

CSC139 Operating System Principles

Fall 2019, Part 2-1

Instructor: Dr. Yuan Cheng

Session Plan

- Process Concept
- Process States and Scheduling
- Operations on Processes
- Inter-process Communication

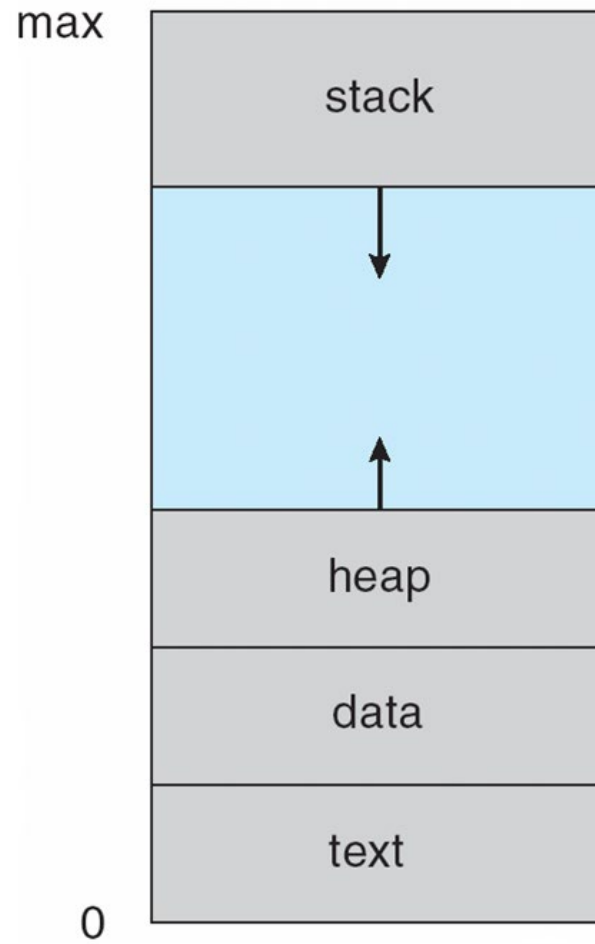
Chapter Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast inter-process communication using shared memory and message passing.
- Design programs that uses pipes and POSIX shared memory to perform inter-process communication.
- Describe client-server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.

