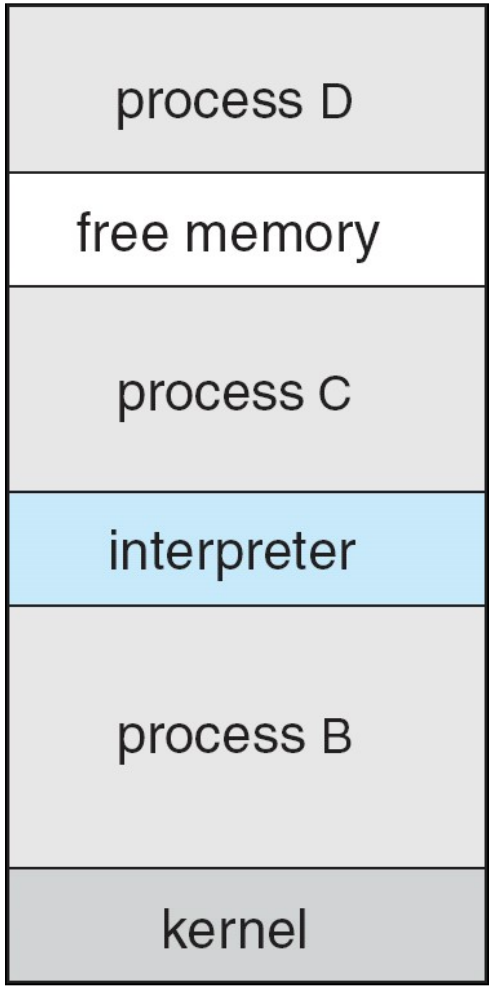
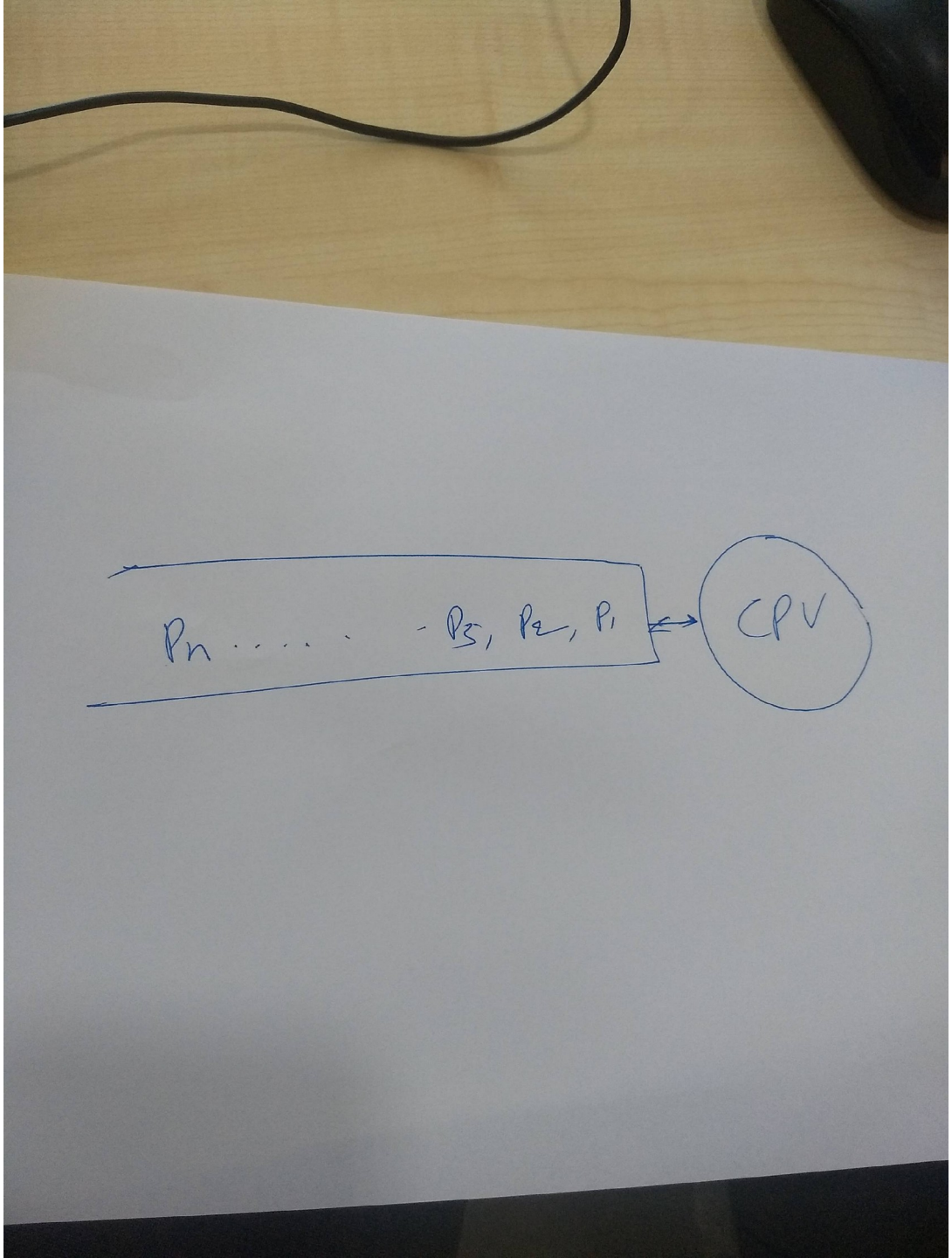
Chapter 3: Process Concept

- Process Concept
- Process Scheduling
- Operations on Processes
- Cooperating Processes
- Interprocess Communication

FreeBSD Running Multiple Programs




Pn P3, P2, P1
CPU

An Analogy

- Example
  - Various workers (males, females, skilled, unskilled, semi-skilled) are working on some job.
  - Question: How to organize them effectively to improve the performance ?
  - Sitting worker is similar to program.
    - Does not use any tools and resources.
  - Working worker is similar to process
    - Uses tools, interacts with other,
    - If he is working with others, cooperation and synchronization is required. Otherwise accidents might occur.
    - If he is working alone, no cooperation is needed.
  - To manage working workers, the manager should have some information about the working worker.
    - What kind of tools he is using.
    - The nature of the job, and status of the job.
    - ….
  - In computer system
    - Operating system is similar to manager
    - The sitting workers are similar to the programs reside on the disk.
    - The working workers are processes
    - CPU is an expensive tool which should not be kept idle. It is OK if some workers wait for expensive tool.

Process Concept

- Early systems
  - One program at a time was executed and a single program has a complete control.
- Modern OSs allow multiple programs to be loaded in to memory and to be executed concurrently.
- This requires firm control over execution of programs.
- The notion of process emerged **to control the execution of programs**.
- A process
  - Unit of work
  - Program in execution
- OS consists of a collection of processes
  - OS processes executes system code.
  - User processes executes user code.
- By switching CPU between processes, the OS can make the computer more productive.

