CSE 421/521 - Operating Systems
Fall 2013

Lecture - III

Processes

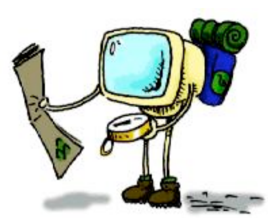
Tevfik Koşar

University at Buffalo

September 3rd, 2013

Roadmap

- Processes
  - Basic Concepts
  - Process Creation
  - Process Termination
  - Context Switching
  - Process Queues
  - Process Scheduling
  - Interprocess Communication



Process Concept

- a Process is a program in execution;

Pasta for six

- *boil 1 quart salty water*
- *stir in the pasta*
- *cook on medium until "al dente"*
- *serve*

thread of execution

Program

CPU

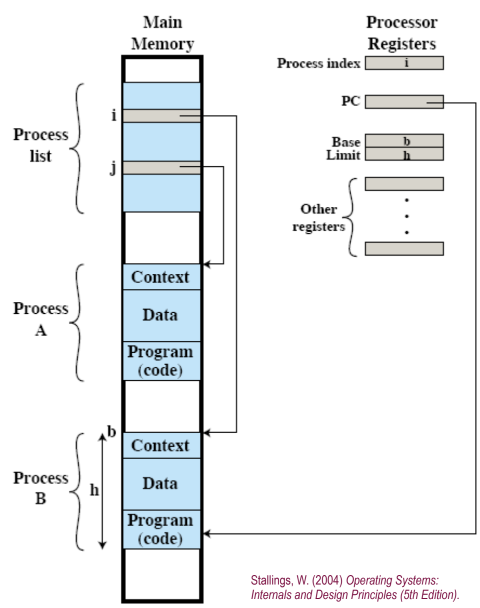
input data

Process

Process Concept

- a Process is a program in execution;
  - <u>A process image consists of three components</u>

user address space
1. an executable <u>program</u>
2. the associated <u>data</u> needed by the program
3. the execution <u>context</u> of the process, which contains all information the O/S needs to manage the process (ID, state, CPU registers, stack, etc.)

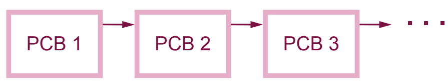


Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

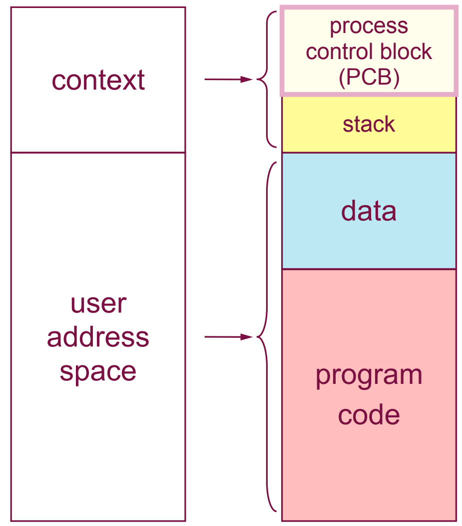
Typical process image implementation

Process Control Block

- The Process Control Block (PCB)
  - ✓ is included in the context, along with the stack
  - ✓ is a "snapshot" that contains all necessary and sufficient data to restart a process where it left off (ID, state, CPU registers, etc.)
  - ✓ is one entry in the operating system's **process table** (array or linked list)