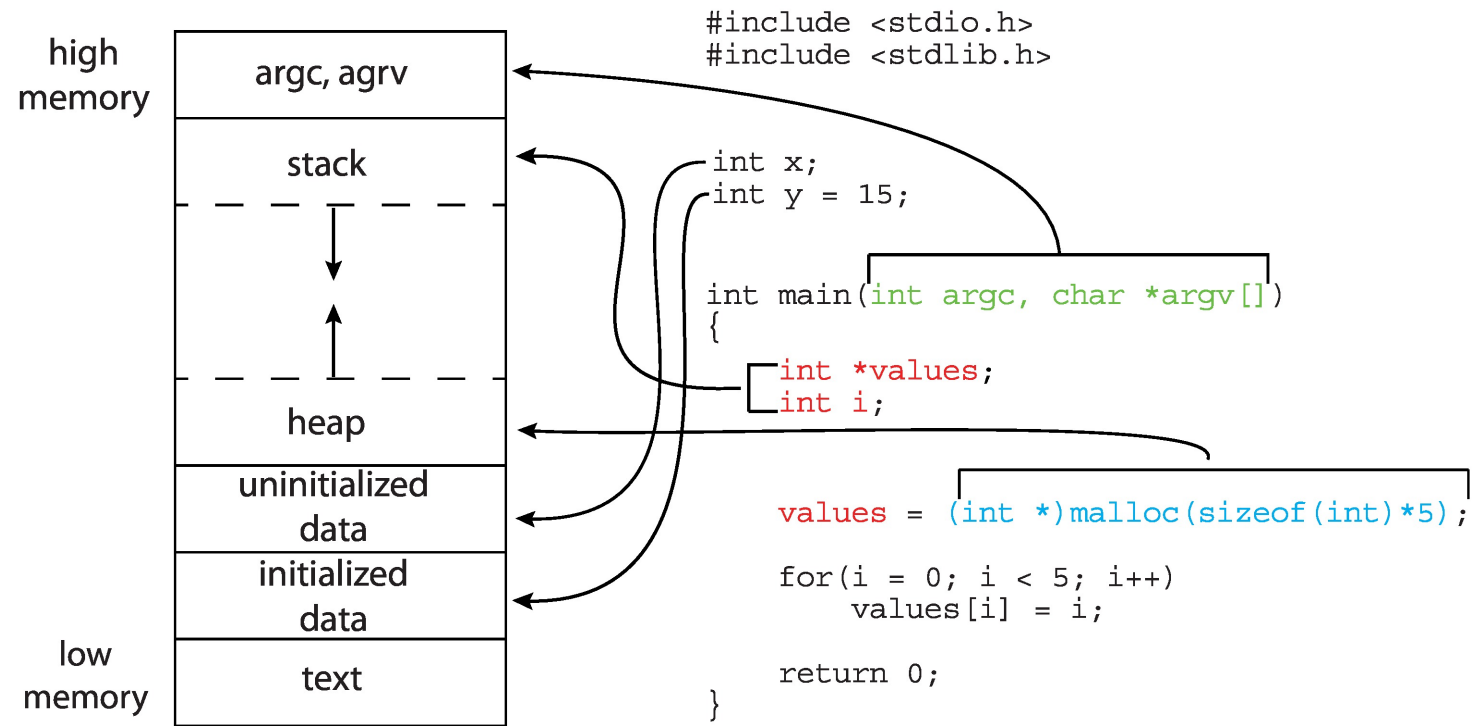
Process Concept

- **Process** – a program in execution
 - process execution must progress in sequential fashion
- A *program* is a *passive* entity, whereas a process is an *active* entity with a program counter and a set of associated resources
- Each process has its own address space
 - The program code, also called **text section**
 - **Stack** containing temporary data
 - function parameters, return addresses, local variables
 - **Data section** containing global variables
 - **Heap** containing memory dynamically allocated during run time
- **Program counter** and CPU registers are part of the **process context**

Process in Memory



Memory Layout of a C Program



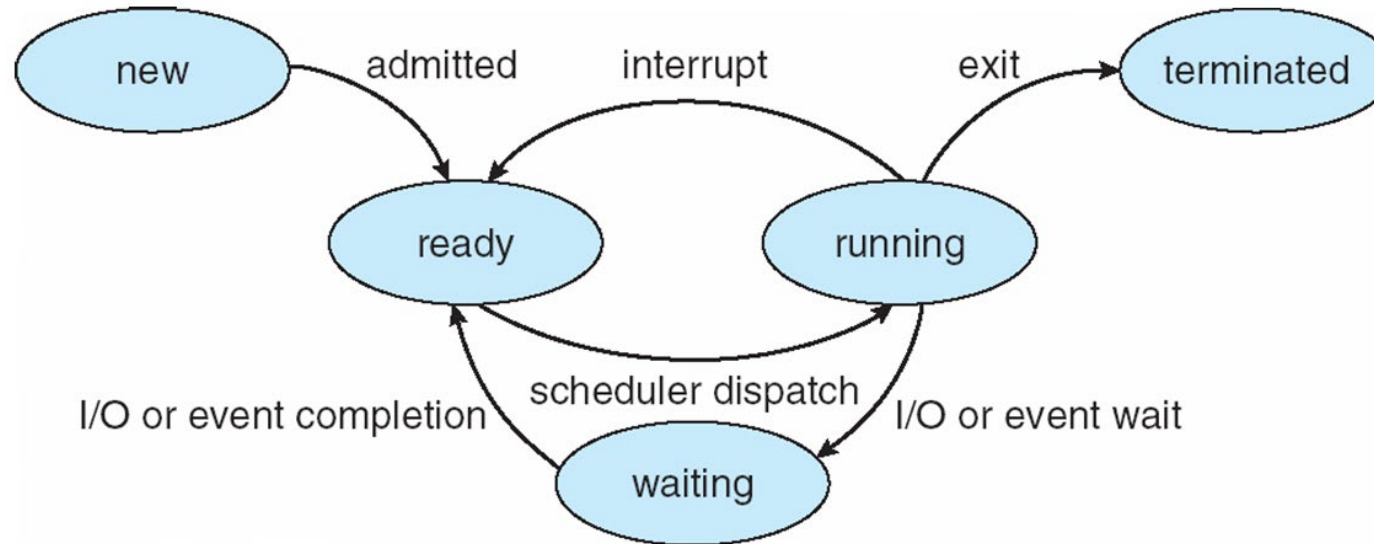
Peeking Inside

- Processes share code, but each has its own “context”
- CPU
 - Instruction pointer (Program Counter)
 - Stack pointer
- Memory
 - Set of memory addresses (“address space”)
 - `cat /proc/<PID>/maps`
- Disk
 - Set of file descriptors
 - `cat /proc/<PID>/fdinfo/*`

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

Diagram of Process State



- As the program executes, it moves from state to state, as a result of the program actions (e.g., system calls), OS actions (scheduling), and external actions (interrupts).

