Processes

Pertemuan 5 dan 6

Kompetensi Khusus

- Mahasiswa mampu menjelaskan bagaimana proses SO dalam mengatur pengelolaan yang dilakukan oleh sistem komputer (C2, A2)(C1)

Materi

1. Process Concept
2. Process Scheduling
3. Operating on Processes
4. Interprocess Communication
5. IPC in Shared-Memory System
6. IPC in Message-Passing System
7. Communication in Client-Server System

1. Process Concept

Process Concept

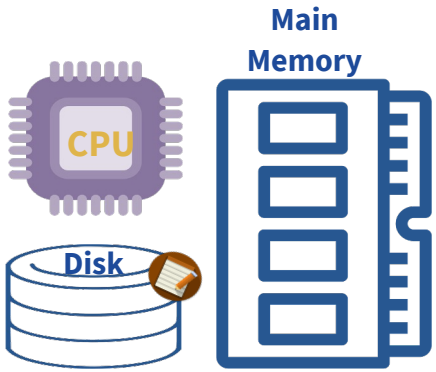
- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms job and process almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

Process Concept (Cont.)

- Program is **passive** entity stored on disk (**executable file**); process is **active**
  - Program becomes process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - Consider multiple users executing the same program

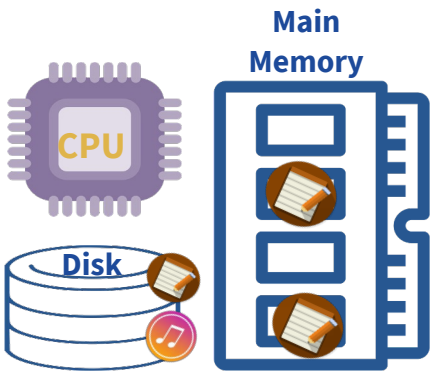
What is a process?

OS manages hardware on behalf of applications



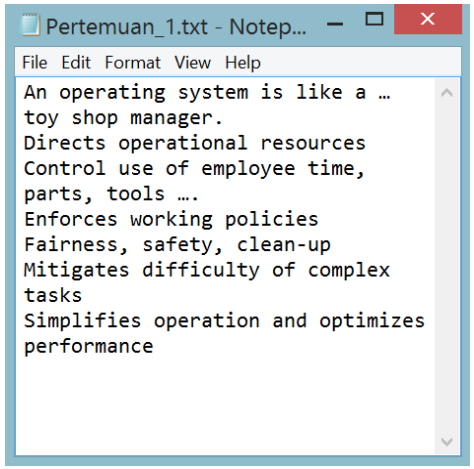
Application → program on disk, flash memory …. (static entity)

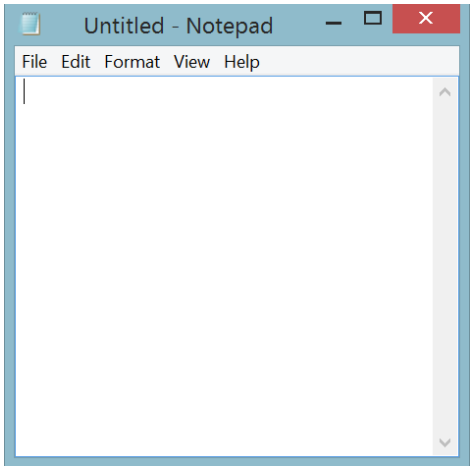
Process → state of a program when executing loaded in memory (active entity)



What is a process?

So, process therefore **represents the execution state of an active application.**





Process in Memory

A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section**, which contains global variables. A process may also include a **heap**, which is memory that is dynamically allocated during process run time.

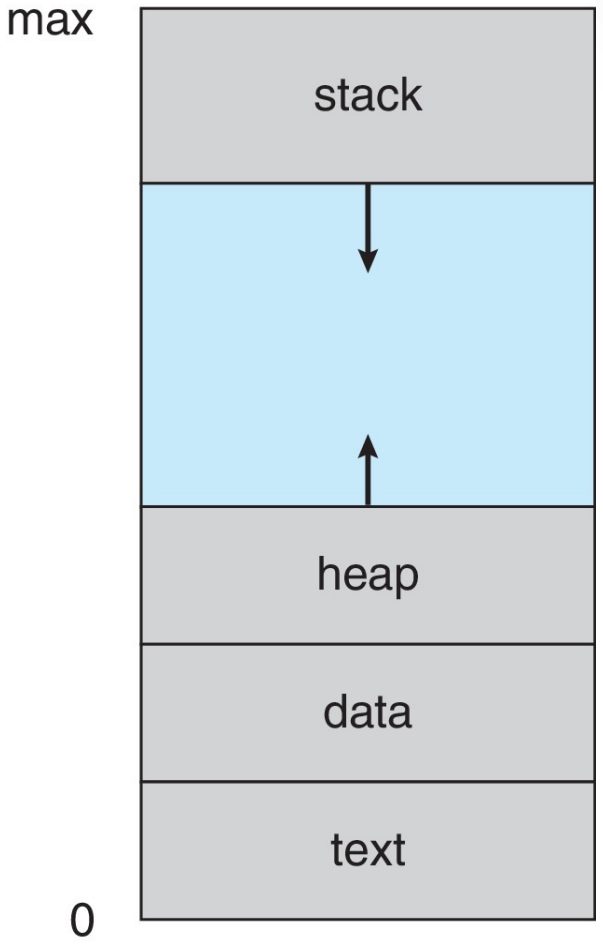
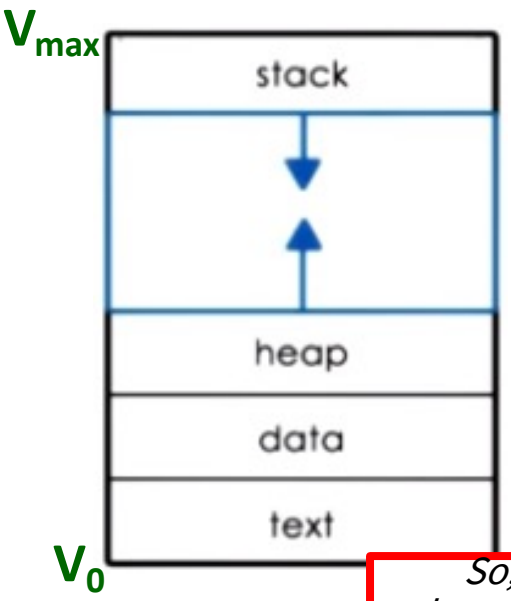


Fig. Process in memory.

What does a process look like?

