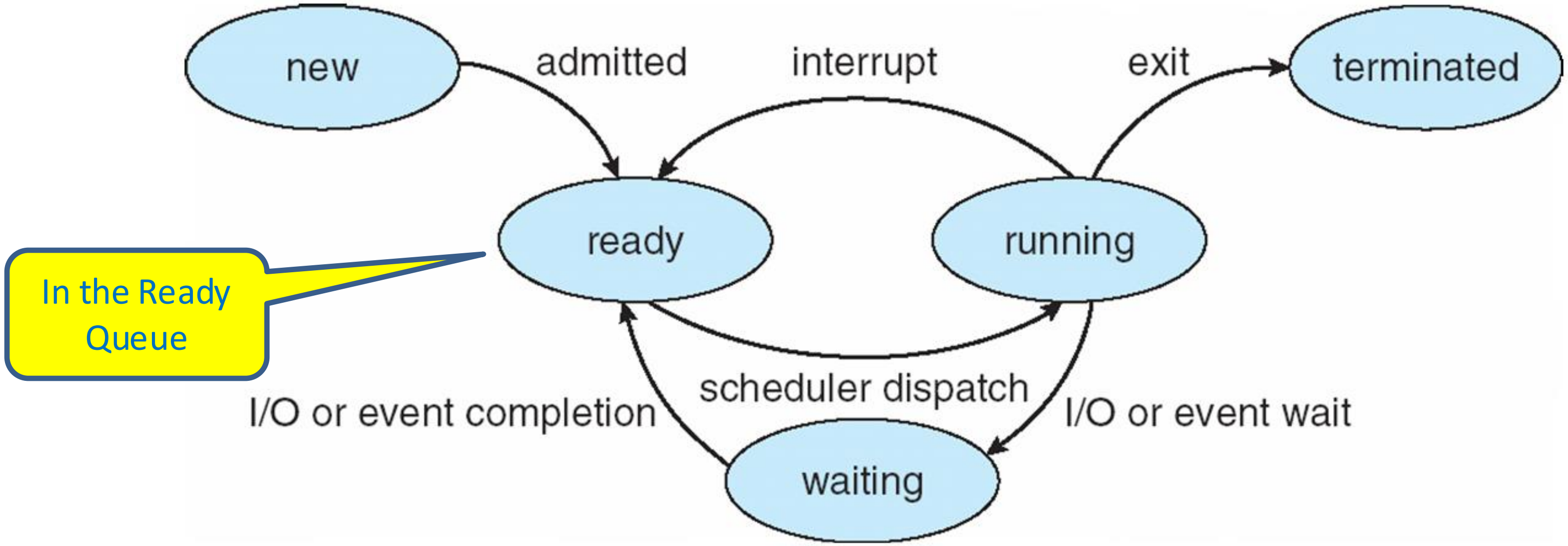
Slides based on

- 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

Diagram of Process State

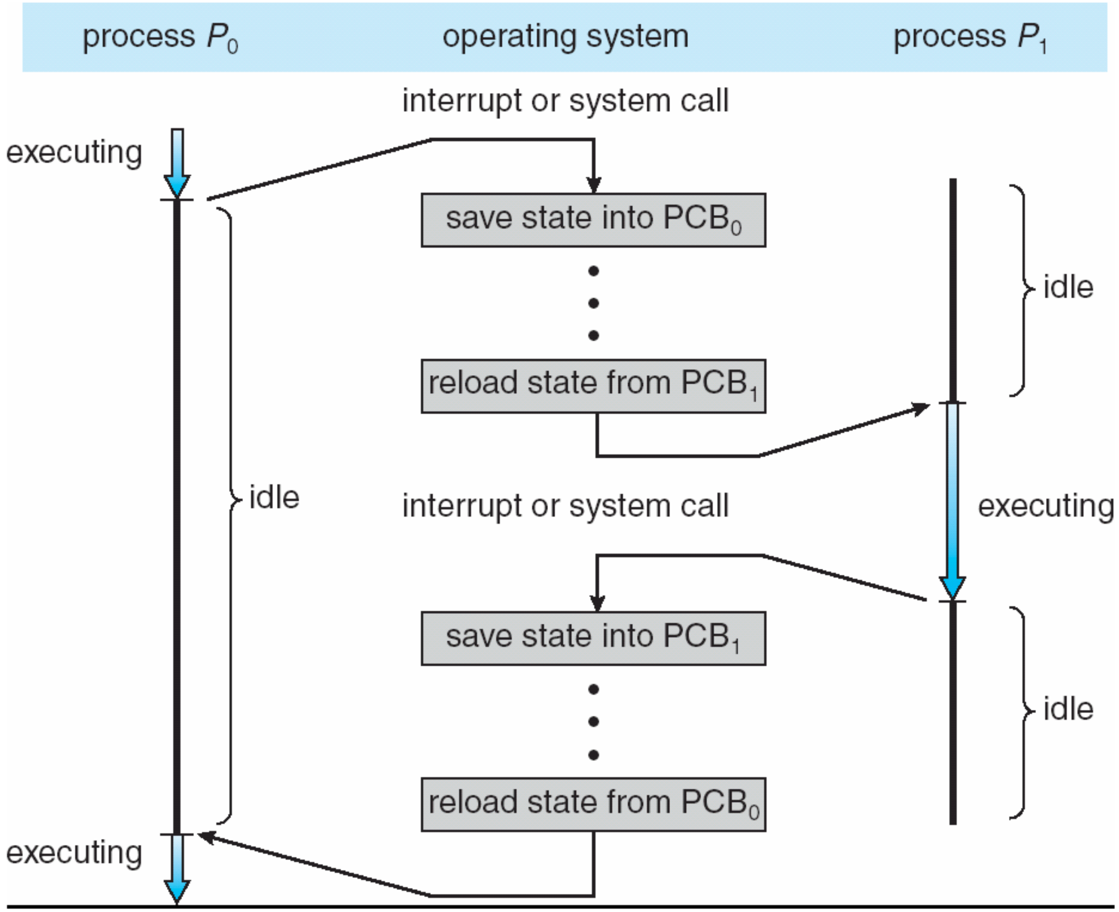


Transitions:
Ready to Running: scheduled by scheduler
Running to Ready: scheduler picks another process, back in ready queue

Running to Waiting (Blocked) : process blocks for input/output
Waiting to Ready: I/O or event done