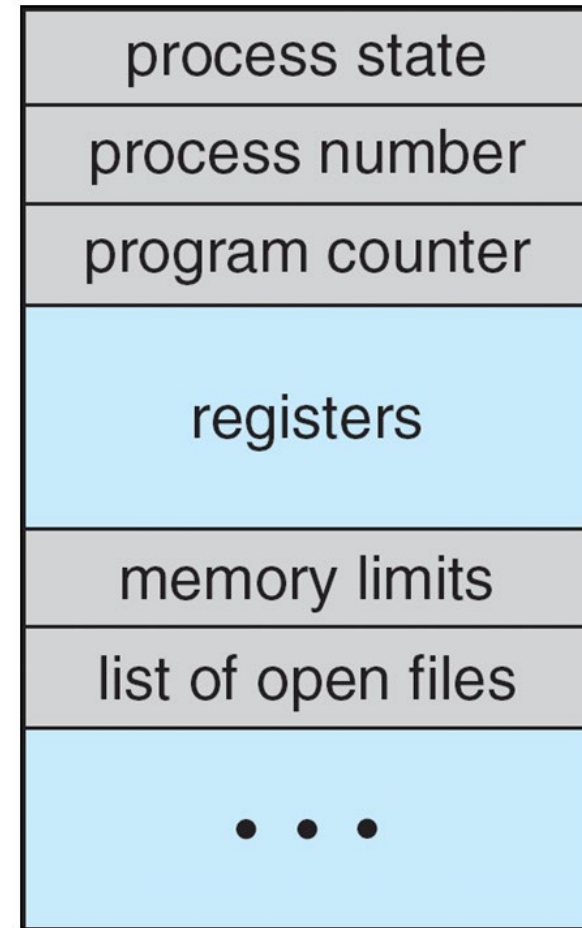
Process Control Block

PCB: OS data structures to keep track of all processes

- The PCB tracks the execution state and location of each process
- The OS allocates a new PCB on the creation of each process and places it on a state queue
- The OS deallocates the PCB when the process terminates

The PCB contains:

- 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



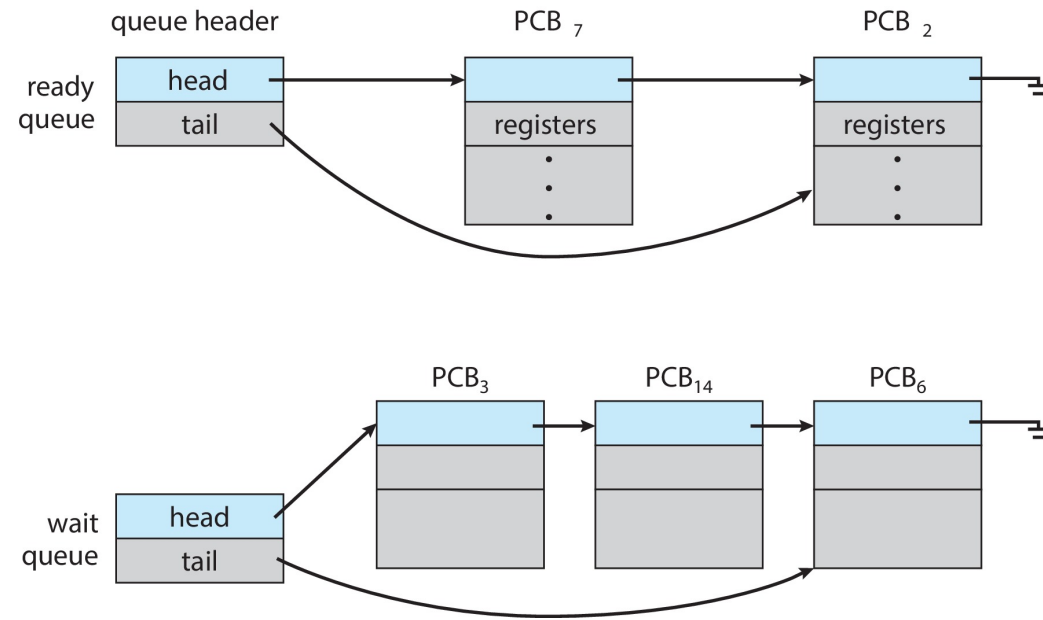
Process Scheduling

- The operating system is responsible for managing the scheduling activities
 - A uniprocessor system can have only one running process at a time
 - The main memory cannot always accommodate all processes at runtime for multi-programmed OS
 - The operating system will need to decide on which process to execute next ([CPU scheduling](#)), and which processes will be brought to the main memory ([job scheduling](#))