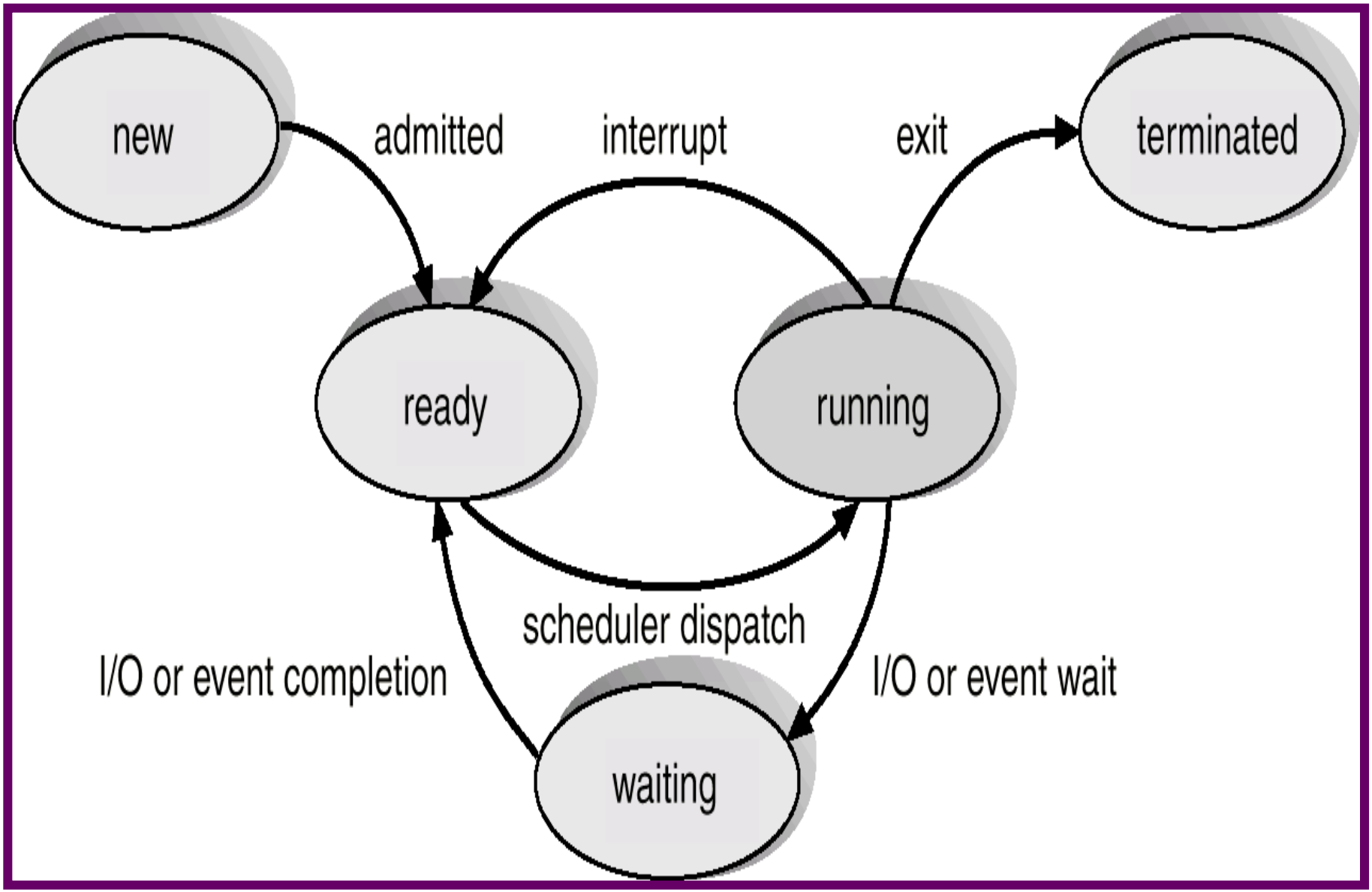
Process Concept…

- A program is simply a text
- Process (task or job) includes the current activity.– a program in execution; process execution must progress in sequential fashion.
- The components of a process are
  - The program to be executed
  - The data on which the program will execute
  - The resources required by the program– such as memory and file (s)
  - The status of execution
    - ✔ program counter
    - ✔ Stack
- A program is a passive entity, and a process is an active entity with the value of the PC. It is an execution sequence.
- Multiple processes can be associated with the same program (editor). They are separate execution sequences.
- For CPU, all processes are similar
  - Batch Jobs and user programs/tasks
  - Word file
  - Internet browser
  - System call
  - Scheduler
  - ….

Process State

- As a process executes, it changes *state*
- *Each process can be in one of*
  - **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.
- The names may differ between OSs.

Diagram of Process State



State change

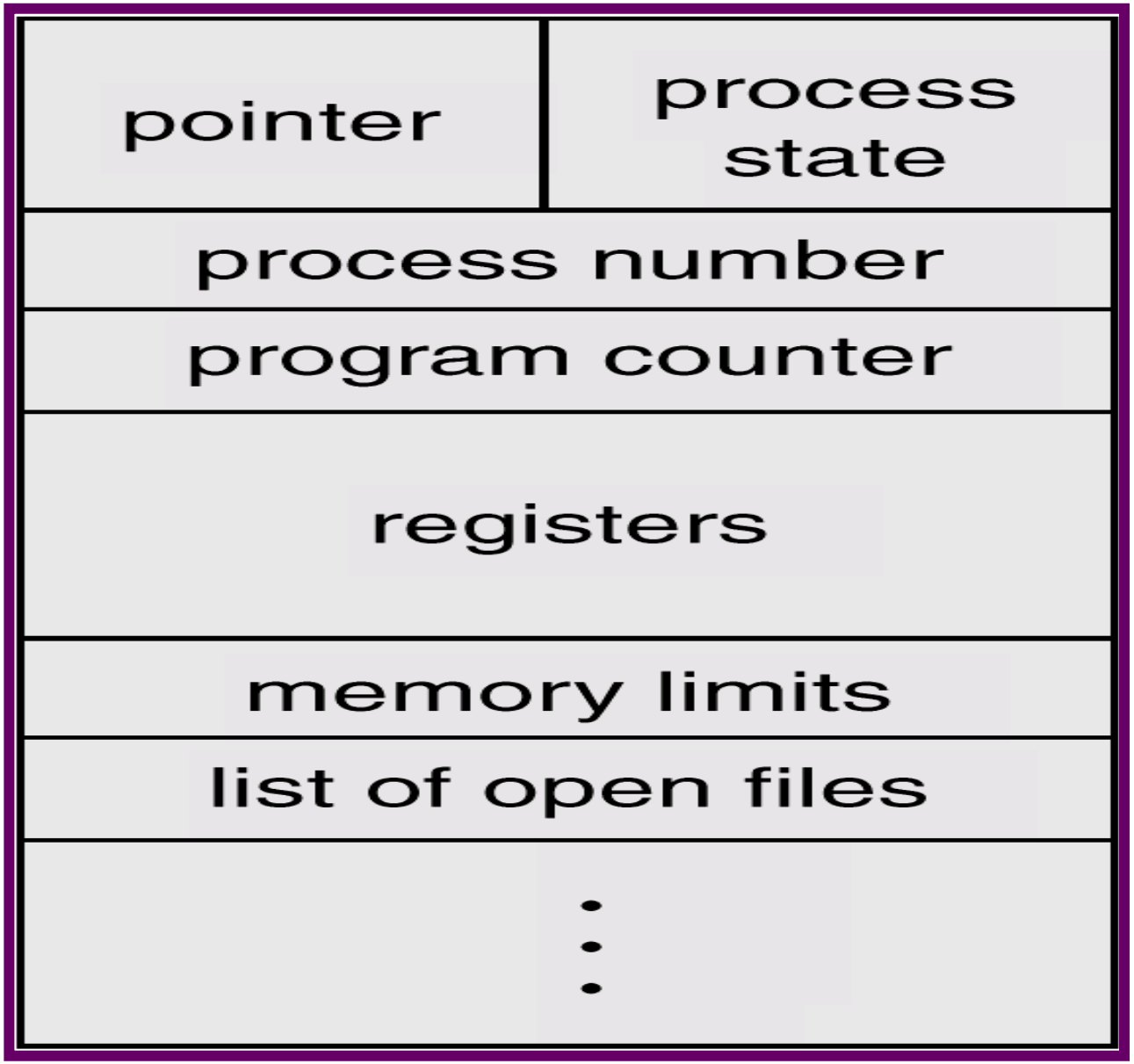
- **Null → New :** a new process is created to execute the program
  - New batch job, log on
  - Created by OS to provide the service
- **New → ready**: OS will move a process from prepared to ready state when it is prepared to take additional process.
- **Ready → Running**: when it is a time to select a new process to run, the OS selects one of the process in the ready state.
- **Running → terminated:** The currently running process is terminated by the OS if the process indicates that it has completed, or if it aborts.
- **Running → Ready**: The process has reached the maximum allowable time or interrupt.
- **Running → Waiting:** A process is put in the waiting state, if it requests something for which it must wait.
  - Example: System call request.
- **Waiting → Ready**: A process in the waiting state is moved to the ready state, when the event for which it has been waiting occurs.
- **Ready → Terminated:** If a parent terminates, child process should be terminated
- **Waiting → Terminated:** If a parent terminates, child process should be terminated