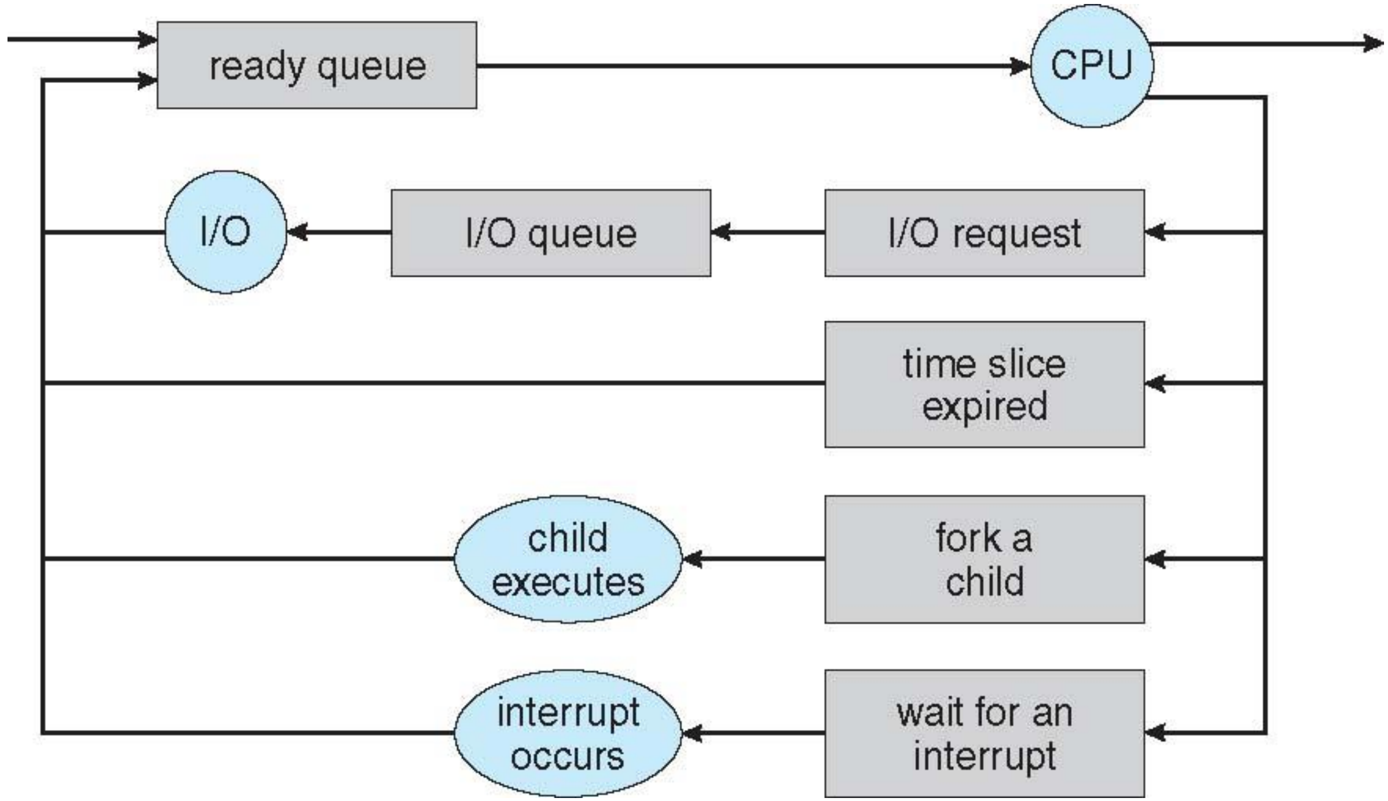
Colorado State University

CPU Switch From Process to Process



Representation of Process Scheduling

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