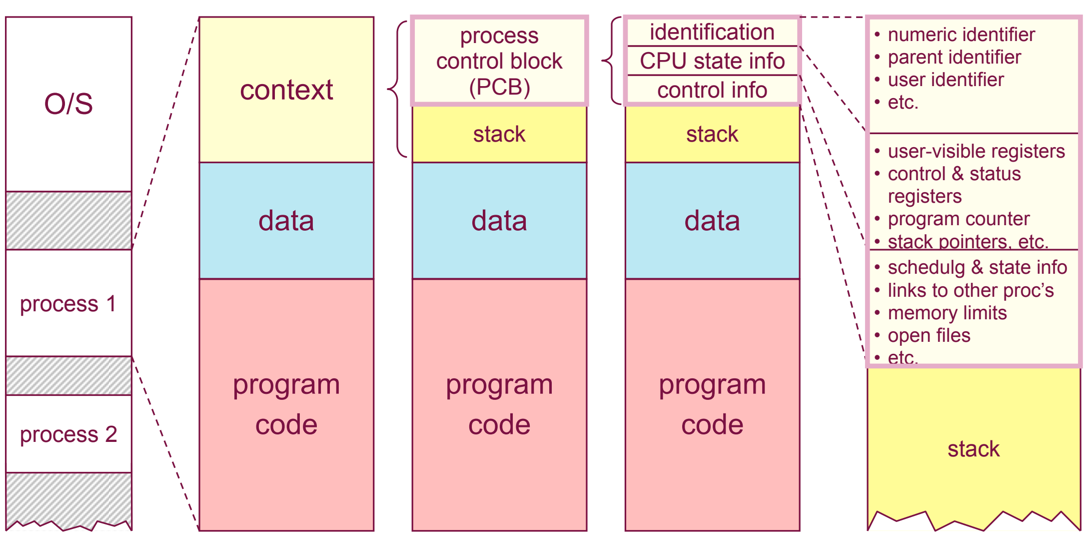


Typical process image implementation



Process Control Block

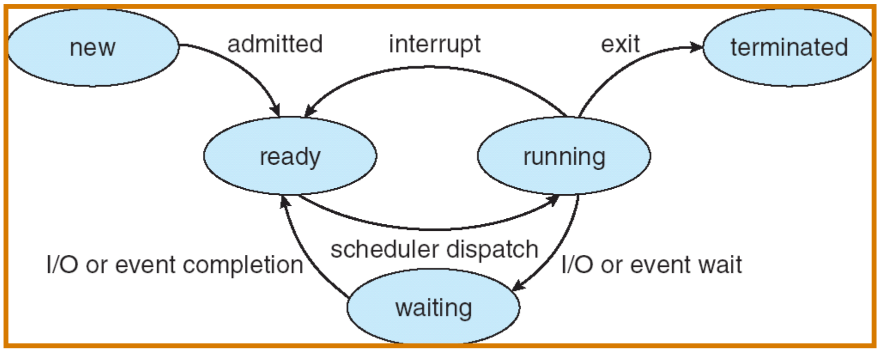
- Example of process and PCB location in memory



Illustrative contents of a process image in (virtual) memory

Process State

- As a process executes, it changes *state*
  - **new:** The process is being created
  - **ready:** The process is waiting to be assigned to a processor
  - **running:** Instructions are being executed
  - **waiting:** The process is waiting for some event to occur
  - **terminated:** The process has finished execution



Process Creation

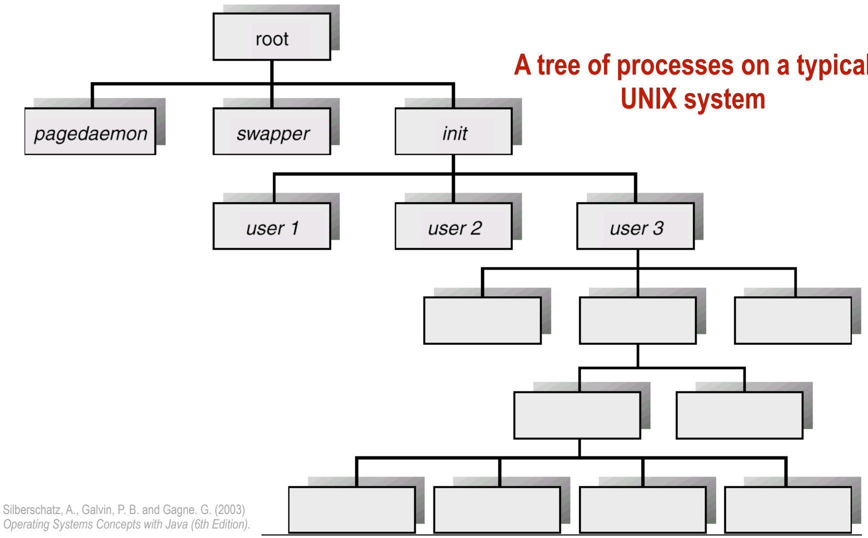
➢ <u>Some events that lead to process creation (enter)</u>