Process Scheduling

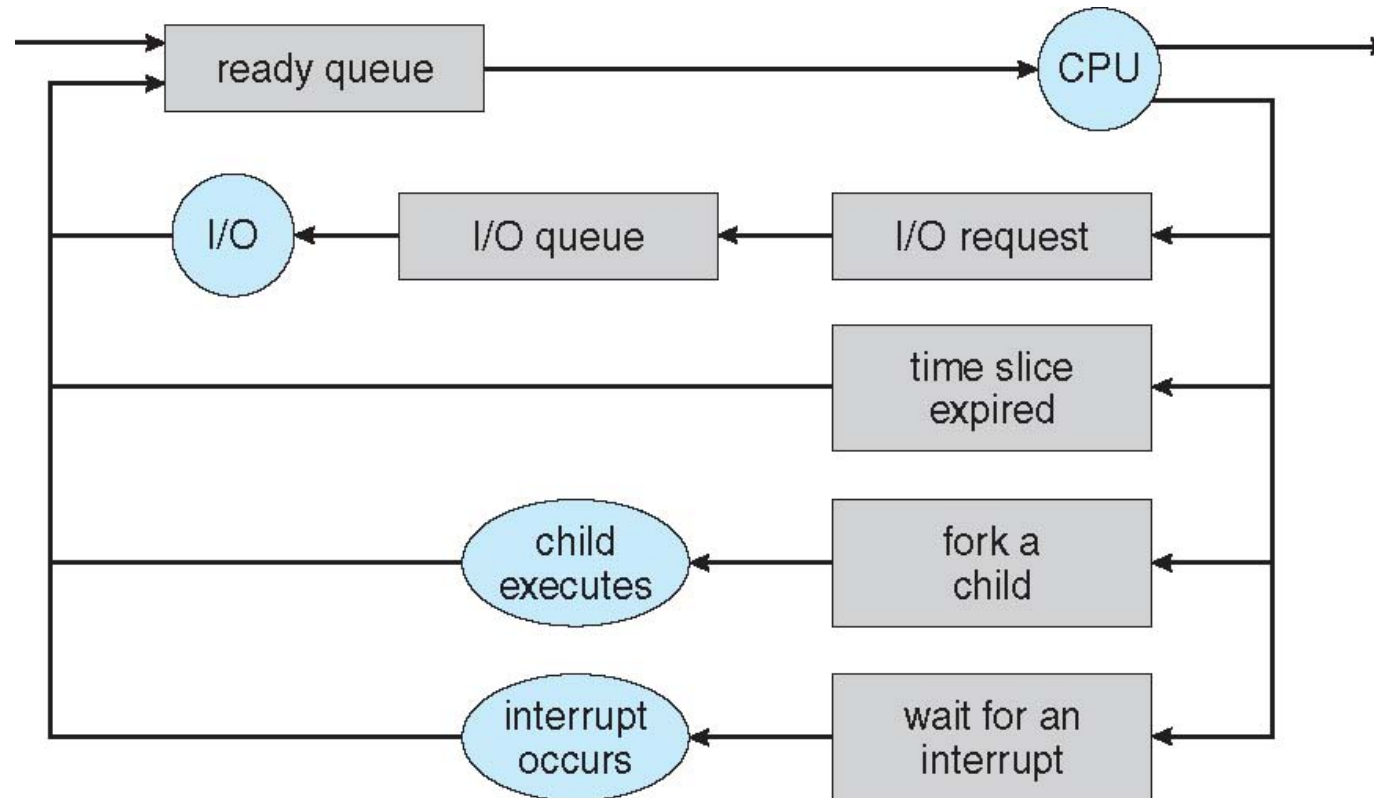
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
 - **Job queue** – set of all processes in the system
 - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
 - **Wait queues** – set of processes waiting for an event (e.g., I/O)
 - Processes migrate among the various queues

Ready and Wait Queues



Representation of Process Scheduling

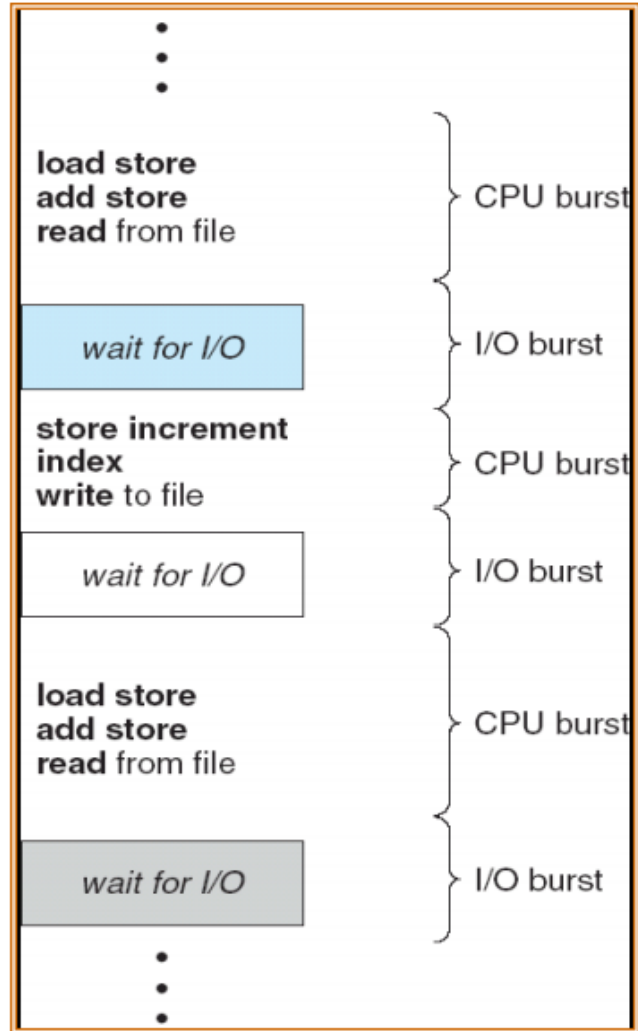
- Queueing diagram represents queues, resources, flows