Assumes a single CPU. Common until recently

Colorado State University

Context Switch

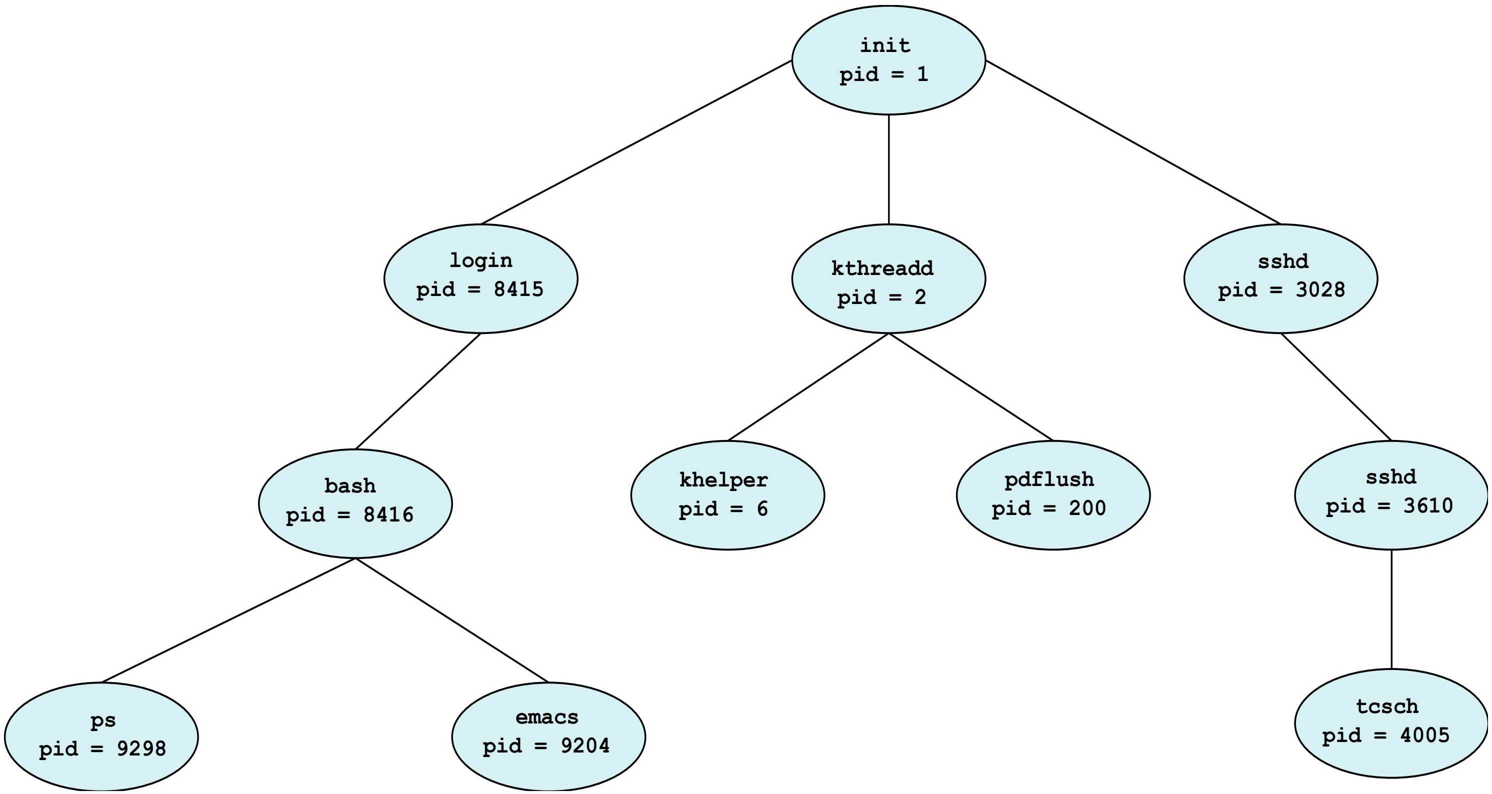
- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - 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