all cases of process spawning

- ✓ the system boots
  - when a system is initialized, several background processes or "daemons" are started (email, logon, etc.)
- ✓ a user requests to run an application
  - by typing a command in the CLI shell or double-clicking in the GUI shell, the user can launch a new process
- ✓ an existing process spawns a child process
  - for example, a server process (i.e. web server, file server) may create a new process for each request it handles
  - the *init* daemon waits for user login and spawns a shell
- ✓ a batch system takes on the next job in line

Process Creation

➢ Process creation by spawning



A tree of processes on a typical UNIX system

Silberschatz, A., Galvin, P. B. and Gagne, G. (2003)
Operating Systems Concepts with Java (6th Edition).

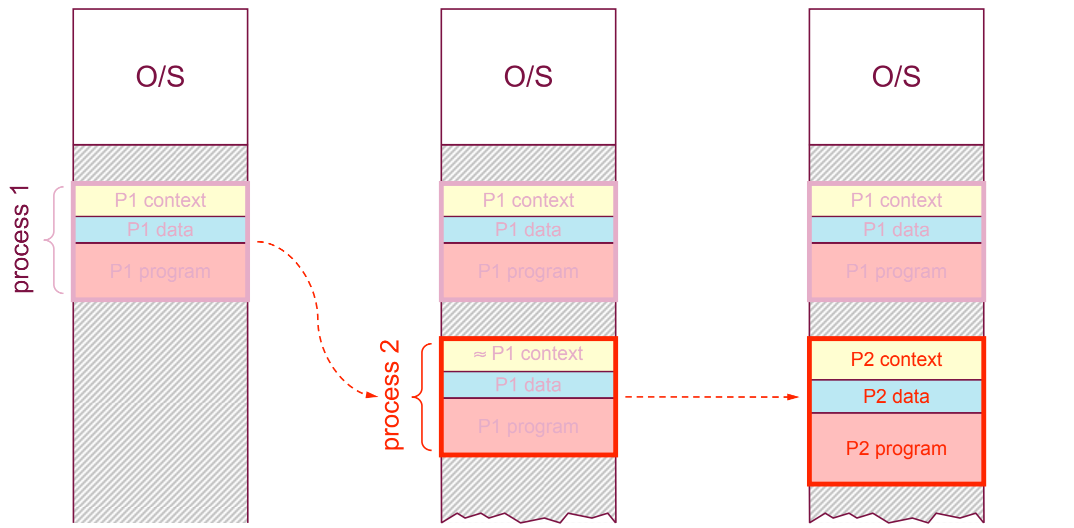
Process Creation

```
...
int main(...)
{
    ...
    if ((pid = fork()) == 0)                          // create a process
    {
        fprintf(stdout, "Child pid: %i\n", getpid());
        err = execvp(command, arguments);             // execute child
                                                      //   process
        fprintf(stderr, "Child error: %i\n", errno);
        exit(err);
    }
    else if (pid > 0)                                 // we are in the
    {                                                 //   parent process
        fprintf(stdout, "Parent pid: %i\n", getpid());
        pid2 = waitpid(pid, &status, 0);              // wait for child
        ...                                           //   process
    }
    ...

    return 0;
}
```

Process Creation

1. Clone child process
   - ✓ **pid = fork()**
2. Replace child's image
   - ✓ **execve(name, ...)**



Fork Example 1

```
    #include    <stdio.h>

main()
{
    int ret_from_fork, mypid;

    mypid = getpid();               /* who am i?    */
    printf("Before: my pid is %d\n", mypid);   /* tell pid */

    ret_from_fork = fork();

    sleep(1);
    printf("After: my fork returns pid : %d, said %d\n",
           ret_from_fork, getpid());
}
```

Fork Example 2

```
    #include    <stdio.h>

main()
{
    fork();
    fork();
    fork();
    printf("my pid is %d\n", getpid() );
}
```

How many lines of output will this produce?