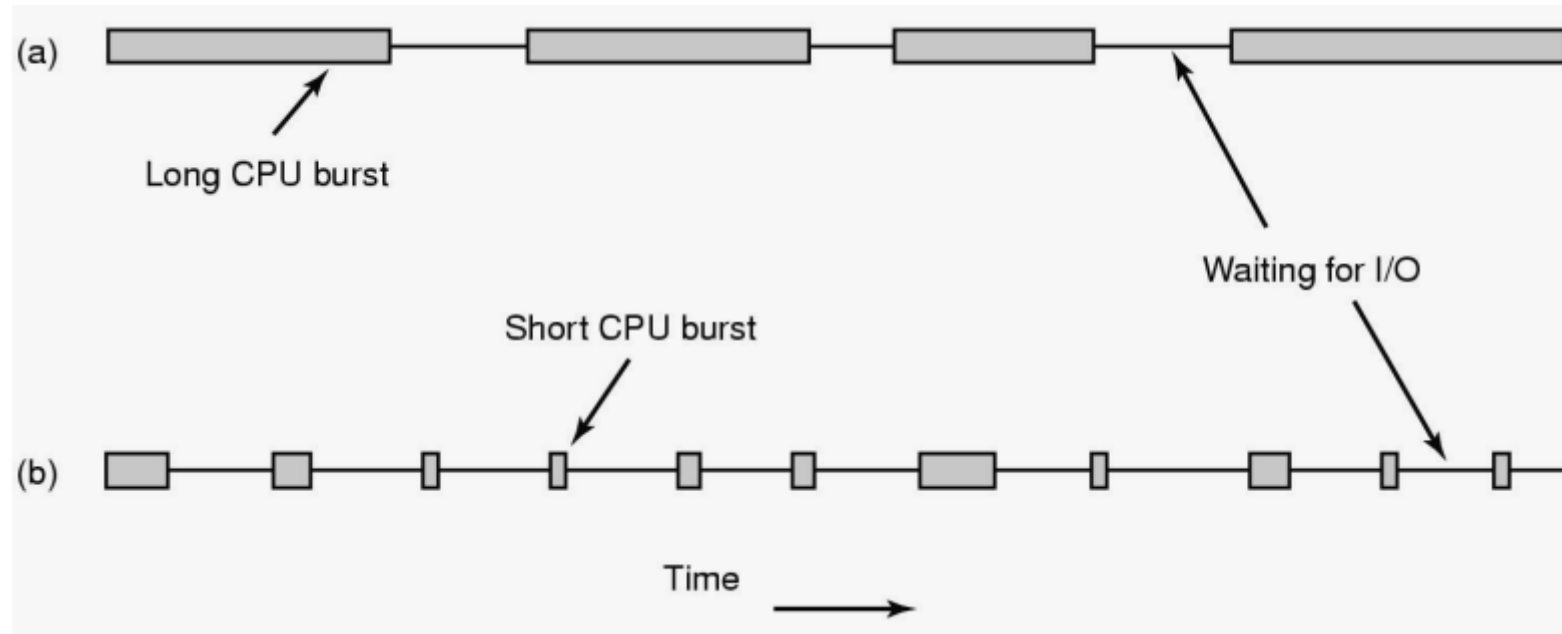
Schedulers

- **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 (milliseconds) \Rightarrow (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) \Rightarrow (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, 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***