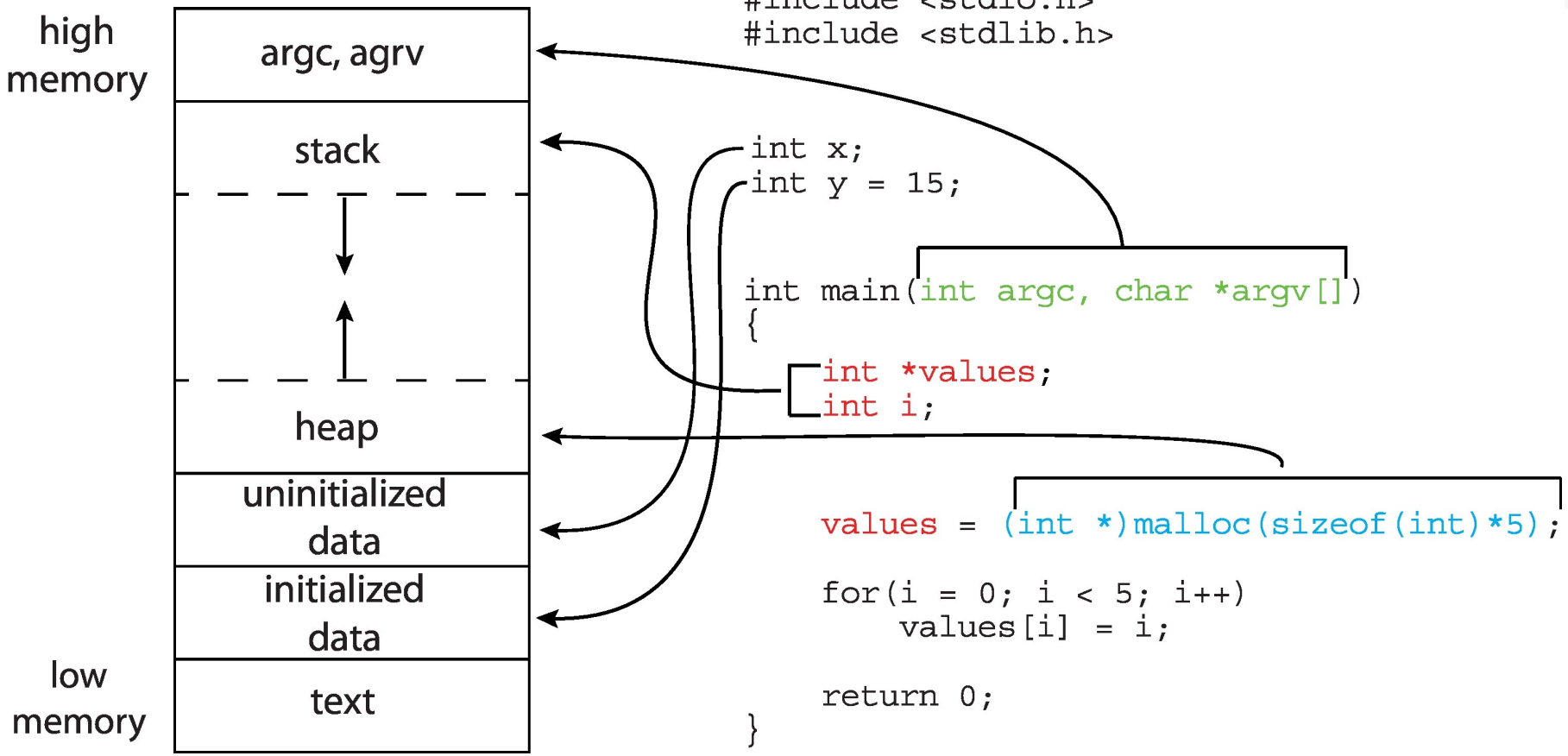


So, an OS abstraction used to encapsulates all of the process state is an address space.

Types of state in a process address space:

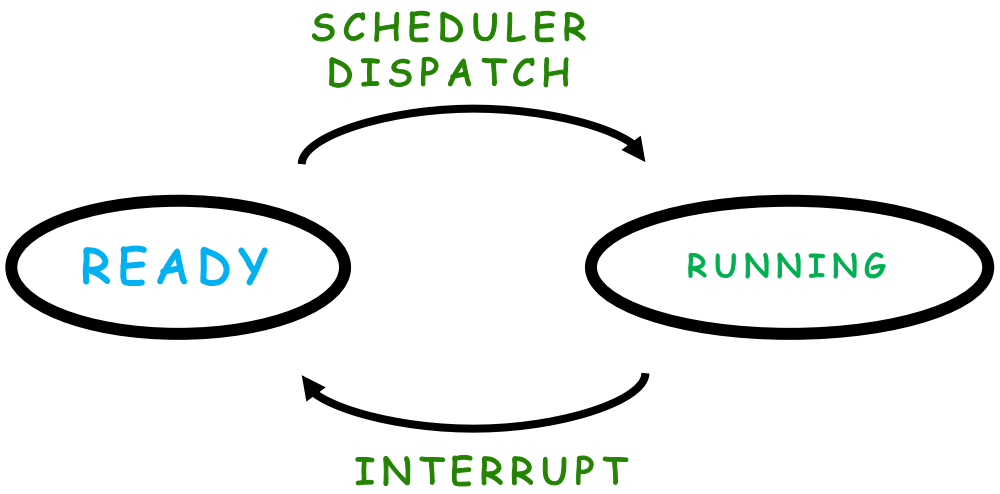
- **Text and data**
  - *static state when process first loads*
  - *the program code/the executable code*
  - *global variables*
- **Heap Section**
  - *memory that is dynamically allocated during program run time (dynamically created during execution)*
- **Stack**
  - *temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)*
  - *Grow and shrinks ~ LIFO*

Memory Layout of a C Program



Process Life Cycle

- Processes can be **running** or **idle**



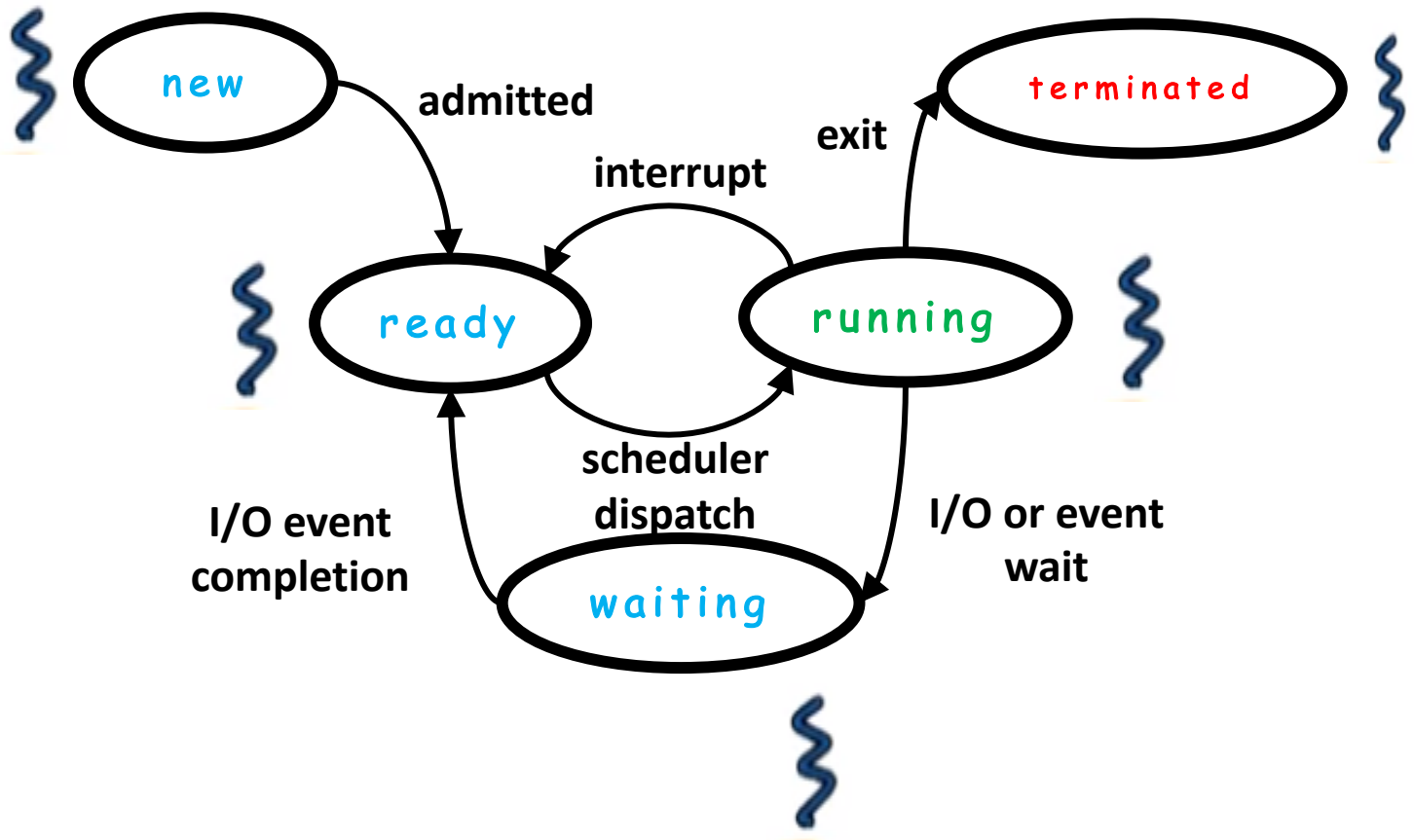
What other states can a process be in?