Process Termination

- Some events that lead to process termination (exit)
  - ✓ regular completion, with or without error code
    - the process voluntarily executes an **`exit(err)`** system call to indicate to the O/S that it has finished
  - ✓ fatal error (uncatchable or uncaught)
    - service errors: no memory left for allocation, I/O error, etc.
    - total time limit exceeded
    - arithmetic error, out-of-bounds memory access, etc.
  - ✓ killed by another process via the kernel
    - the process receives a **`SIGKILL`** signal
    - in some systems the parent takes down its children with it

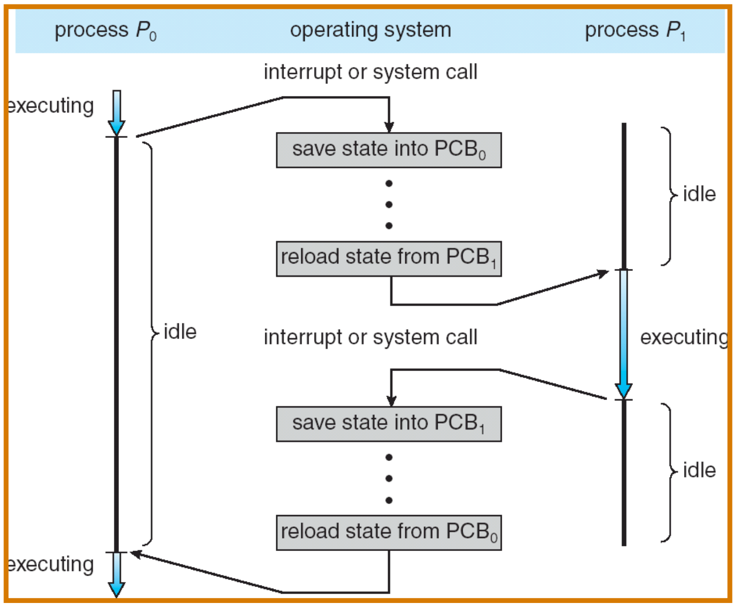
Process Pause/Dispatch

- **Some events that lead to process pause / dispatch**
  - ✓ I/O wait
    - O/S-triggered (following system call)
    - a process invokes an I/O system call that blocks waiting for the I/O device: the O/S puts the process in "Waiting" mode and dispatches another process to the CPU
  - ✓ preemptive timeout
    - the process receives a timer interrupt and relinquishes control back to the O/S dispatcher: the O/S puts the process in "Ready" mode and dispatches another process to the CPU
    - hardware interrupt-triggered (timer)
    - not to be confused with "total time limit exceeded", which leads to process termination

Process "Context" Switching

- 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
- Context-switch time is overhead; the system does no useful work while switching
- Switching time is dependent on hardware support

CPU Switch From Process to Process



Process "Context" Switching

➢ <u>How does a full process switch happen, step by step?</u>

1. save CPU context, including PC and registers *(the only step needed in a simple mode switch)*
2. update process state (to "Ready", "Blocked", etc.) and other related fields of the PCB
3. move the PCB to the appropriate queue
4. select another process for execution: this decision is made by the CPU scheduling algorithm of the O/S
5. update the PCB of the selected process (state = "Running")
6. update memory management structures
7. restore CPU context to the values contained in the new PCB

Process "Context" Switching

- What events trigger the O/S to switch processes?
  - ✓ **interrupts** — external, asynchronous events, independent of the currently executed process instructions
    - clock interrupt → O/S checks time and may block process
    - I/O interrupt → data has come, O/S may unblock process
    - memory fault → O/S may block process that must wait for a missing page in memory to be swapped in
  - ✓ **exceptions** — internal, synchronous (but involuntary) events caused by instructions → O/S may terminate or recover process
  - ✓ **system calls** — voluntary synchronous events calling a specific O/S service → after service completed, O/S may either resume or block the calling process, depending on I/O, priorities, etc.