Process Control Block (PCB)

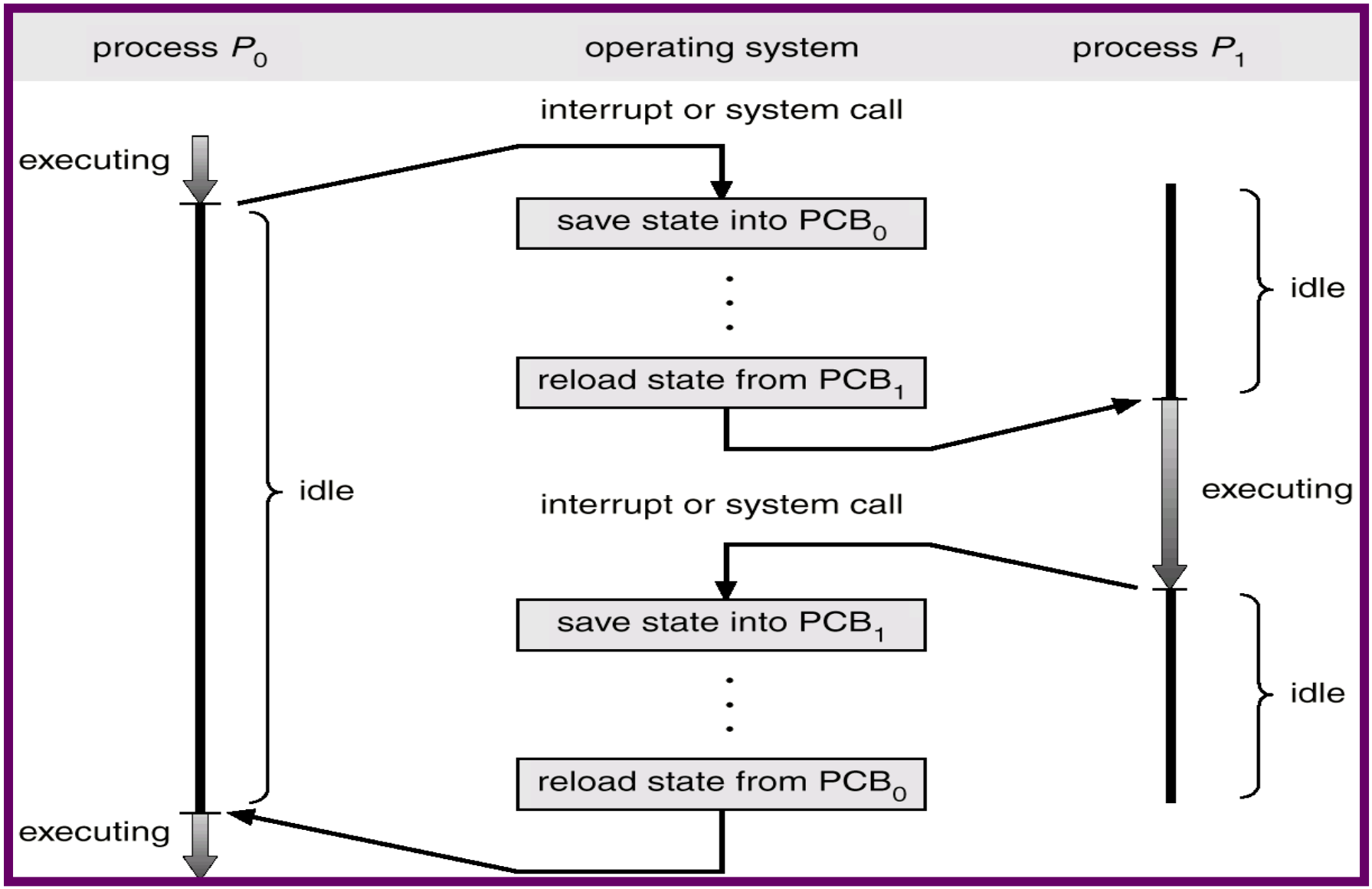
Information associated with each process.

- **Process state**: new, ready, running,…
- **PC**: address of the next instruction to execute
- **CPU registers**: includes data registers, stacks, condition-code information, etc
- **CPU scheduling information**: process priorities, pointers to scheduling queues, etc.
- **Memory-management information**: locations including value of base and limit registers, page tables and other virtual memory information.
- **Accounting information**: the amount of CPU and real time used, time limits, account numbers, job or process numbers etc.
- **I/O status information**: List of I/O devices allocated to this process, a list of open files, and so on

Process Control Block (PCB)



