CS370 Operating Systems

Colorado State University
Yashwant K Malaiya
Fall 2016 Lecture 8

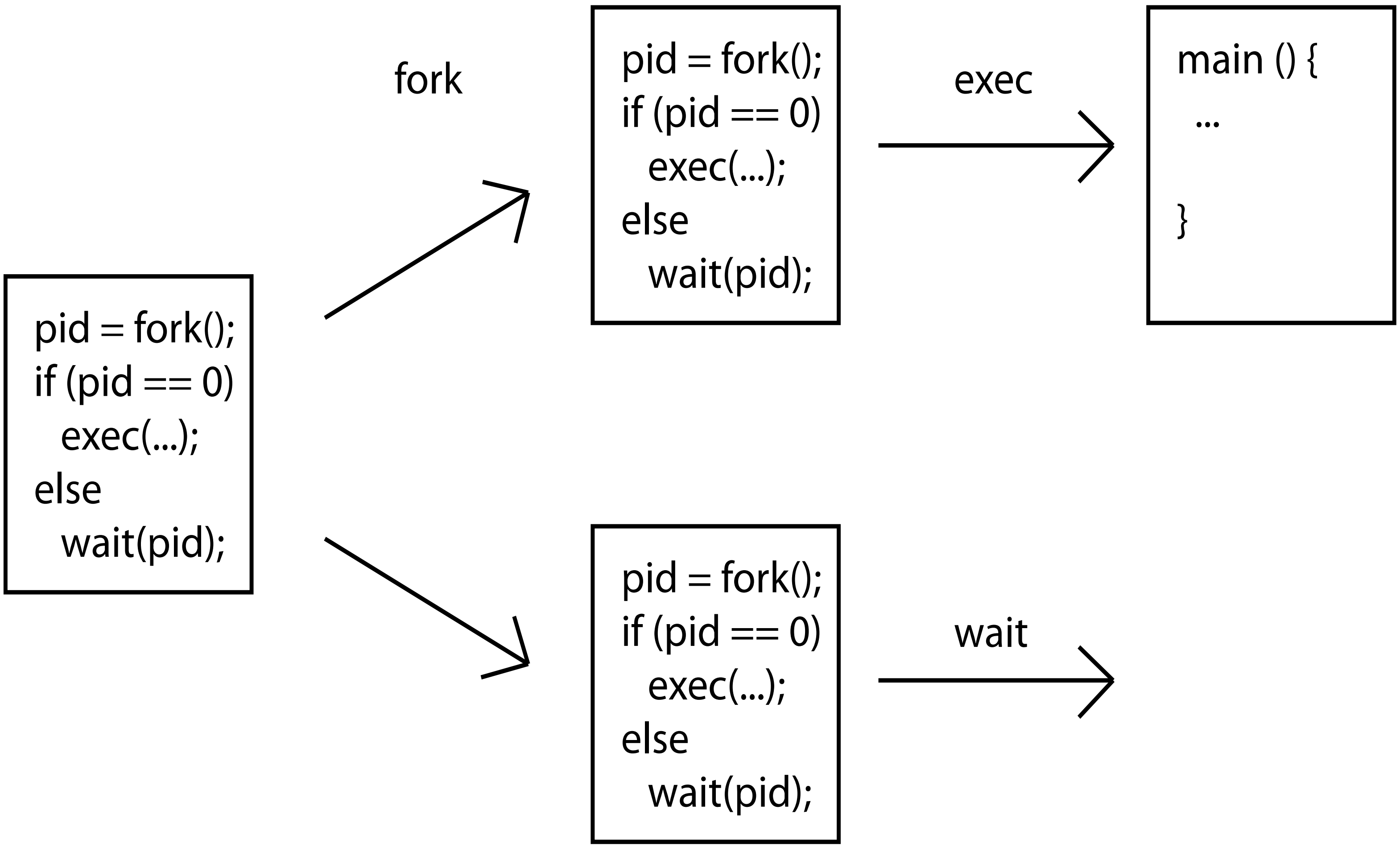
Slides based on

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

FAQ

- What is process control block PCB? Where is it? Is it stack? Typical size?
  - Data structure stored in kernel's memory space, perhaps as structs in a linked list.
- Context switch time?: depends on PCB size, cache & TLB
- Scheduler: hw or sw? part of kernel
- How the scheduler chooses? Details coming up soon
- Parent & child processes: difference between them? Is it necessary to have a tree structure?
- Is it exec ( ) that makes child run a different process?
- Does the parent process always wait( ) for a child to finish?
- Difference?: wait(int *wstatus)   ex: wait(NULL)
  - waitpid(pid_t pid, int *wstatus, int options); [see man pages]