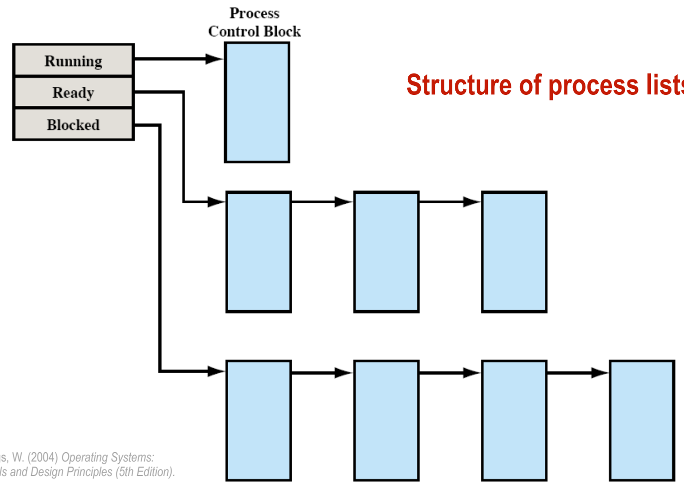
traps

Process Scheduling Queues

- **Job queue** – set of all jobs in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Process Queues

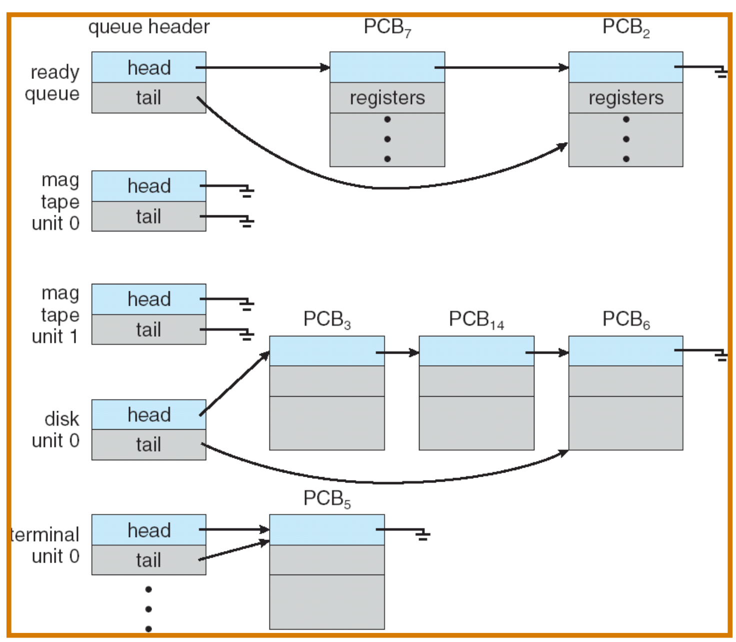
- ➢ The process table can be split into per-state queues
  - ✓ PCBs can be linked together if they contain a pointer field