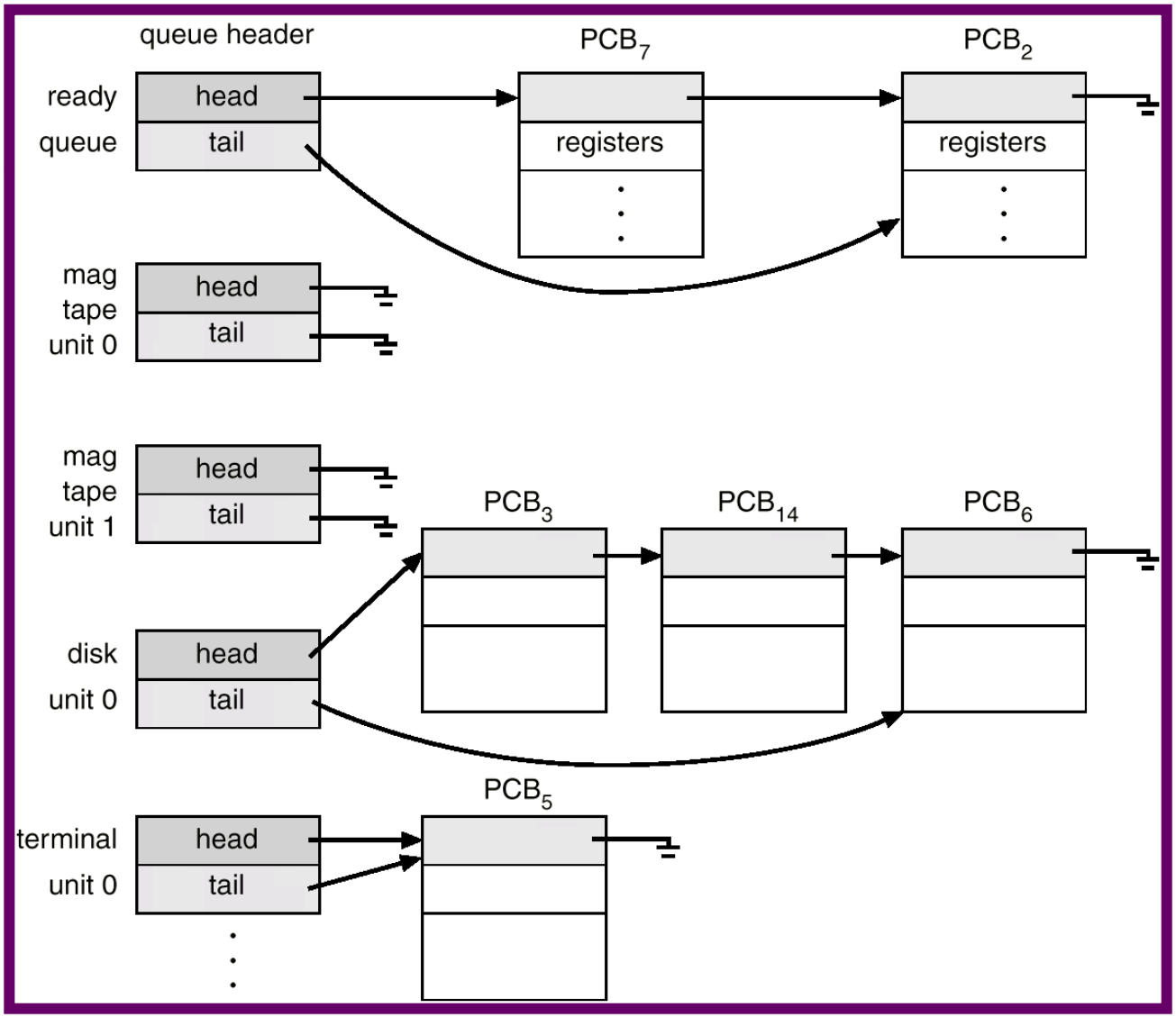
CPU Switch From Process to Process



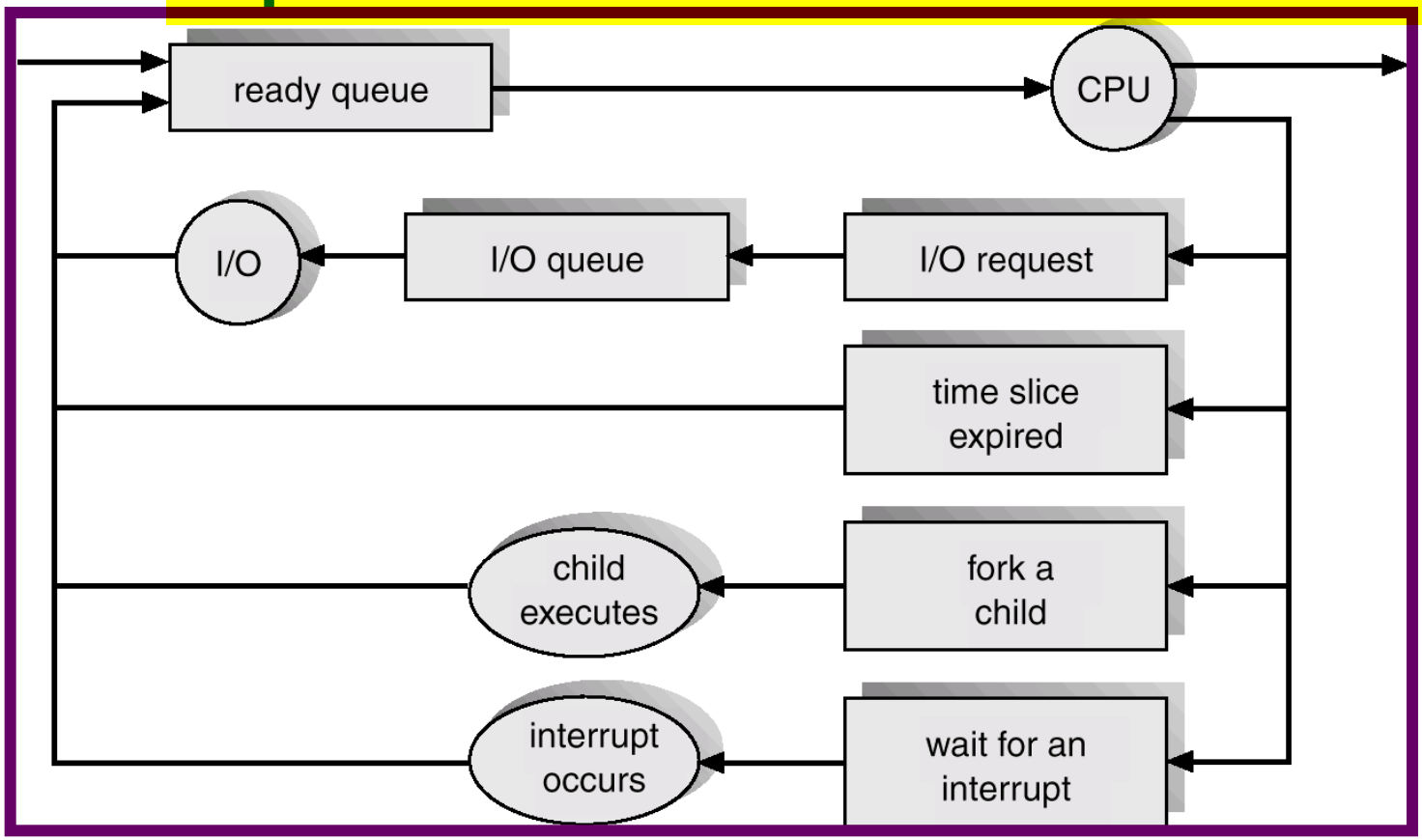
Process Scheduling Queues

- Scheduling is to decide which process to execute and when
- The objective of multi-program
  - To have some process running at all times.
- **Timesharing**: Switch the CPU frequently that users can interact can interact with the program while it is running.
- If there are many processes, the rest have to wait until CPU is free.
- Scheduling is to decide which process to execute and when.
- 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.
    - ✔ Each device has its own queue.
- Process migrates between the various queues during its life time.