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;
}
```

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$
```


Colorado State University

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
  - < -1 any child process whose process group ID equal to the absolute is value of pid
- wait(&wstatus) is equivalent to waitpid(-1, &wstatus, 0);
- Status: where status info needs to be saved
- WEXITSTATUS(wstatus) returns exit status of child …

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

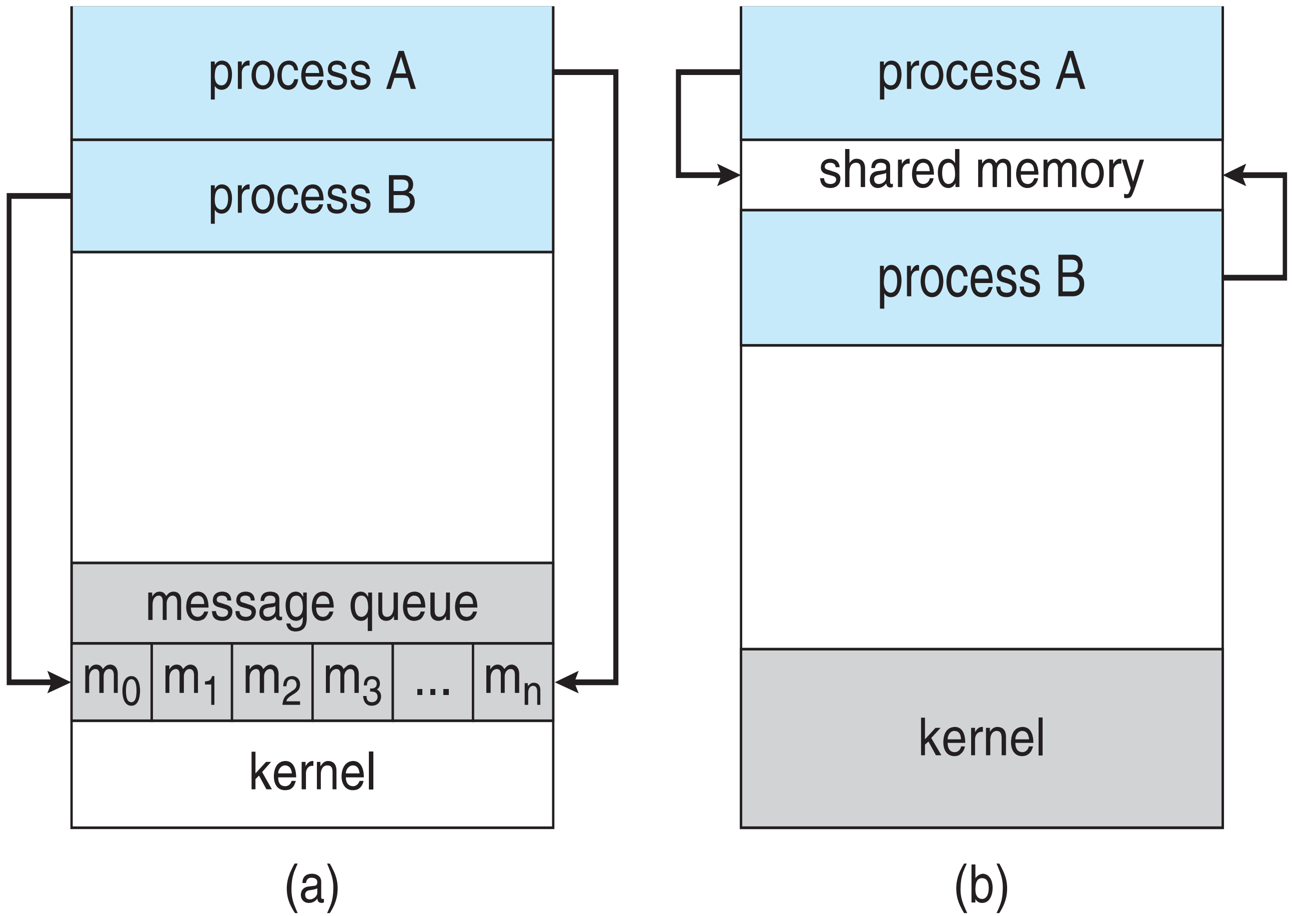
Colorado State University

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.