CPU and I/O Bursts



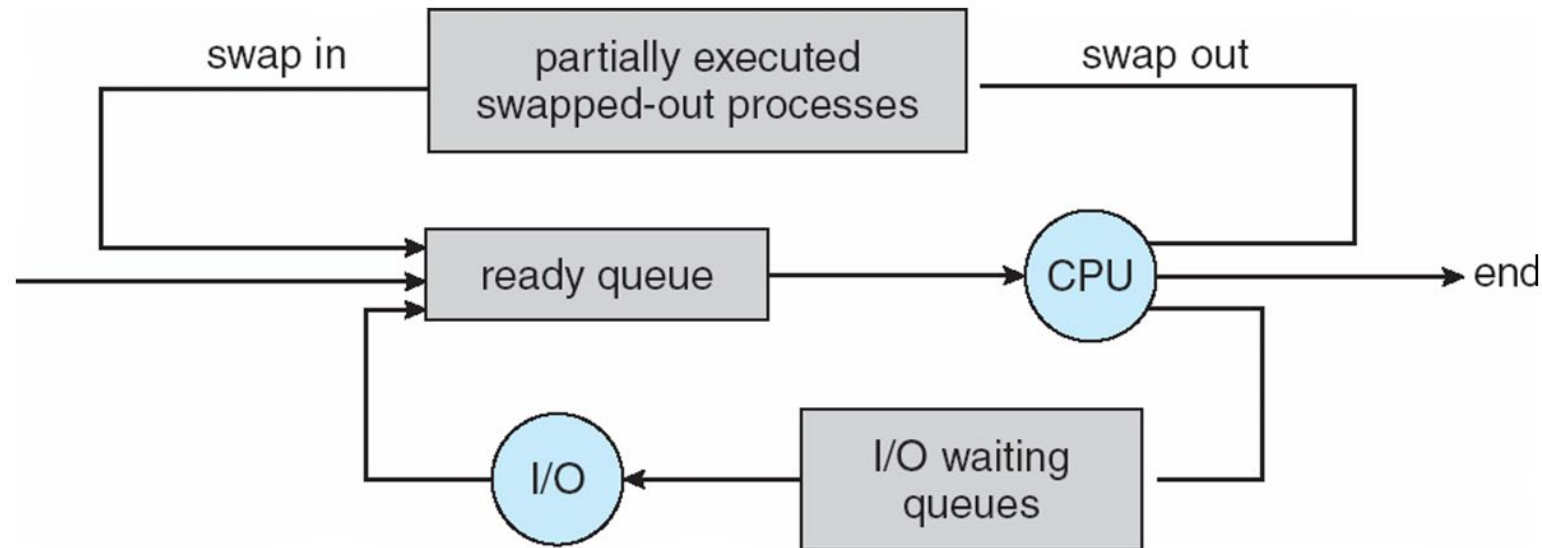
CPU-bound vs. I/O-bound Processes



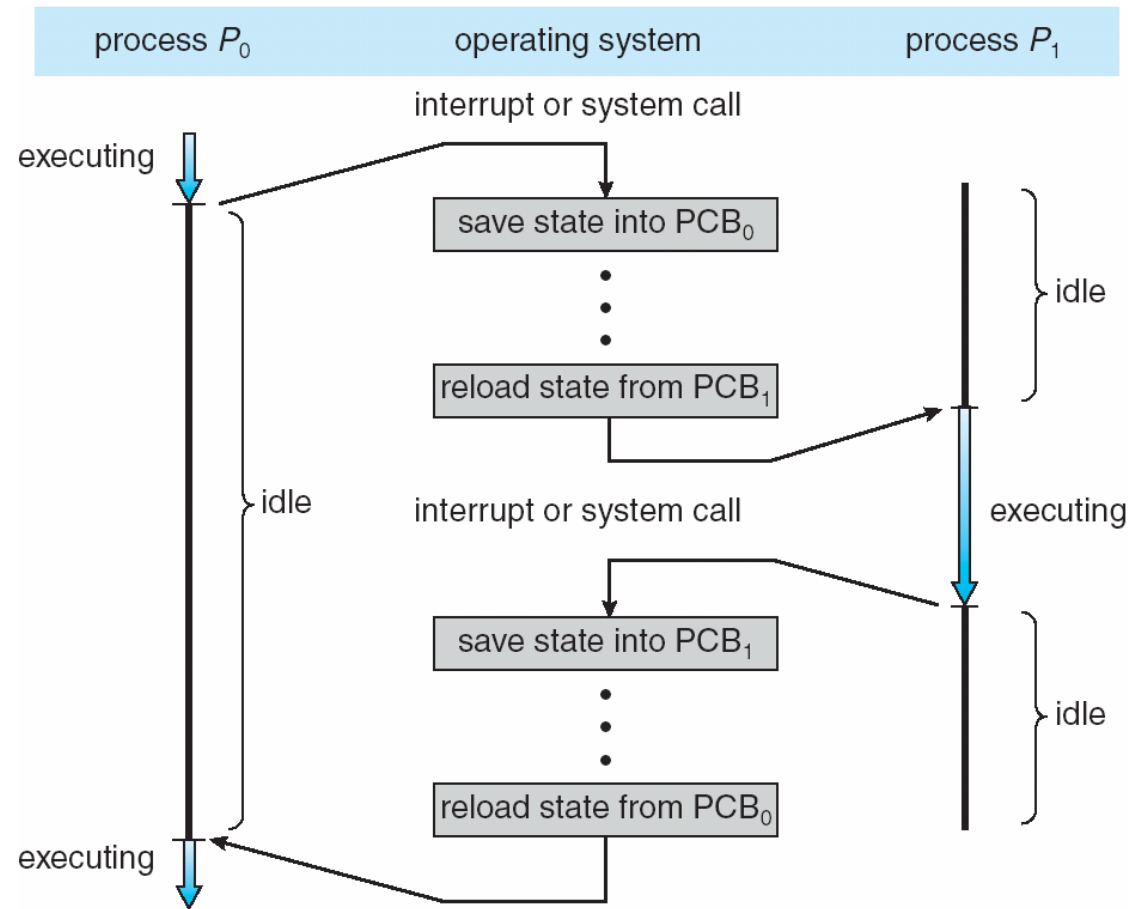
- (a) A CPU-bound process
- (b) An I/O-bound process

Addition of 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**



CPU Switch from Process to Process



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

Operations on Processes

- System must provide mechanisms for:
 - process creation,
 - process termination,
 - and so on