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



- A new process is initially put in the ready queue
- Once a process is allocated CPU, the following events may occur
  - A process could issue an I/O request
  - A process could create a new process
  - The process could be removed forcibly from CPU, as a result of an interrupt.
- When process terminates, it is removed from all queues. PCB and its other resources are de-allocated.

Schedulers

- A process migrates between the various scheduling queues throughout its lifetime.
- The OS must select a process from the different process queues in some fashion. The selection process is carried out by a scheduler.
- In a batch system the processes are spooled to mass-storage device.
- **Long-term scheduler (or job scheduler)** – selects which processes should be brought into the ready queue.
  - It may take long time
- **Short-term scheduler (or CPU scheduler)** – selects which process should be executed next and allocates CPU.
  - It is executed at least once every 100 msec.
  - If 10 msec is used for selection, then 9 % of CPU is used (or wasted)
- The long-term scheduler executes less frequently.
- The long-term scheduler controls degree of multiprogramming.
- **Multiprogramming:** the number of processes active in the system.
- If MPL is stable: average rate of process creation= average departure rate of processes.

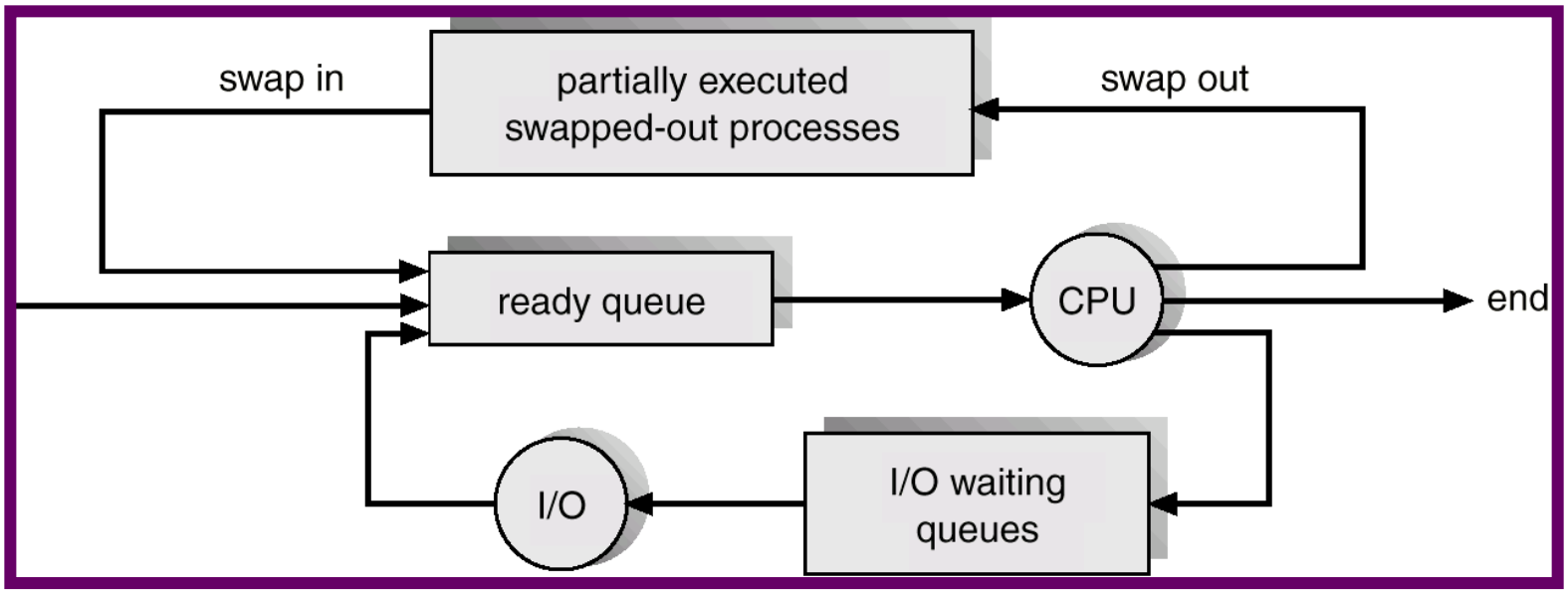
Schedulers…

- The long-term scheduler should make a careful selection.
- Most processes are either I/O bound or CPU bound.
  - **I/O bound process** spends more time doing I/O than it spends doing computation.
  - **CPU bound process** spends most of the time doing computation.
- The LT scheduler should select a good mix of I/O-bound and CPU-bound processes.
- Example:
  - If all the processes are I/O bound, the ready queue will be empty
  - If all the processes are CPU bound, the I/O queue will be empty, the devices will go unutilized and the system will be imbalanced.
- Best performance: best combination of CPU-bound and I/O-bound process.

Schedulers…

- Some OSs introduced a medium-term scheduler using swapping.
  - Key idea: it can be advantageous, to remove the processes from the memory and reduce the multiprogramming.
- **Swapping:** removal of process from main memory to disk to improve the performance. At some later time, the process can be reintroduced into main memory and its execution can be continued when it left off.
- Swapping improves the process mix (I/O and CPU), when main memory is unavailable.

Addition of Medium Term Scheduling



Context Switch

- Context switch is a task of switching the CPU to another process by saving the state of old process and loading the saved state for the new process
- **Context** of old process is saved in PCB and loads the saved context of old process.
- Context-switch time is **overhead**; the system does no useful work while switching.
- **New structures threads were incorporated.**
- Time dependent on hardware support.
  - 1 to 1000 microseconds

Operations on Processes: process creation

- A system call is used to create process.
  - Assigns unique id
  - Space
  - PCB is initialized.
- The creating process is called parent process.
- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.
  - In UNIX **pagedaemon, swapper,** and **init** are children are root process. Users are children of **init** process.
- A process needs certain resources to accomplish its task.
  - CPU time, memory, files, I/O devices.
- When a process creates a new process,
  - Resource sharing possibilities.
    - ✔ Parent and children share all resources.
    - ✔ Children share a subset of parent's resources.
    - ✔ Parent and child share no resources.
  - Execution possibilities
    - ✔ 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.
  - The new process is a copy of the original process.
  - The exec system call is used after a fork by one of the two processes to replace the process memory space with a new program.
- DEC VMS
  - Creates a new process, loads a specified program into that process, and starts it running.
- WINDOWS NT supports both models:
  - Parent address space can be duplicated or
  - parent can specify the name of a program for the OS to load into the address space of the new process.