How is that determined?

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 process
  - **terminated**: The process has finished execution

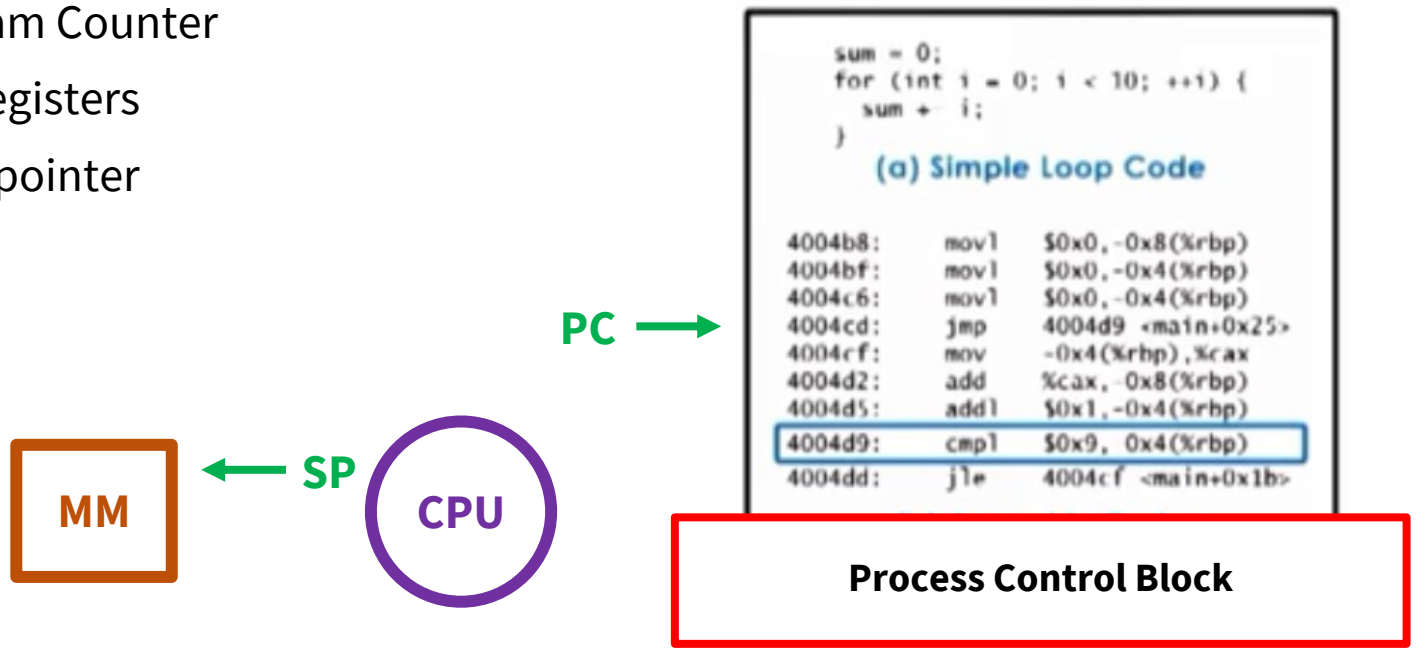
Process Life Cycle: States



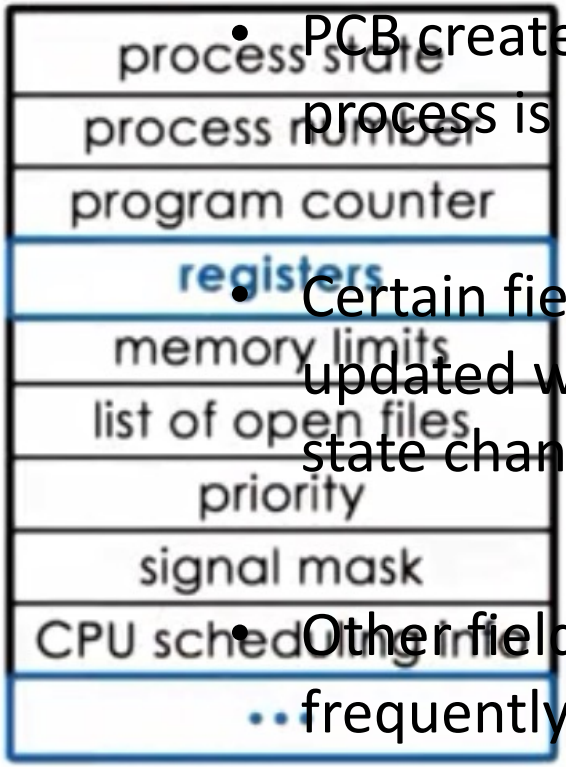
Process Control Block (PCB)

How does the OS know what a process is doing?

- Program Counter
- CPU registers
- Stack pointer



Process Control Block (PCB)

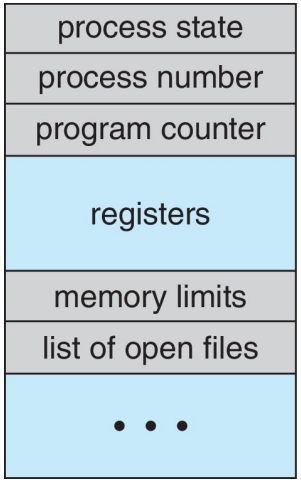


- PCB created when process is created
- Certain fields are updated when process state changes
- Other fields change too frequently

Process Control Block (PCB)

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

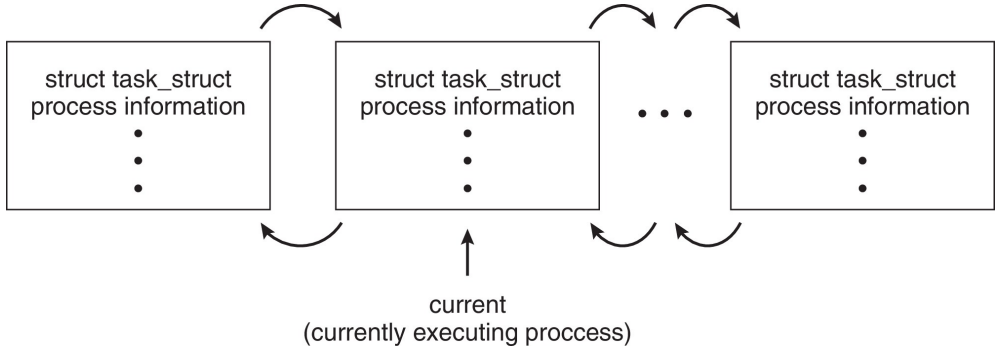
- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



Example of Process Representation in Linux

Represented by the C structure `task_struct`

```
pid t_pid;                      /* process identifier */
long state;                     /* state of the process */
unsigned int time_slice         /* scheduling information */
struct task_struct *parent;  /* this process's parent */
struct list_head children;   /* this process's children */
struct files_struct *files;  /* list of open files */
struct mm_struct *mm;           /* address space of this process
*/
```