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 or just a few*?
- Execution options
  - Parent and children execute concurrently?
  - Parent waits until children terminate*?

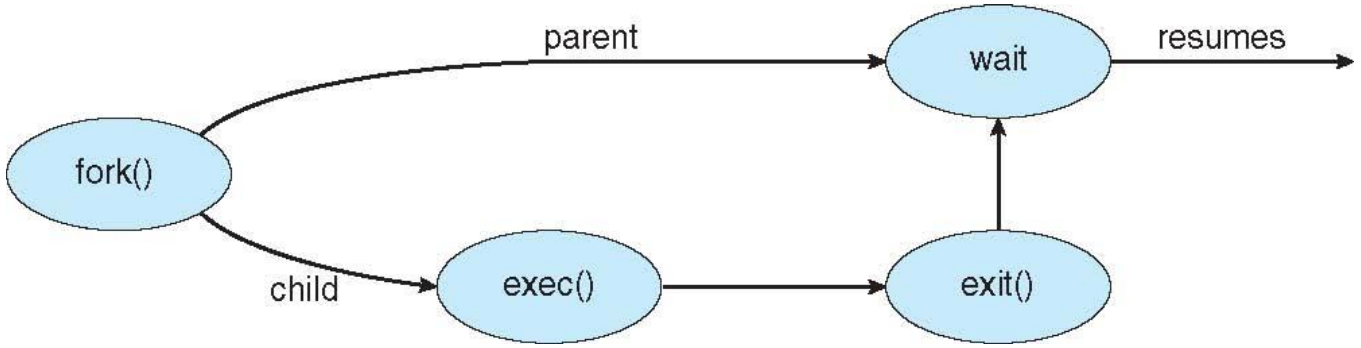
A Tree of Processes in Linux



Colorado State University

Process Creation (Cont.)