UNIX: fork() system call

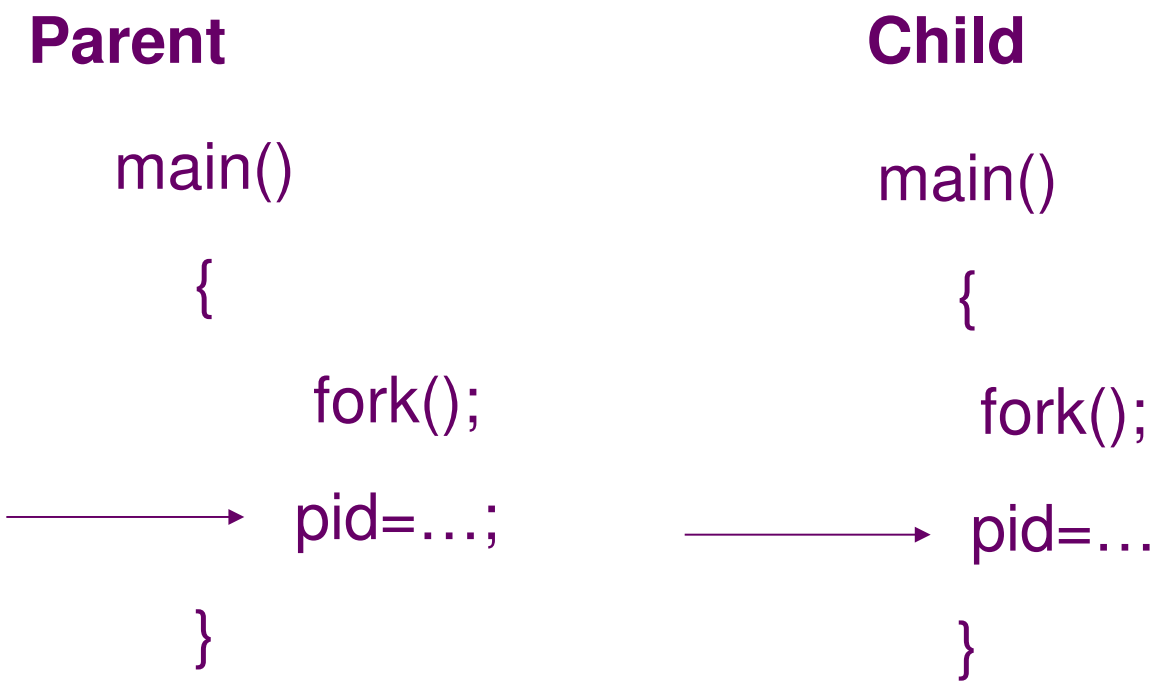
- fork() is used to create processes. It takes no arguments and returns a process ID.
- fork() creates a new process which becomes the child process of the caller.
- After a new process is created, both processes will execute the next instruction following the fork() system call.
- The checking the return value, we have to distinguish the parent from the child.
- fork()
  - If returns a negative value, the creation is unsuccessful.
  - Returns 0 to the newly created child process.
  - Returns positive value to the parent.
- Process ID is of type pit_t defined in sys/types.h
- getpid() can be used to retrieve the process ID.
- The new process consists of a copy of address space of the original process.

UNIX: fork() system call

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#define  MAX_COUNT 200
#define  BUF_SIZE 100
void main(void)
    {
                pid_t pid;
                int i;
      char buf[BUF_SIZE];
                fork();
       pid=getpid();
                for(i=1; i<=MAX_COUNT;i++)
        {
                    sprintf(buf,"This line is from pid %d, value=%d\n",pid,i);
          write(1,buf,strlen(buf));
        }
}
```

UNIX: fork() system call

- If the fork() is executed successfully, Unix will
  - Make two identical copies of address spaces; one for the parent and one for the child.
  - Both processes start their execution at the next statement after the fork().



UNIX: fork() system call

```
#include <stdio.h>
#include <sys/types.h>
#define  MAX_COUNT 200
void  ChildProcess(void);
void ParentProcess(void);
#define  BUF_SIZE 100
void main(void)
    {
                pid_t pid;
                pid=fork();
      if (pid==0)

    ChildProcess();
    else

    ParentProcess();
}
}
```

```
Void ChildProcess(void)
    {
                int i;
                for(i=1;i<=MAX_COUNT;i++)
      {
                   printf(buf,"This line is from child,
    value=%d\n",i);
                Printf(" *** Child Process is done ***\n");
      }
}

Void ParentProcess(void)
    {
                int i;
                for(i=1;i<=MAX_COUNT;i++)
      {
                 printf(buf,"This line is from parent,
    value=%d\n",i);
                printf(" *** Parent Process is done ***\
    n");
      }
}
```

UNIX: fork() system call

Parent

PID=3456

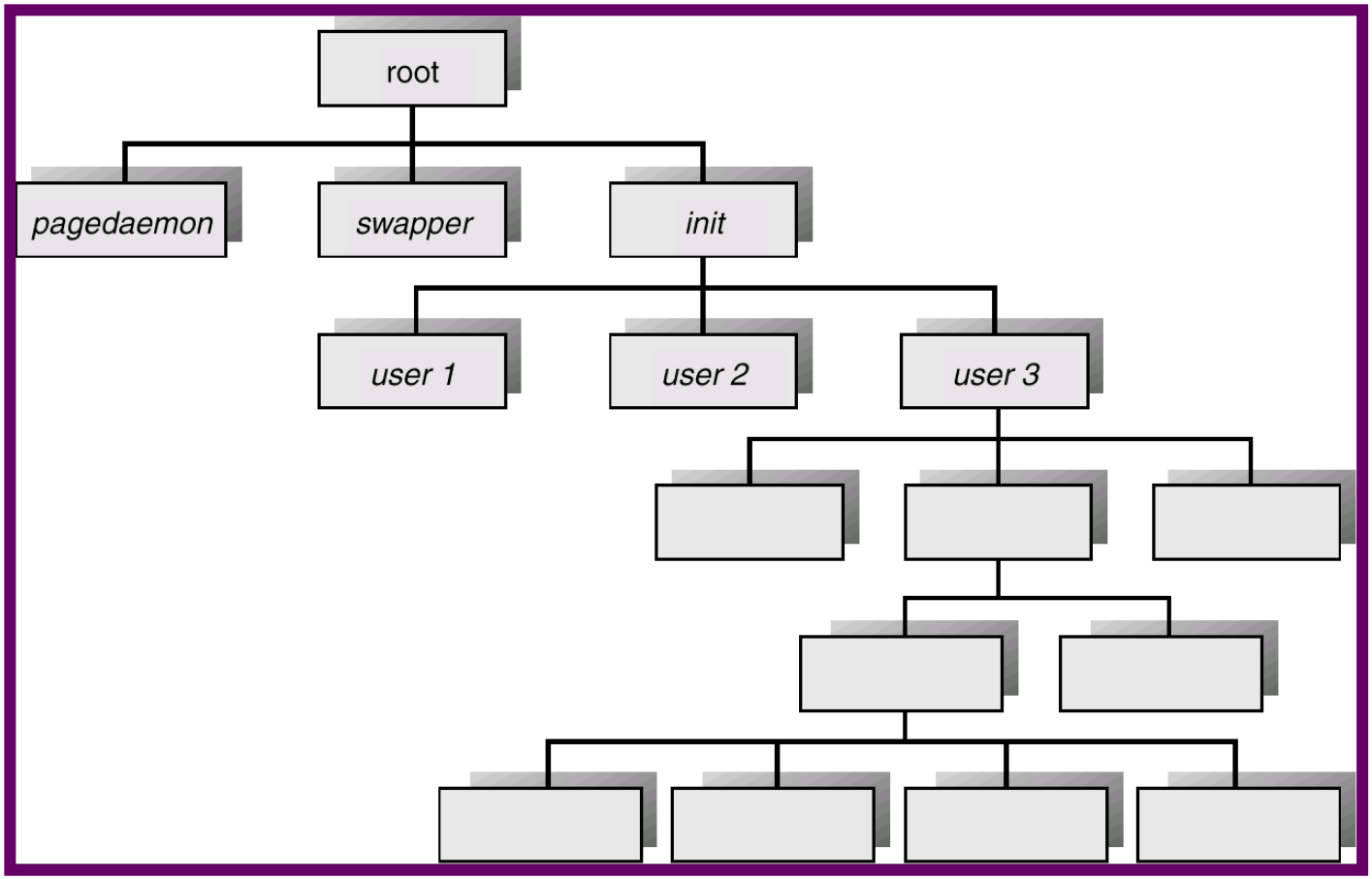
```
void main(void)
    {
    pid=fork();
      if (pid==0)
    ChildProcess();
    else
    ParentProcess();
  }
}
Void ChildProcess(void)
    {
    }