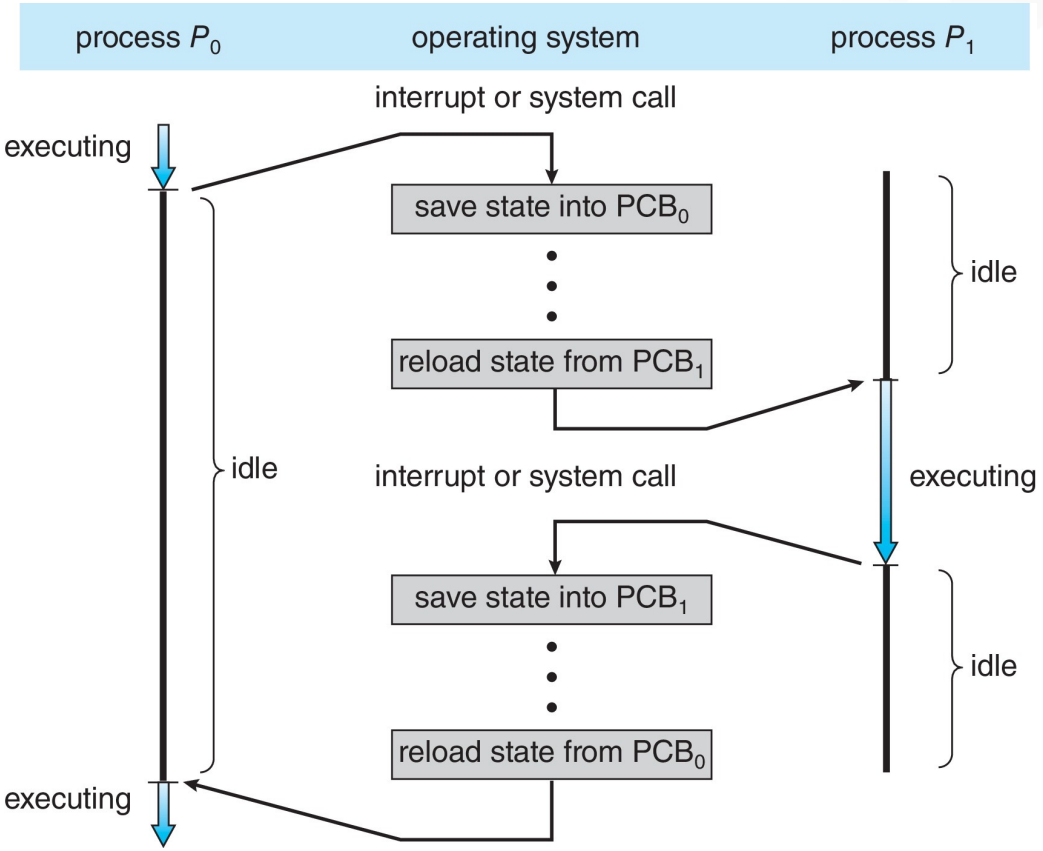


Context Switch

- 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 pure 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

CPU Switch From Process to Process

- A **context switch** occurs when the CPU switches from one process to another.



2. Process Scheduling

Process Scheduling

- The objective of multiprogramming is **to have some process running at all times** so as **to maximize CPU utilization**.
- The objective of time sharing is **to switch a CPU core among processes so frequently that users can interact with each program while it is running**.
- To meet these objectives, the **process scheduler selects an available process** (possibly from a set of several available processes) **for program execution on a core**.
- Each CPU core can run one process at a time.

Process Scheduling

- For a system with a single CPU core, there will never be more than one process running at a time, whereas **a multicore system can run multiple processes at one time**.
- If there are more processes than cores, excess processes will have to wait until a core is free and can be rescheduled.
- The number of processes currently in memory is known as the **degree of multiprogramming**.

Representation of Process Scheduling

Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

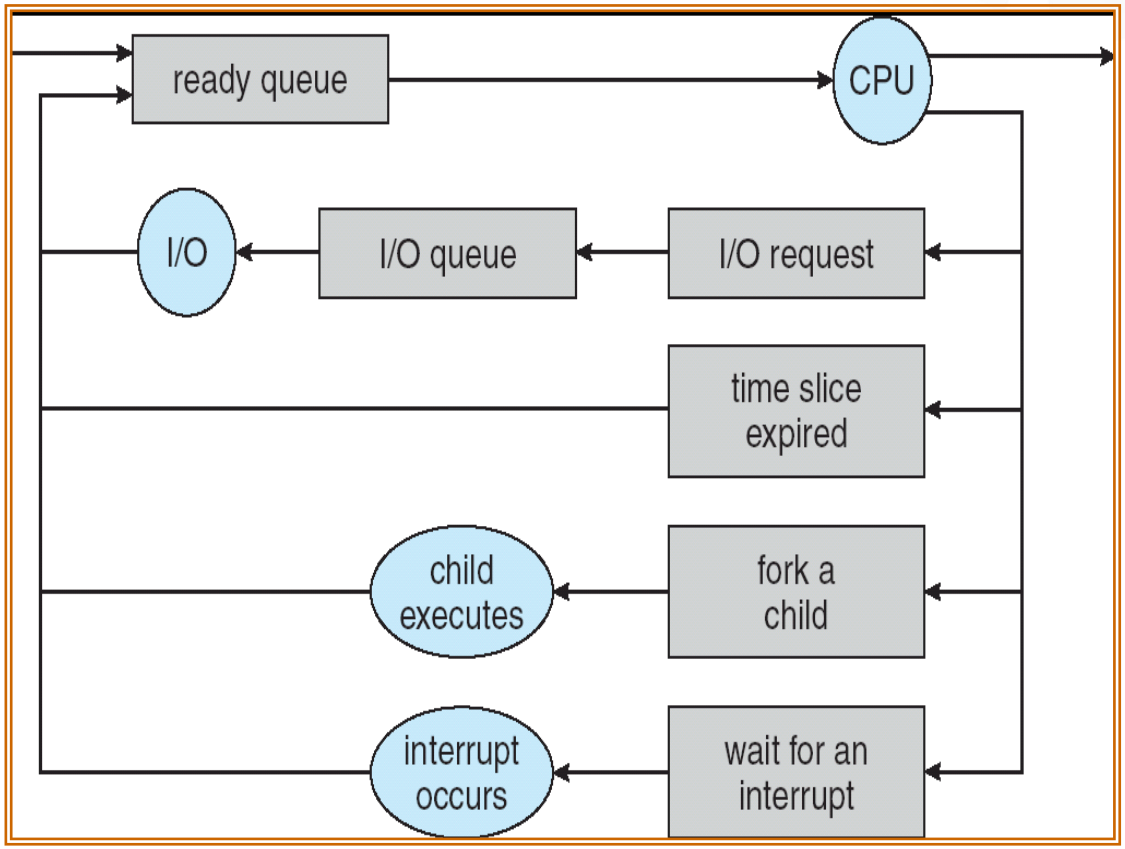


Fig. Queueing-diagram representation of process scheduling.

Ready Queue And Various I/O Device Queues

Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue.