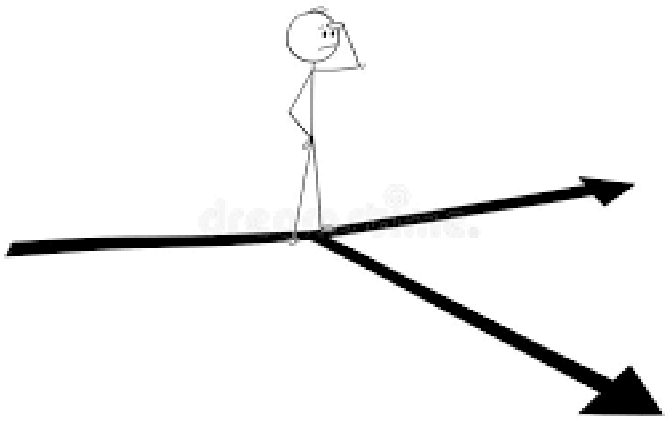
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program



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! Almost ..
  - Memory, File Descriptors (next topic), etc…



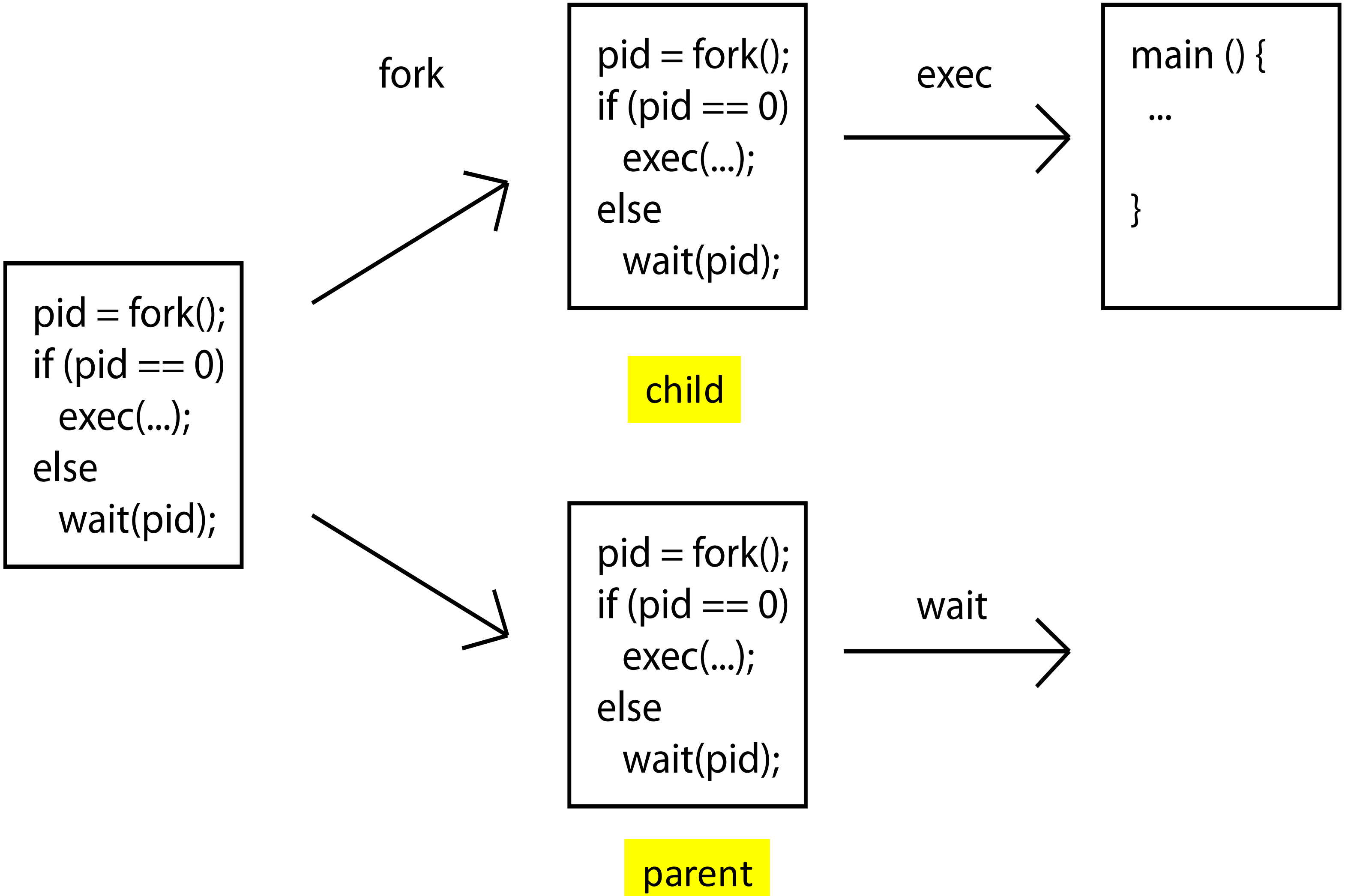
Process Management System Calls

- 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]

Some examples:

- 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*/

UNIX Process Management



C Program Forking Separate Process

```c
#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;
}
```

<sys/types.h> definitions of derived types
<unistd.h> POSIX API

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

Forking PIDs

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

See self-exercise in Teams

https://www.tutorialspoint.com/compile_c_online.php

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
  - -1 wait for any child process *(equi to wait ( ))*
- Status: where status info needs to be saved

Linux: fork ( )

- Search for man fork( )
- http://man7.org/linux/man-pages/man2/fork.2.html

NAME fork - create a child process

SYNOPSIS #include <unistd.h>

pid_t fork(void);

DESCRIPTION fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. …

The child process and the parent process run in separate memory spaces…

The child process is an exact duplicate of the parent process except for the following points: ….

RETURN VALUE On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

EXAMPLE See pipe(2) and wait(2).

…

errno is a global variable in errno.h

Process Group ID

- Process group is a collection of related processes
- Each process has a process group ID
- Process group leader?
  - Process with pid equal to pgid
- A process group has an associated controlling terminal, usually the user's keyboard