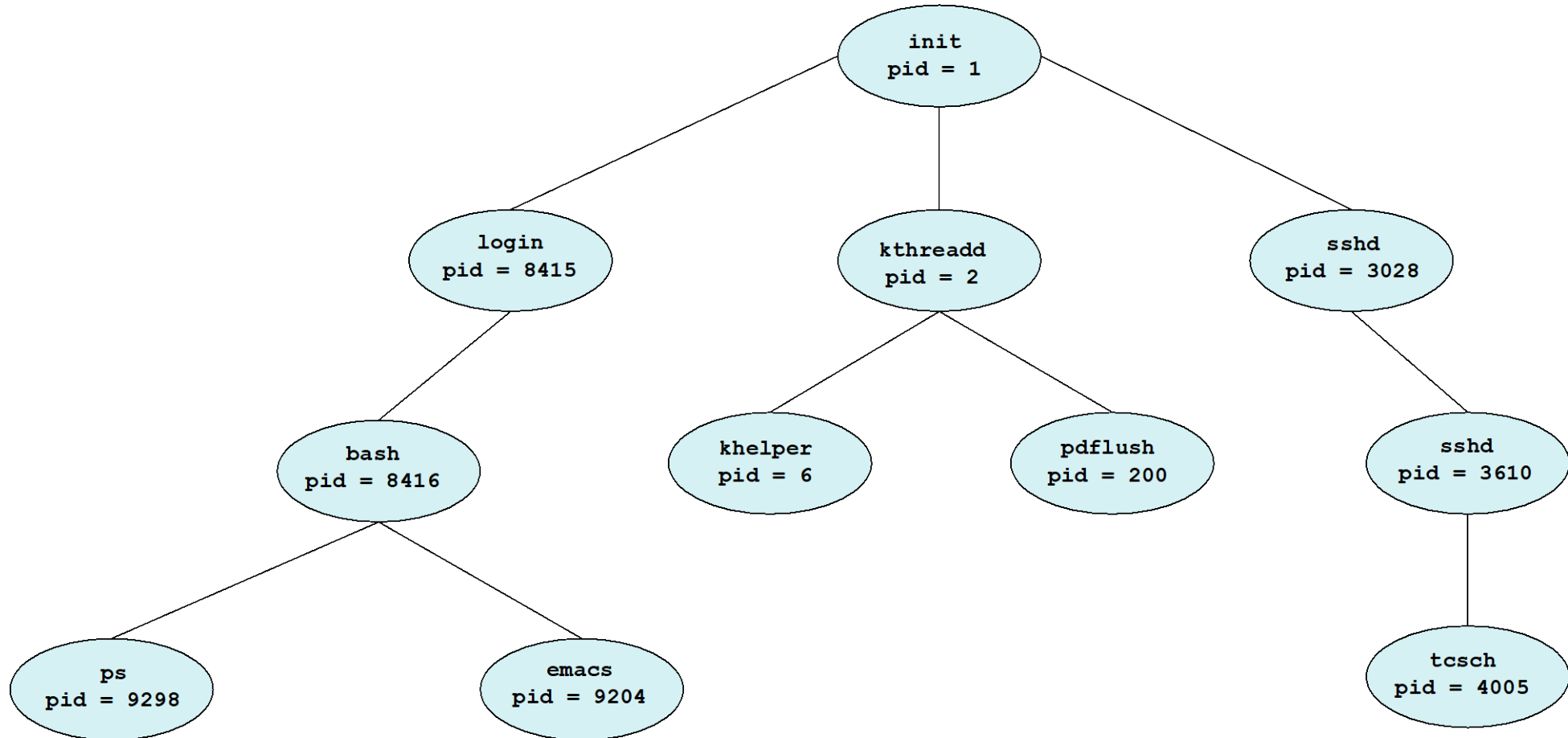
Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** (hierarchy) of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Issues
 - Will the parent and child execute **concurrently**?
 - How will the **address space** of the child be related to that of the parent?
 - Will the parent and child **share some resources**?

Process Creation (cont.)

- 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

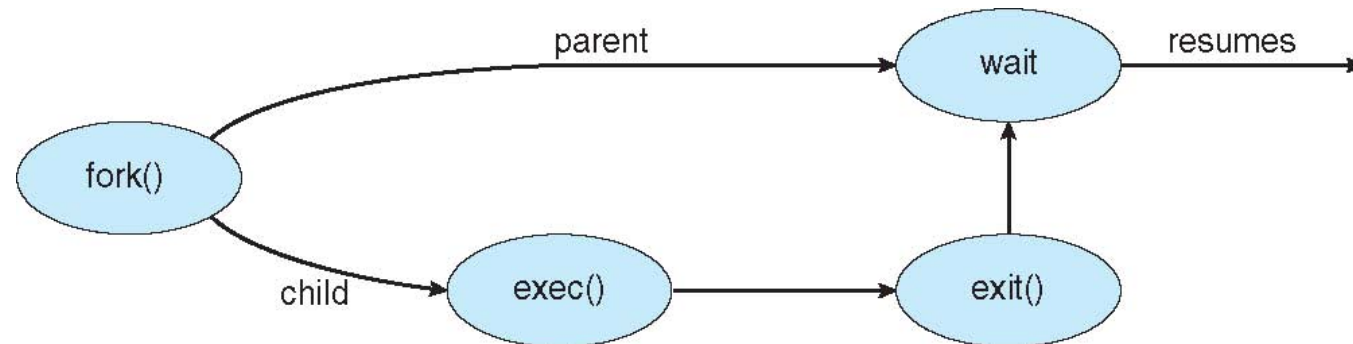


How to View Process Tree in Linux

- `% ps auxf`
 - `f` is the option to show the process tree
- `% pstree`

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



Process Creation in Linux

- Each process has a **process identifier (pid)**
- The parent executes `fork()` system call to spawn a child
- The child process has a **separate copy** of the parent's address space
- Both the parent and the child continue execution at the instruction following the `fork()` system call
- The return value for the `fork()` system call is
 - **Zero** value for the new (child) process
 - **Non-zero pid** for the parent process
- Typically, a process can execute a system call like `exec()` to load a binary file into memory

man page of `fork()`

- <http://man7.org/linux/man-pages/man2/fork.2.html>

RETURN VALUE top

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.

ERRORS top

EAGAIN A system-imposed limit on the number of threads was encountered. There are a number of limits that may trigger this error:

- * the `RLIMIT_NPROC` soft resource limit (set via `setrlimit(2)`), which limits the number of processes and threads for a real user ID, was reached;
- * the kernel's system-wide limit on the number of processes and threads, `/proc/sys/kernel/threads-max`, was reached (see `proc(5)`);
- * the maximum number of PIDs, `/proc/sys/kernel/pid_max`, was reached (see `proc(5)`); or
- * the PID limit (`pids.max`) imposed by the cgroup "process number" (PIDs) controller was reached.