Void ParentProcess(void)
    {

}
```

Child

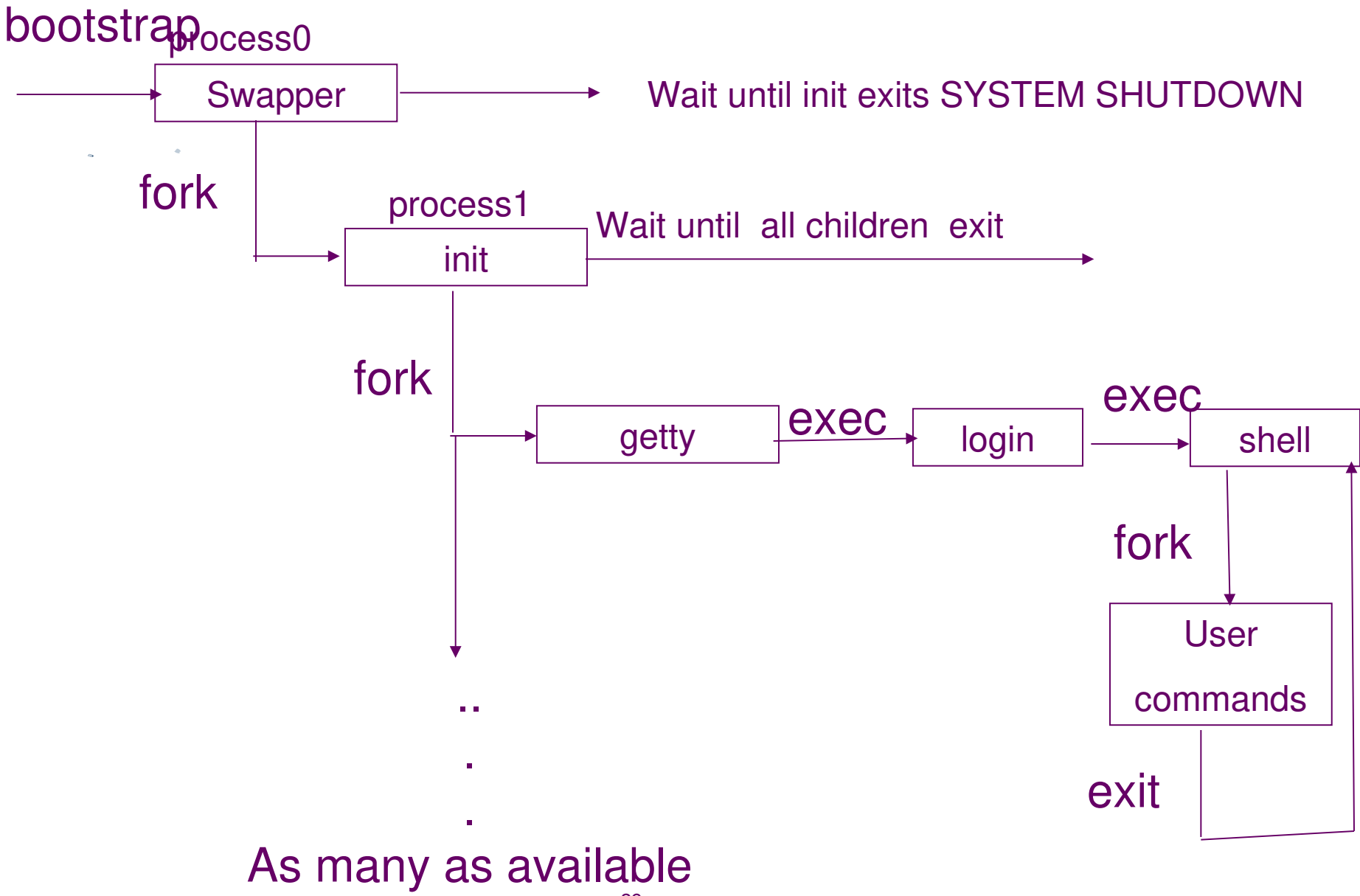
PID=0

```
void main(void)
    {
    pid=fork();
      if (pid==0)
    ChildProcess();
    else
    ParentProcess();
  }
}
Void ChildProcess(void)
    {
    }