  - Control-C: sends interrupt signal (SIGINT) to all processes in the process group
  - Control-Z: sends the suspend signal (SIGSTOP) to all processes in the process group

Applies to foreground processes: those interacting
With the terminal

Process Groups

A child Inherits parent's process group ID. Parent or child can change group ID of child by using setpgid.

By default, a Process Group comprises:

- Parent (and further ancestors)
- Siblings
- Children (and further descendants)

A process can only send *signals* to members of its process group

- Signals are a limited form of inter-process communication used in Unix.
- Signals can be sent using system call
  - int kill(pid_t pid, int sig);

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 **`kill( )`** 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

```
kill(child_pid,SIGKILL);
```

Colorado State University

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 (it is still running, reclaimed by init)

Zombie: a process that has completed execution (via the exit system call) but still has an entry in the process table

Multi-process Program Ex – 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

Multitasking

Colorado State University

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

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**

Colorado State University

Producer-Consumer Problem

- Common paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Why do we need a buffer (shared memory region)?
- The producer and the consumer process operate at their own speeds. Items wait in the buffer when consumer is slow.

Where does the bounded buffer "start"?
- It is circular

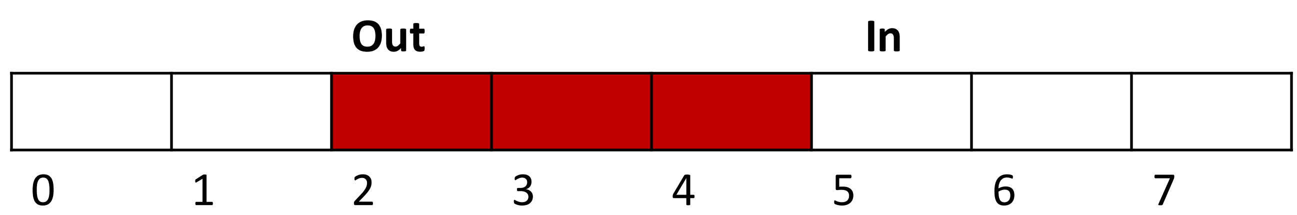
Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 8
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