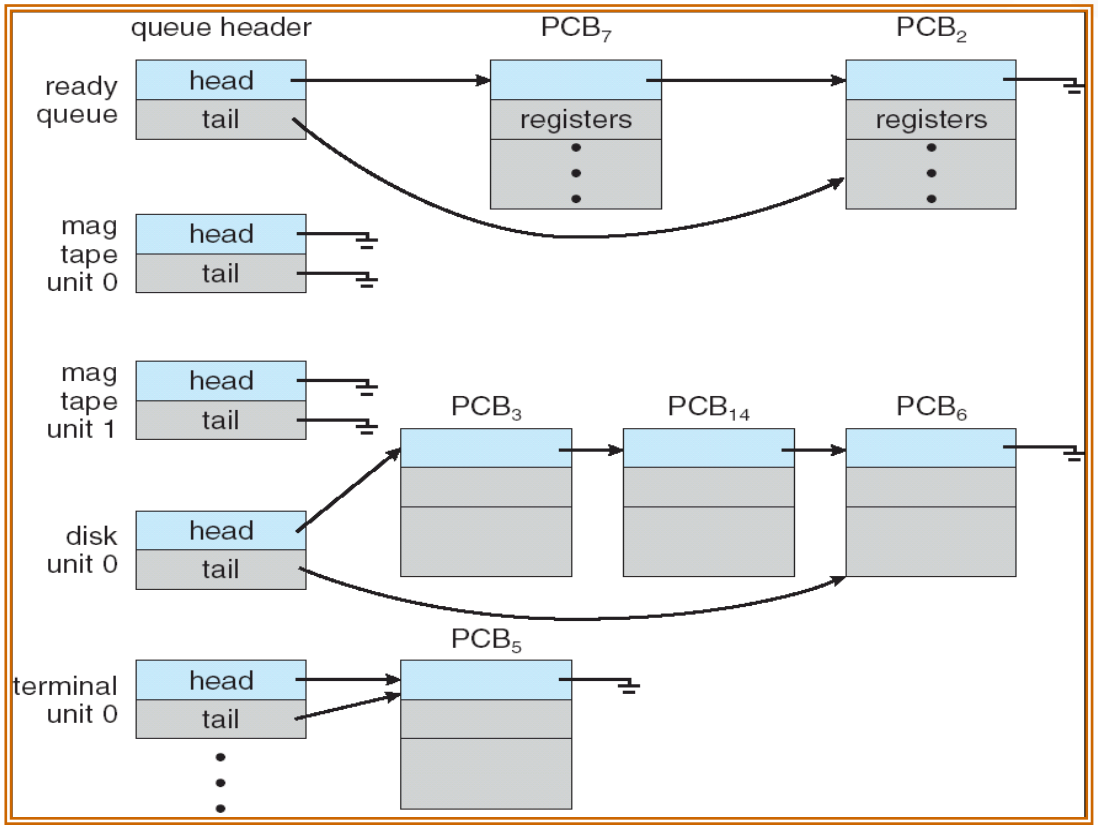


Fig. The ready queue and various I/O device queues.

Process Scheduling Queues

- Job queue – set of all processes 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

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) --> (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) --> (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

Addition of Medium Term Scheduling

The key idea behind a medium-term scheduler is that **sometimes it can be advantageous to remove processes from memory** (and from active contention for the CPU) **and thus reduce the degree of multiprogramming**.

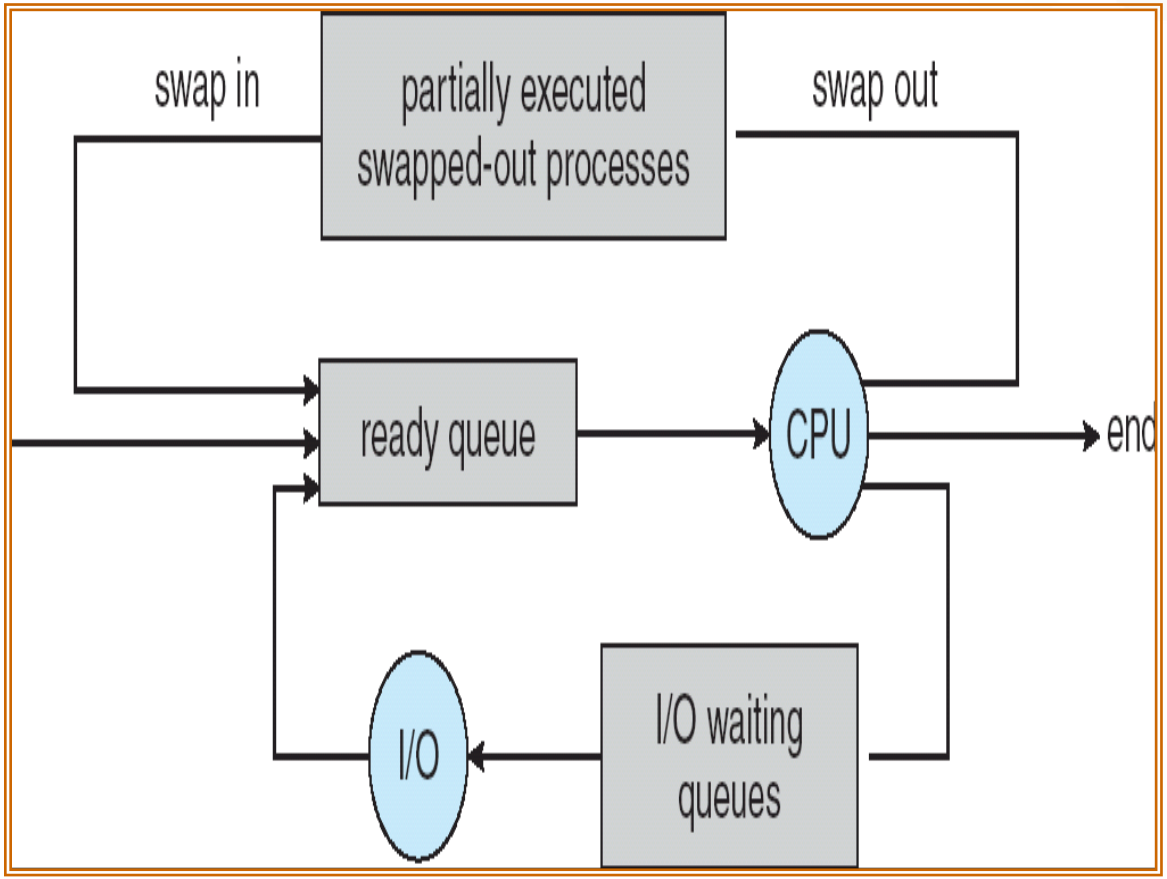


Fig. Addition of medium-term scheduling to the queueing diagram

3. Operations on Processes

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

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
  - Parent process calls **`wait()`** waiting for the child to terminate

Process Creation (Cont.)

As an alternative example, we next consider process creation in Windows. Processes are created in the Win32 API using the CreateProcess() function, which is similar to fork () in that a parent creates a new child process.