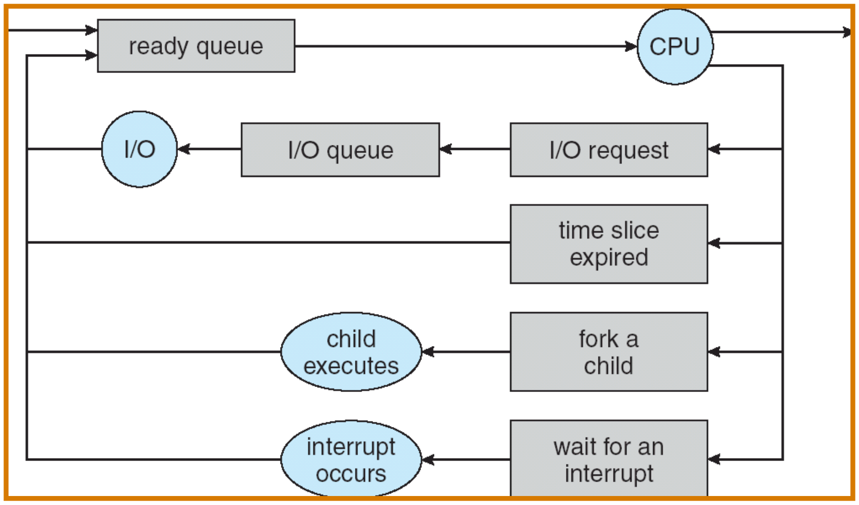
Structure of process lists or queues

Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

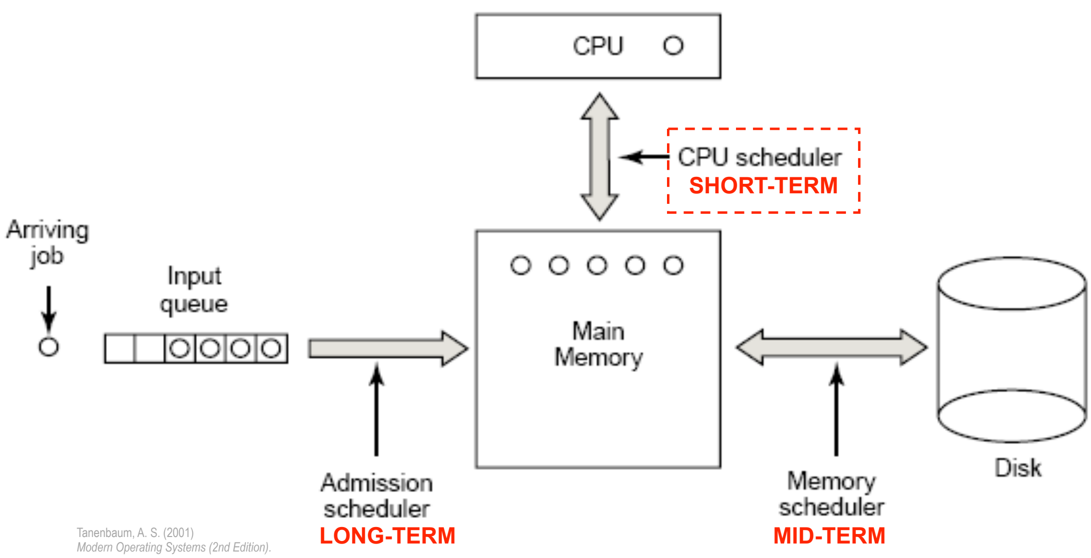
Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



Three Level CPU Scheduling



Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

Three-level scheduling

OS Scheduling

- <u>**Long-term scheduling**</u>
  - ✓ the decision to add a program to the pool of processes to be executed (job scheduling)
- <u>**Medium-term scheduling**</u>
  - ✓ the decision to add to the number of processes that are partially or fully in main memory ("swapping")
- <u>**Short-term scheduling = CPU scheduling**</u>
  - ✓ the decision as to which available processes in memory are to be executed by the processor ("dispatching")
- <u>**I/O scheduling**</u>
  - ✓ the decision to handle a process's pending I/O request

fine- to coarse-grain level

frequency of intervention

Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

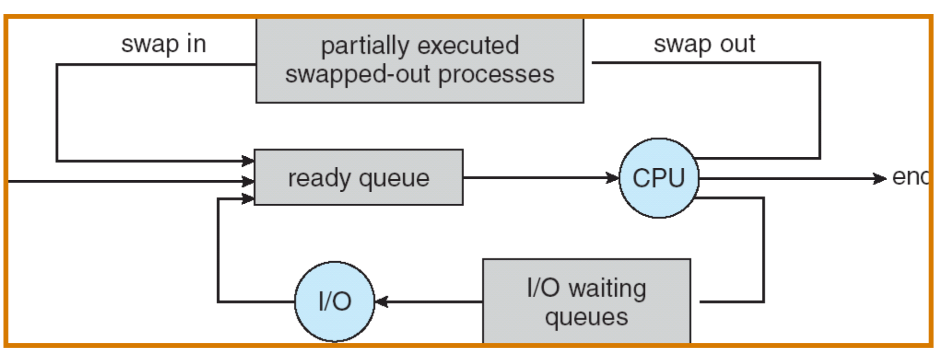
Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast)
- Long-term scheduler is invoked very 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 schedulers need to make careful decision