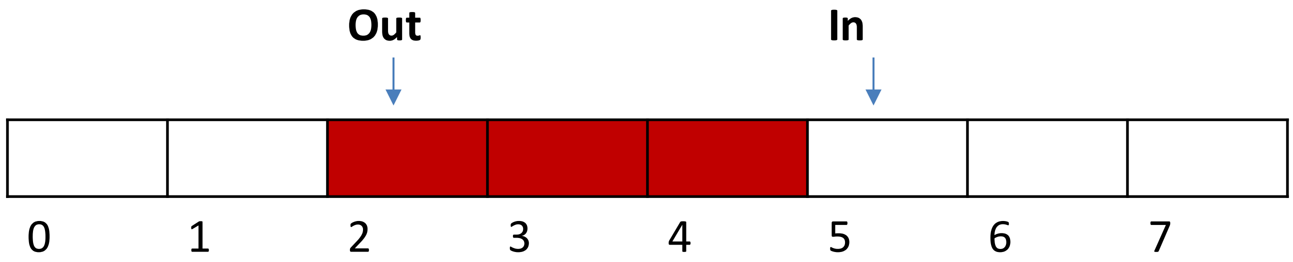
- **in** points to the **next free position** in the buffer
- **out** points to the **first full position** in the buffer.
- Buffer is empty when **in == out**;
- Buffer is full when
  ((in + 1) % BUFFER SIZE) == out. (Circular buffer)
- This scheme can only use BUFFER_SIZE-1 elements



(2+1)%8 =3 but (7+1)%8 =0

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

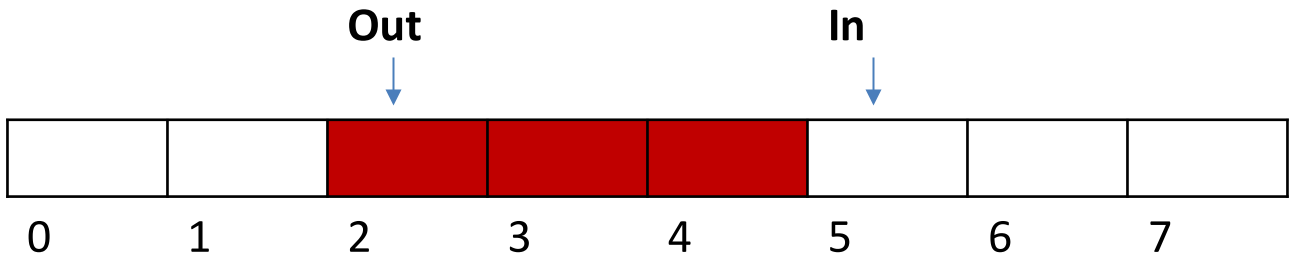


Colorado State University

Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```



Interprocess Communication – Shared Memory

- Each process has its own private address space.
- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the user processes, not the operating system.
- Major issue is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
  - Synchronization is discussed in great details in a later Chapter.
- Example soon.

Only one process may access shared memory at a time

Colorado State University

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical: Options (details next)
    - Direct (process to process) or indirect (mail box)
    - Synchronous (blocking) or asynchronous (non-blocking)
    - Automatic or explicit buffering

Direct Communication

- Processes must name each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

  `send`(*A, message*) – send a message to mailbox A

  `receive`(*A, message*) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Possible Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization*(blocking or not)*

- Message passing may be either blocking or non-blocking
- **Blocking** is termed **synchronous**
  - **Blocking send** -- sender is blocked until message is received
  - **Blocking receive** -- receiver is blocked until a message is available
- **Non-blocking** is termed **asynchronous**
  - **Non-blocking send** -- sender sends message and continues
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous.**
  - Producer-Consumer problem: Easy if both block

Examples of IPC Systems

OSs support many different forms of IPC*. We will look at two of them

- Shared Memory
- Pipes

* **Linux kernel supports**: Signals, **Anonymous Pipes**, Named Pipes or FIFOs, SysV Message Queues, POSIX Message Queues, SysV Shared memory, **POSIX Shared memory**, SysV semaphores, POSIX semaphores, FUTEX locks, File-backed and anonymous shared memory using mmap, UNIX Domain Sockets, Netlink Sockets, Network Sockets, Inotify mechanisms, FUSE subsystem, D-Bus subsystem