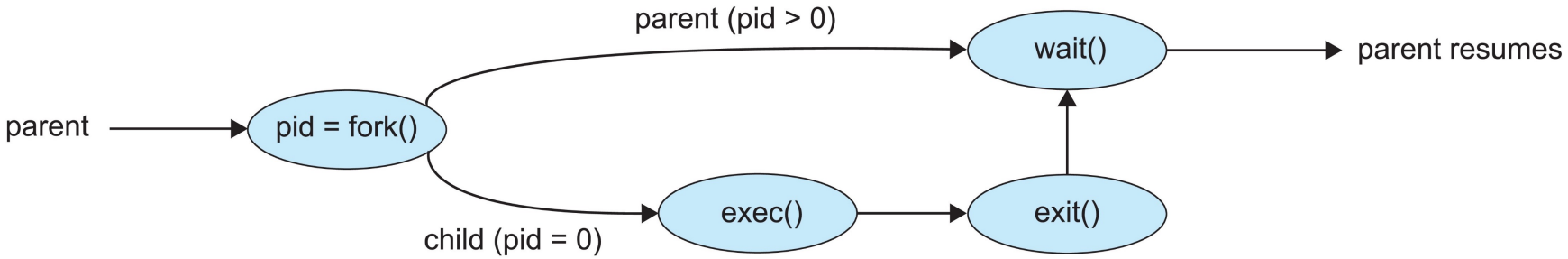
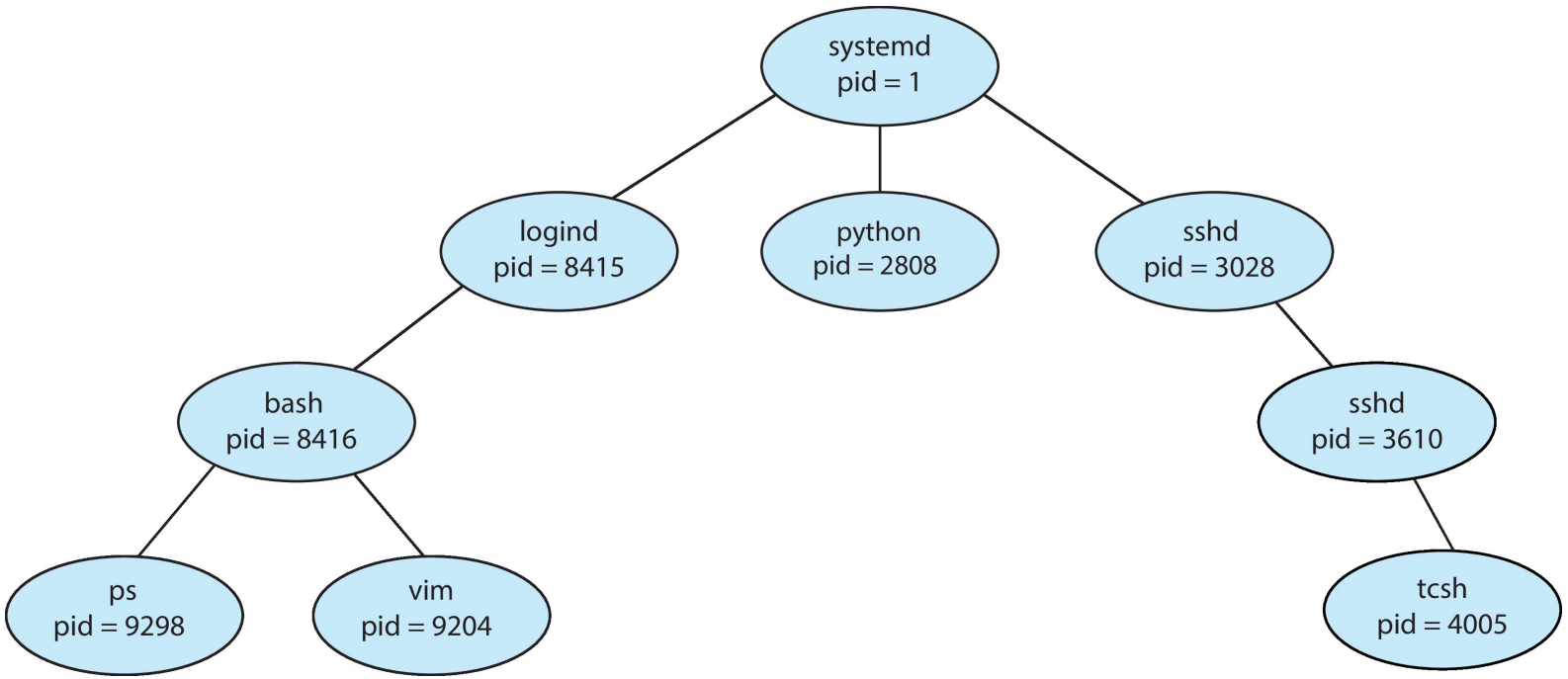


Fig. Process creation

A Tree of Processes in Linux



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;
}
```

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);
```

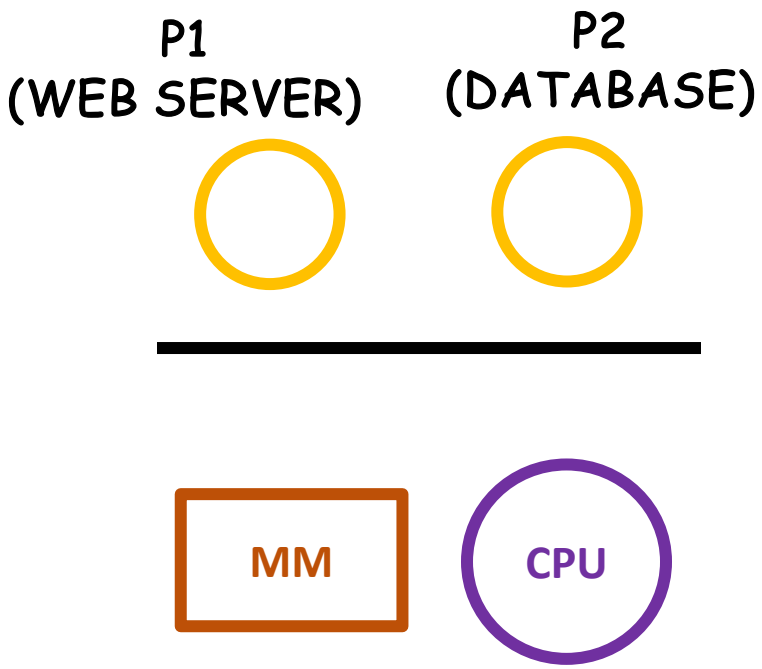
- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
- If parent terminated without invoking **`wait()`**, process is an **orphan**

4. Interprocess Communication

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**

Inter Process Communication



Inter-Process Communication (IPC) mechanisms:

- Transfer data/into between address spaces
- Maintain protection and isolation
- Provide flexibility and performance

Communications Models

There are two fundamental models of interprocess communication: (1) shared memory and (2) message passing. In the shared-memory model, a region of memory that is shared by cooperating processes is established.

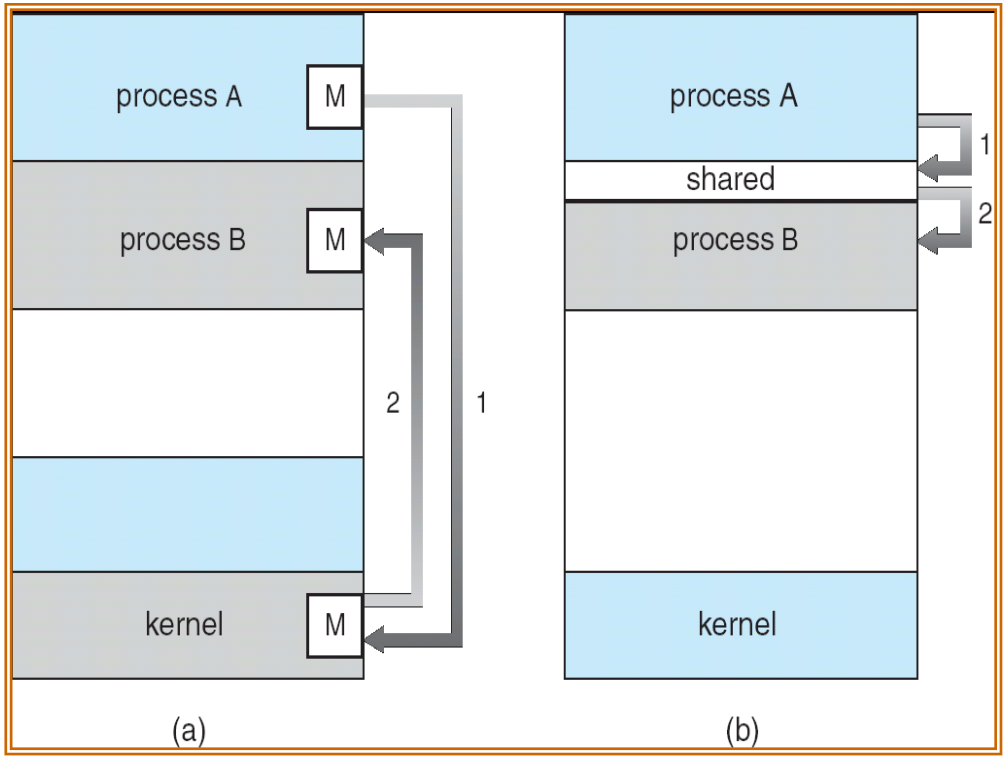
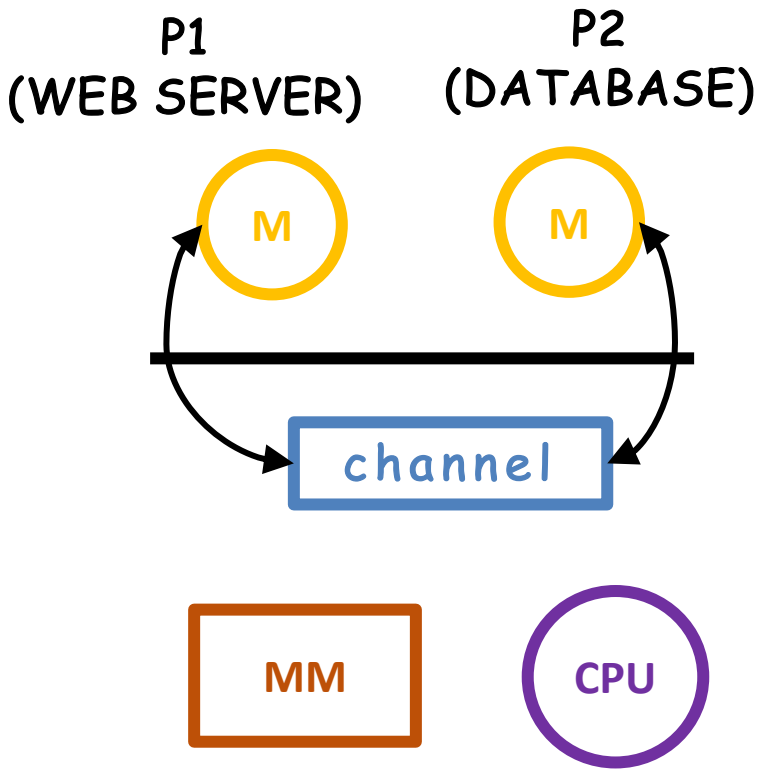