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

- 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 exist 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) ;
```

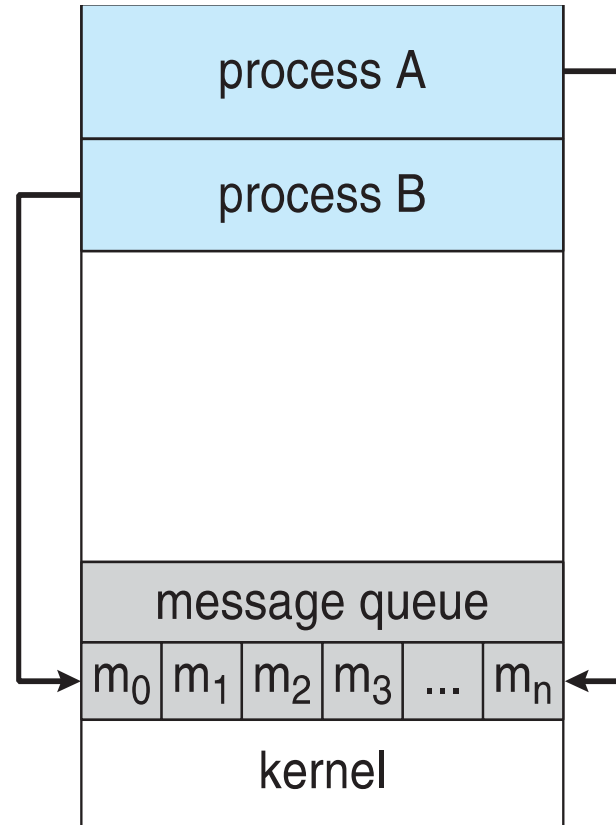
- If no parent waiting (did not invoke **wait()**) process is a **zombie**
- If parent terminated without invoking **wait**, process is an **orphan**

Inter-process Communication

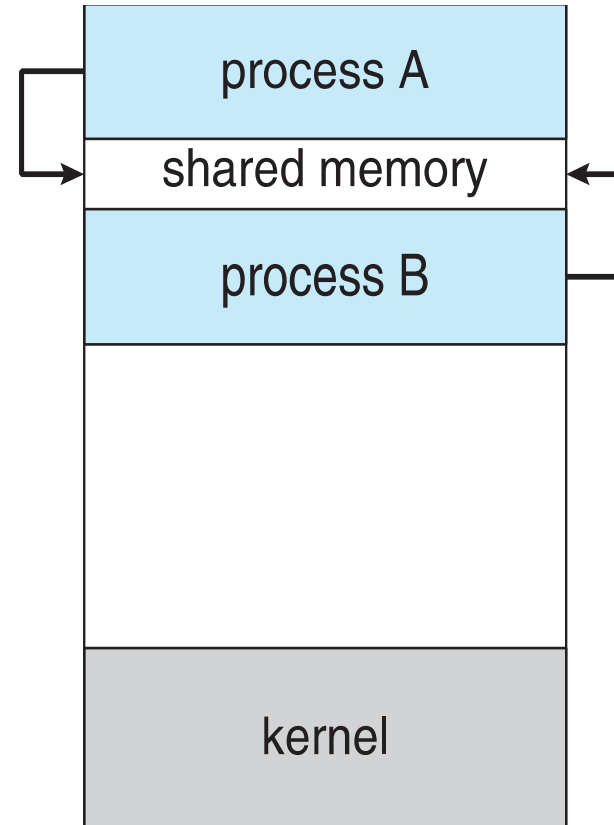
- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience
- Cooperating processes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

Communications Models

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



(a)



(b)

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

Inter-process 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 6 & 7.

Bounded-Buffer – Shared-Memory Solution

- Shared data

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

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

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Producer Process – Shared Memory

```
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;
}
```

Consumer Process – Shared Memory

```
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 */
}
```

Inter-process 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:
 - Direct or indirect
 - Synchronous or asynchronous
 - 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?
- 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

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
 - **Blocking send** -- the sender is blocked until the message is received
 - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
 - **Non-blocking send** -- the sender sends the message and continue
 - **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**

Synchronization (cont.)

- Producer-consumer becomes trivial

```
message next_produced;  
while (true) {  
    /* produce an item in next produced */  
    send(next_produced);  
}  
message next_consumed;  
while (true) {  
    receive(next_consumed);  
    /* consume the item in next consumed */  
}
```


Buffering

- Queue of messages attached to the link.
- implemented in one of three ways
 1. Zero capacity – no messages are queued on a link.
Sender must wait for receiver (rendezvous)
 2. Bounded capacity – finite length of n messages
Sender must wait if link full
 3. Unbounded capacity – infinite length
Sender never waits

Exit Slips

- Take 1-2 minutes to reflect on this lecture
- On a sheet of paper write:
 - One thing you learned in this lecture
 - One thing you didn't understand

Next class

- We will discuss:
 - Threads
- Reading assignment:
 - SGG: Ch. 4

Acknowledgment

- The slides are partially based on the ones from
 - The book site of *Operating System Concepts (Tenth Edition)*: <http://os-book.com/>