Addition of Medium Term Scheduling

- In time-sharing systems: remove processes from memory "temporarily" to reduce degree of multiprogramming.
- Later, these processes are resumed ➔ Swapping



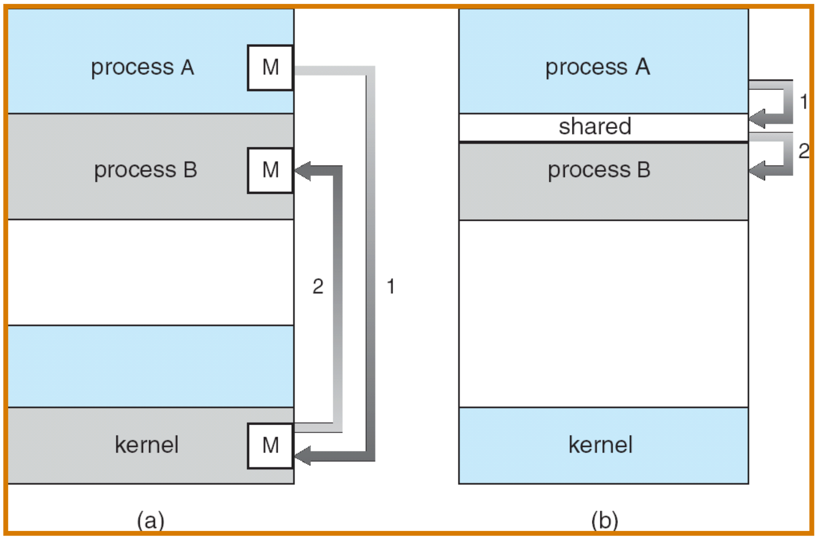
Cooperating Processes

- **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
- Disadvantage
  - Synchronization issues and race conditions

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Shared Memory: by using the same address space and shared variables
- Message Passing: processes communicate with each other without resorting to shared variables

Communications Models



a) Message Passing b) Shared Memory

Message Passing

- Message Passing facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Two types of Message Passing
  - direct communication
  - indirect communication

Message Passing – 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
- Symmetrical vs Asymmetrical direct communication
  - **send** (*P, message*) – send a message to process P
  - **receive**(id, *message*) – receive a message from any process
- Disadvantage of both: limited modularity, hardcoded

Message Passing - 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
- Primitives are defined as:

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

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

Indirect Communication *(cont.)*

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy 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 *(cont.)*

- 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

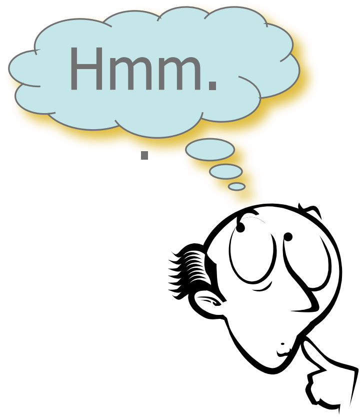
- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**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

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     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

Summary

- Processes
  - Basic Concepts
  - Process Creation
  - Process Termination
  - Context Switching
  - Process Queues
  - Process Scheduling
  - Interprocess Communication



- No Class this Thursday
- Next Lecture: Threads
- Reading Assignment: Chapter 4 from Silberschatz.
- HW 1 will be out next week

Acknowledgements

- "Operating Systems Concepts" book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne
- "Operating Systems: Internals and Design Principles" book and supplementary material by W. Stallings
- "Modern Operating Systems" book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from UNR