Fig. Communications models. (a) Shared Memory. (b) Message Passing.

Inter Process Communication



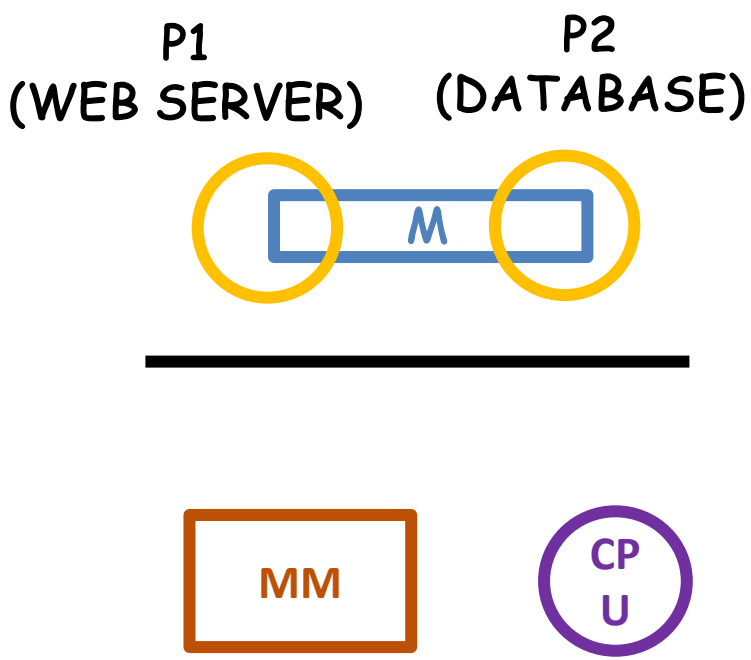
Message-passing IPC :

- OS provides communication channel, like shared buffer
- Process
  - Write (send)/ read (recv)
  - Messages to/from channel

+ : OS manages

- : overheads

Inter Process Communication



Shared Memory IPC :

- OS establishes a shared channel and maps it into each process address space
- Processes directly read/write from this memory

+ : OS is out of the way

- : (re-) implement code due error prone

5. Communication in Client-Server System

5.1 Client-Server Communication

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

5.2 Sockets

- A socket is defined as an endpoint for communication
- Concatenation of IP address and port
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication consists between a pair of sockets

5.2.1 Socket Communication

For example, if a client on host X with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at address 161.25.19.8, host X may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host X and (161.25.19.8:80) on the web server.

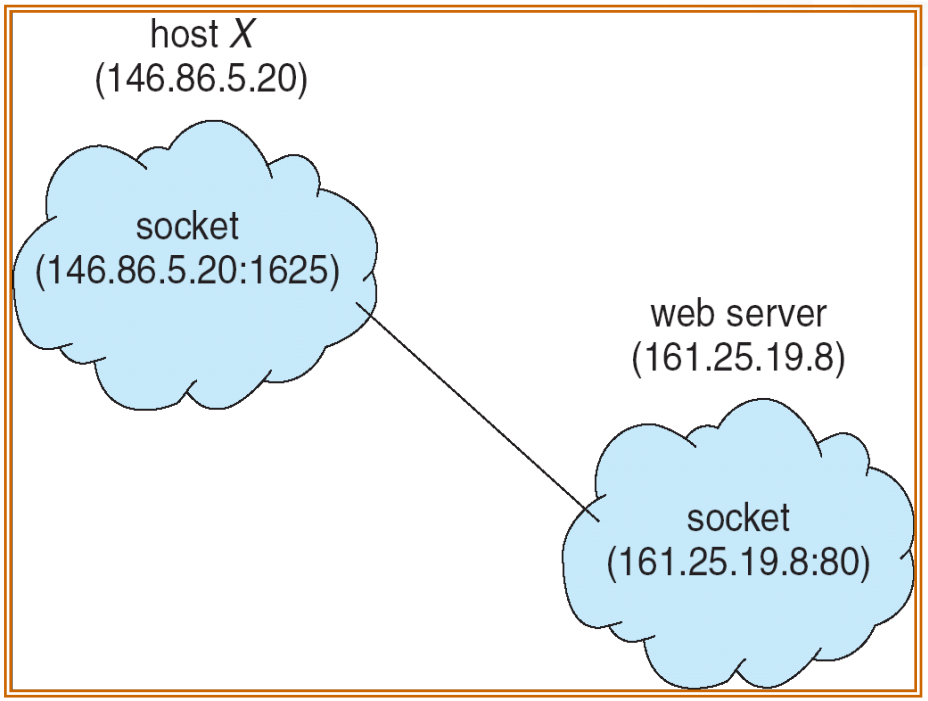


Fig. Communication using sockets.

5.3 Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- **Stubs** – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and marshalls the parameters.
- The server-side stub receives this message, unpacks the marshalled parameters, and peforms the procedure on the server.

5.3.1 Execution of RPC

The port number is returned, and the RPC calls can be sent to that port until the process terminates (or the server crashes). This method requires the extra overhead of the initial request but is more flexible than the first approach.