Colorado State University

Producer-Consumer Problem

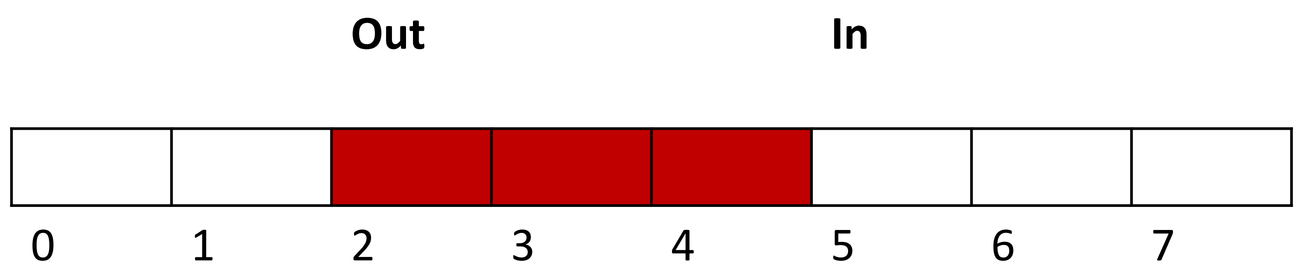
- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

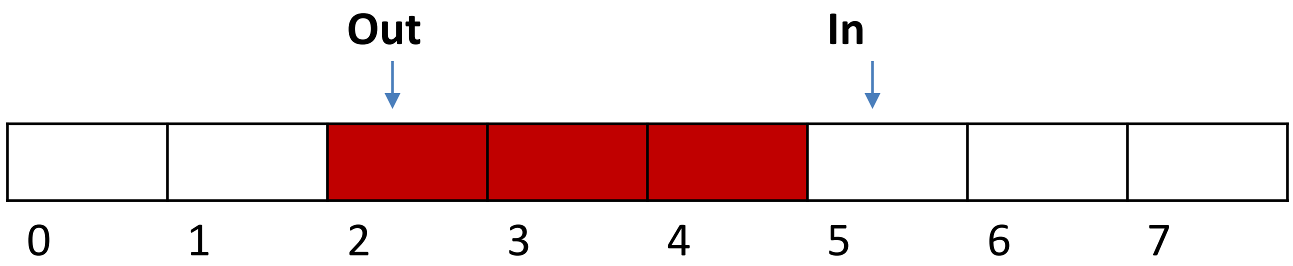
```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

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



Bounded-Buffer – Producer

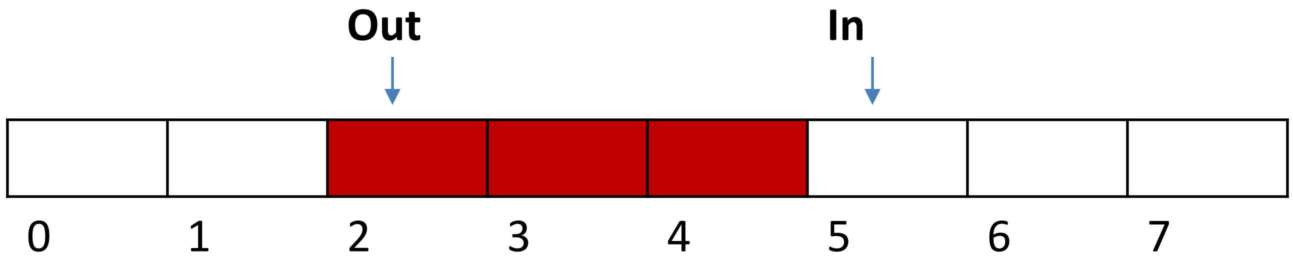
```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```



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 */
}
```



Colorado State University

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues 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 Chapter 5.

- 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

Colorado State University

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.