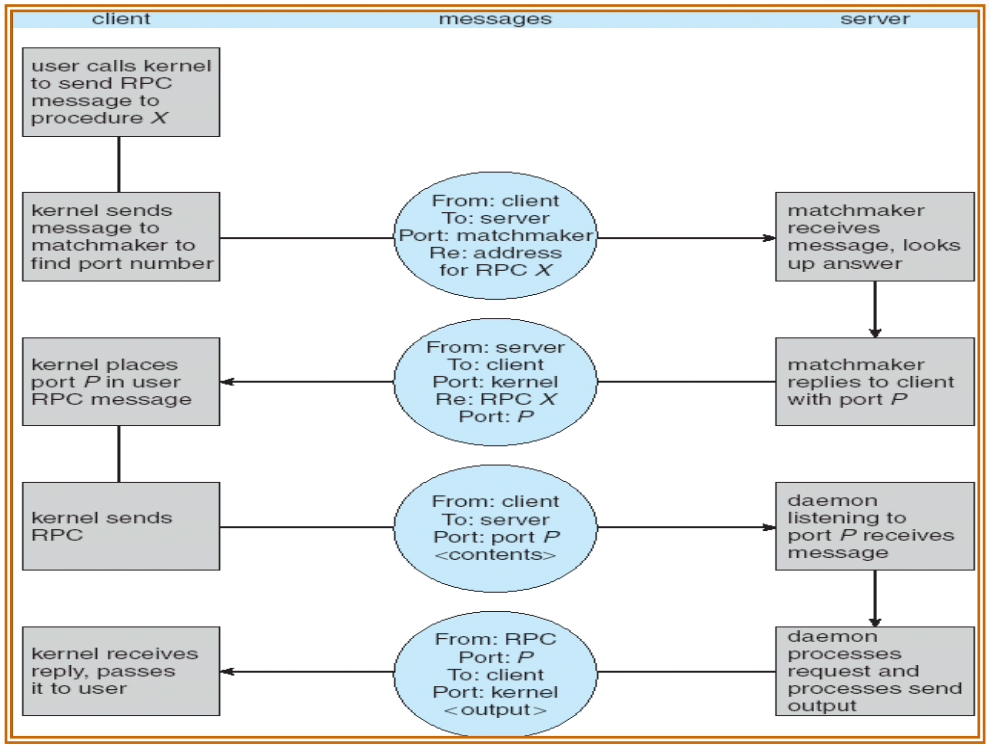


Fig. Execution of a remote procedure call (RPC).

5.4 Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.

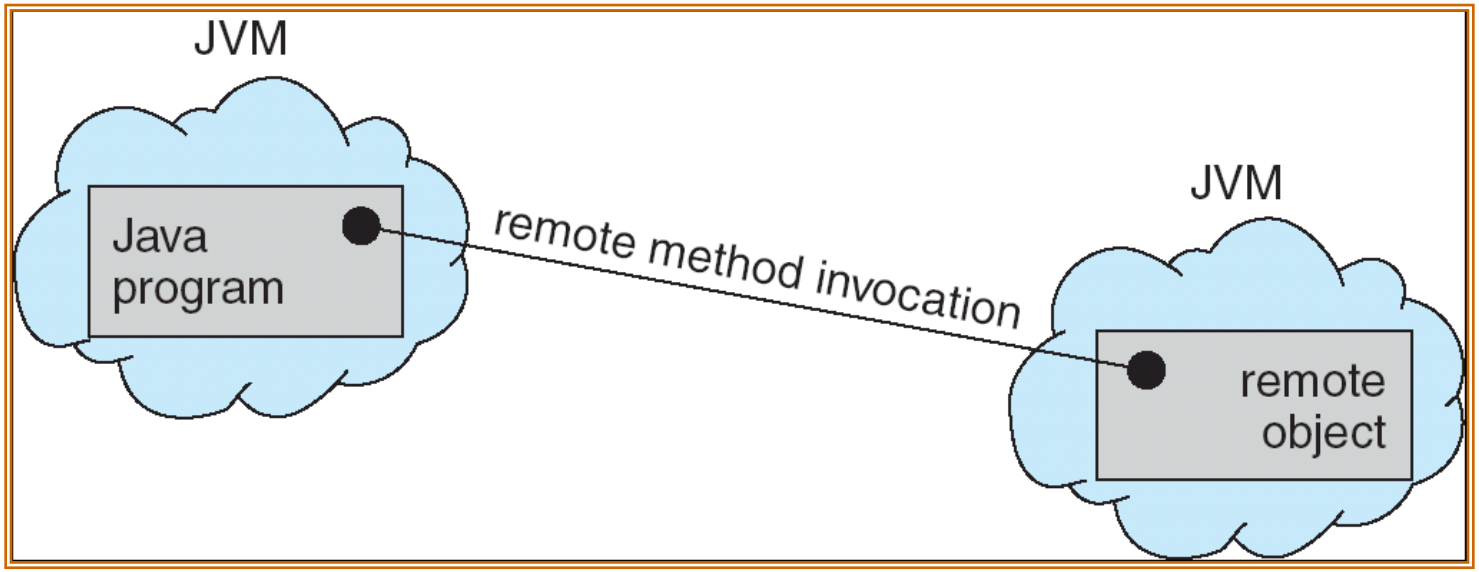


Fig. Remote method invocation

5.4.1 Marshalling Parameters

The actual implementation of someMethod () resides on the server. Once the method is completed, the skeleton marshals the boolean value returned from someMethod () and sends this value back to the client. The stub unmarshals this return value and passes it to the client.

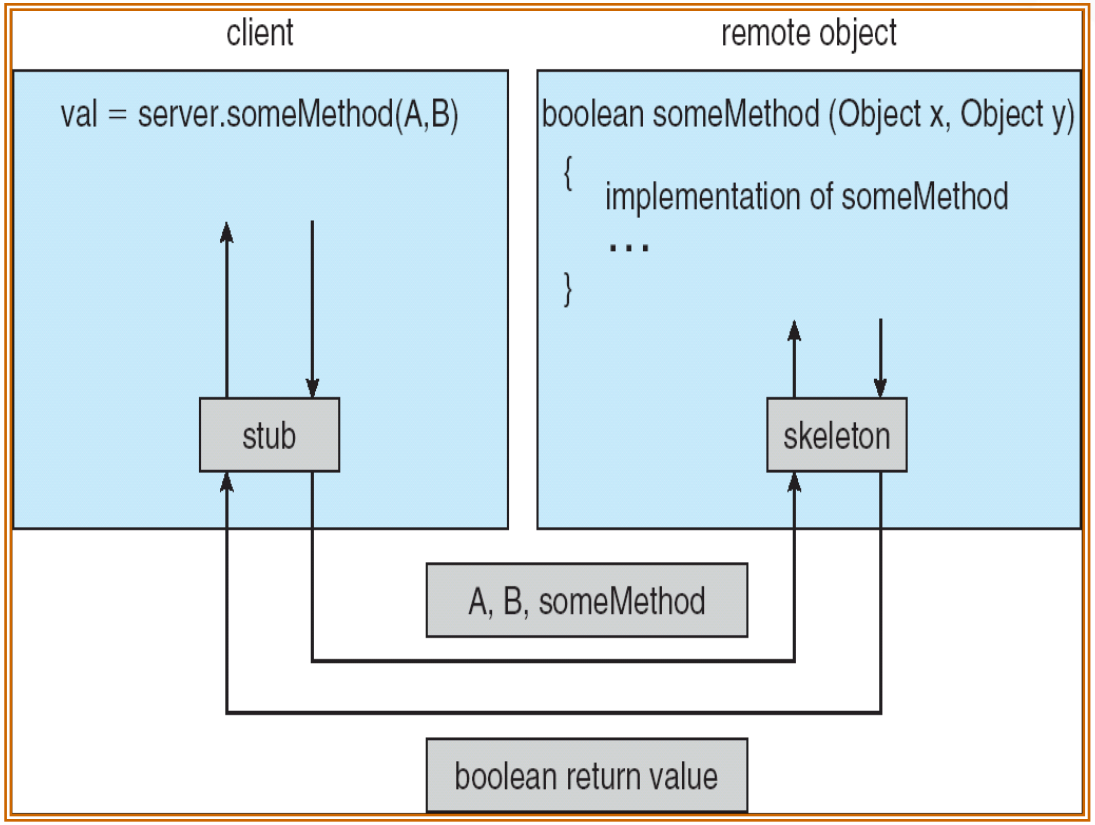


Fig. Marshalling parameters

Summary

- Process State consist of new, running, waiting, ready, and terminated
- 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
- Client-Server Communication consist of Sockets, Remote Procedure Calls, and Remote Method Invocation (Java)
- A socket is defined as an endpoint for communication

Thank You