Void ParentProcess(void)
    {

}
```

Processes Tree on a UNIX System



UNIX system initialization



Process Termination

- Process executes last statement and asks the operating system to decide it (**exit**).
  - Output data from child to parent (via **wait**).
  - Process' resources are deallocated by operating system.
- Parent may terminate the execution of children processes (**abort**).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.
    - ✔ Operating system does not allow child to continue if its parent terminates.
    - ✔ Cascading termination.

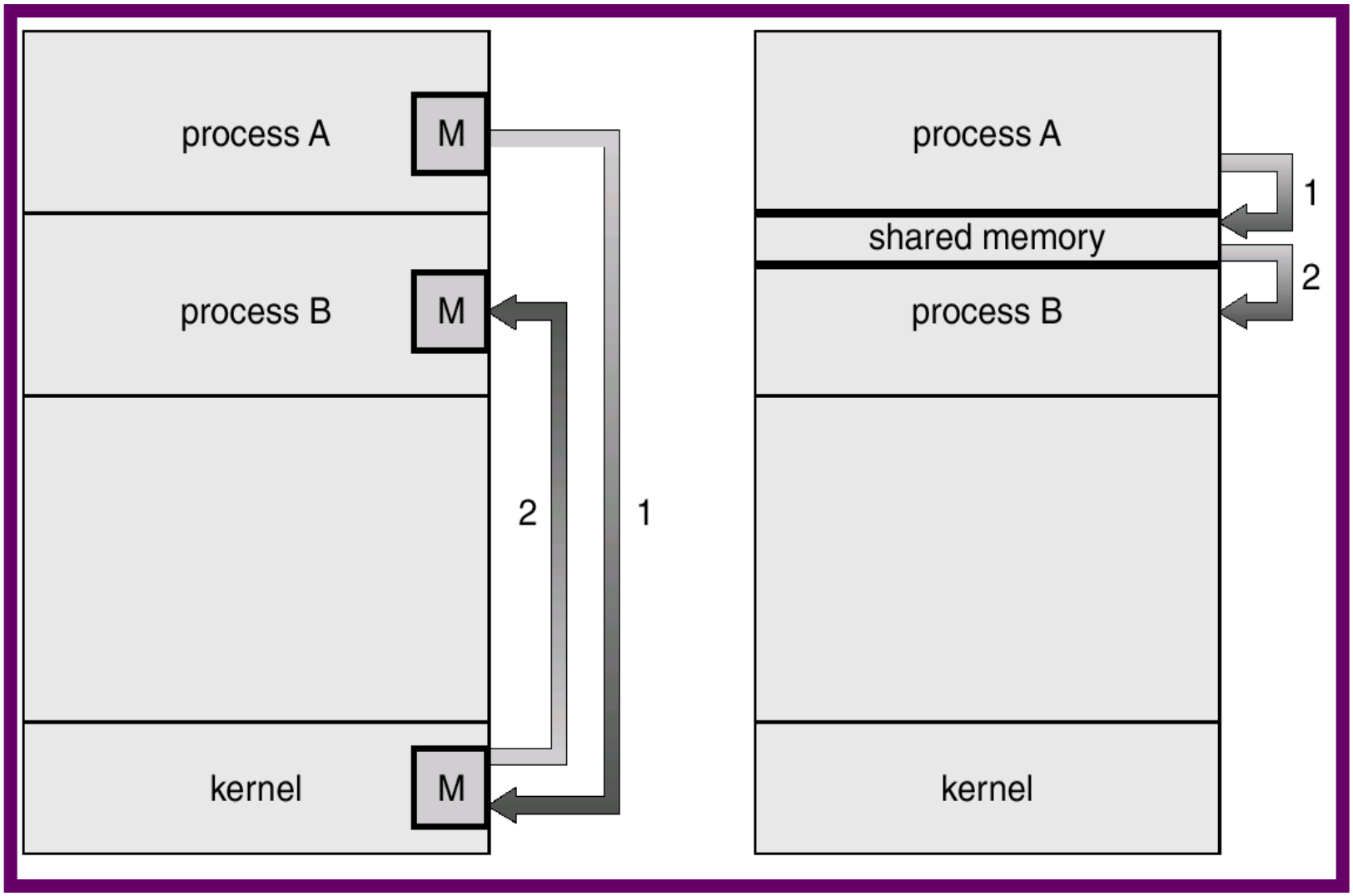
Cooperating Processes

- The processes can be independent or 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
    - ✔ Break into several subtasks and run in parallel
  - Modularity
    - ✔ Constructing the system in modular fashion.
  - Convenience
    - ✔ User will have many tasks to work in parallel
      - Editing, compiling, printing

Inter-process Communication (IPC)

- IPC facility provides a mechanism to allow processes to communicate and synchronize their actions.
- Processes can communicate through **shared memory or message passing.**
  - Both schemes may exist in OS.
- The Shared-memory method requires communication processes to share some variables.
- The responsibility for providing communication rests with the programmer.
  - The OS only provides shared memory.
- Example: producer-consumer problem.

Communication Models



Msg Passing

Shared Memory

Producer-Consumer Problem: Shared memory

- 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.
    - ✔ Producer can produce any number of items.
    - ✔ Consumer may have to wait
  - *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 next free position.
out points to first full position.

Bounded-Buffer – Producer Process

```
item nextProduced;

while (1) {
        /* Produce an item in nextProduced */
        while (((in + 1) % BUFFER_SIZE) ==
out)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded-Buffer – Consumer Process

```
item nextConsumed;

while (1) {
while (in == out)
; /* do nothing */
nextConsumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
/* Consume the item in nextConsumed */
}
```

Inter-process Communication (IPC): Message Passing System

- **Message system** – processes communicate with each other without resorting to shared variables.
- If P and Q want to communicate, a communication link exists between them.
- OS provides this facility.
- IPC 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
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)
    - ✔ We are concerned with logical link.

Fixed and variable message size

- Fixed size message
  - Good for OS designer
  - Complex for programmer
- Variable size messages
  - Complex for the designer
  - Good for programmer

Methods to implement a link

- Direct or Indirect communication
- Synchronous or asynchronous communication
- 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.
- This exhibits both symmetry and asymmetry in addressing
  - Symmetry: Both the sender and the receiver processes must name the other to communicate.
  - Asymmetry: Only sender names the recipient, the recipient is not required to name the sender.
    - ✔ The send and receive primitives are as follows.
      - Send (P, message)– send a message to process P.
      - Receive(id, message)– receive a message from any process.
- **Disadvantages: Changing a name of the process creates problems.**

Indirect Communication

- The messages are sent and received from mailboxes (also referred to as ports).
- A mailbox is an object
  - Process can place messages
  - Process can remove messages.
- Two processes can communicate only if they have a shared mailbox.
- Operations
  - create a new mailbox
  - 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 a 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.
- Properties of a link:
  - A link is established if they have a shared mailbox
  - A link may be associated with more than two boxes.
  - Between a pair of processes they may be number of links
  - A link may be either unidirectional or bi-directional.
- OS provides a facility
  - To create a mailbox
  - Send and receive messages through mailbox
  - To destroy a mail box.
- The process that creates mailbox is a owner of that mailbox
- The ownership and send and receive privileges can be passed to other processes through system calls.

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.
- Example:
- Producer process:

  repeat

  ….

  Produce an item in nextp

  …

  send(consumer,nextp);

  until false;
- Consumer process
  - **repeat** …. receive(producer, nextc);…. Consume the item in nextc … **until false;**

Synchronous or asynchronous

- Message passing may be either blocking or non-blocking.
- **Blocking** is considered **synchronous**
- **Non-blocking** is considered **asynchronous**
- **send** and **receive** primitives may be either blocking or non-blocking.
  - **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.
  - **Non-blocking send:** The sending process sends the message and resumes operation.
  - **Blocking receive:** The receiver blocks until a message is available.
  - **Non-blocking receive:** The receiver receives either a valid message or a null.

Automatic and explicit buffering

- A link has some capacity that determines the number of messages that can reside in it temporarily.
- Queue of messages is 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.
- In non-zero capacity cases a process does not know whether a message has arrived after the send operation.
- The sender must communicate explicitly with receiver to find out whether the later received the message.
- Example: Suppose P sends a message to Q and executes only after the message has arrived.
- Process P:
  - send (Q. message) : send message to process Q
  - receive(Q,message) : Receive message from process Q
- Process Q
  - Receive(P,message)
  - Send(P,"ack")

Exception conditions

- When a failure occurs error recovery (exception handling) must take place.
- Process termination
  - A sender or receiver process may terminate before a message is processed. (may be blocked forever)
  - A system will terminate the other process or notify it.
- Lost messages
  - Messages may be lost over a network
  - Timeouts; restarts.
- Scrambled messages
  - Message may be scrambled on the way due to noise
  - The OS will retransmit the message
  - Error-checking codes (parity check) are used.

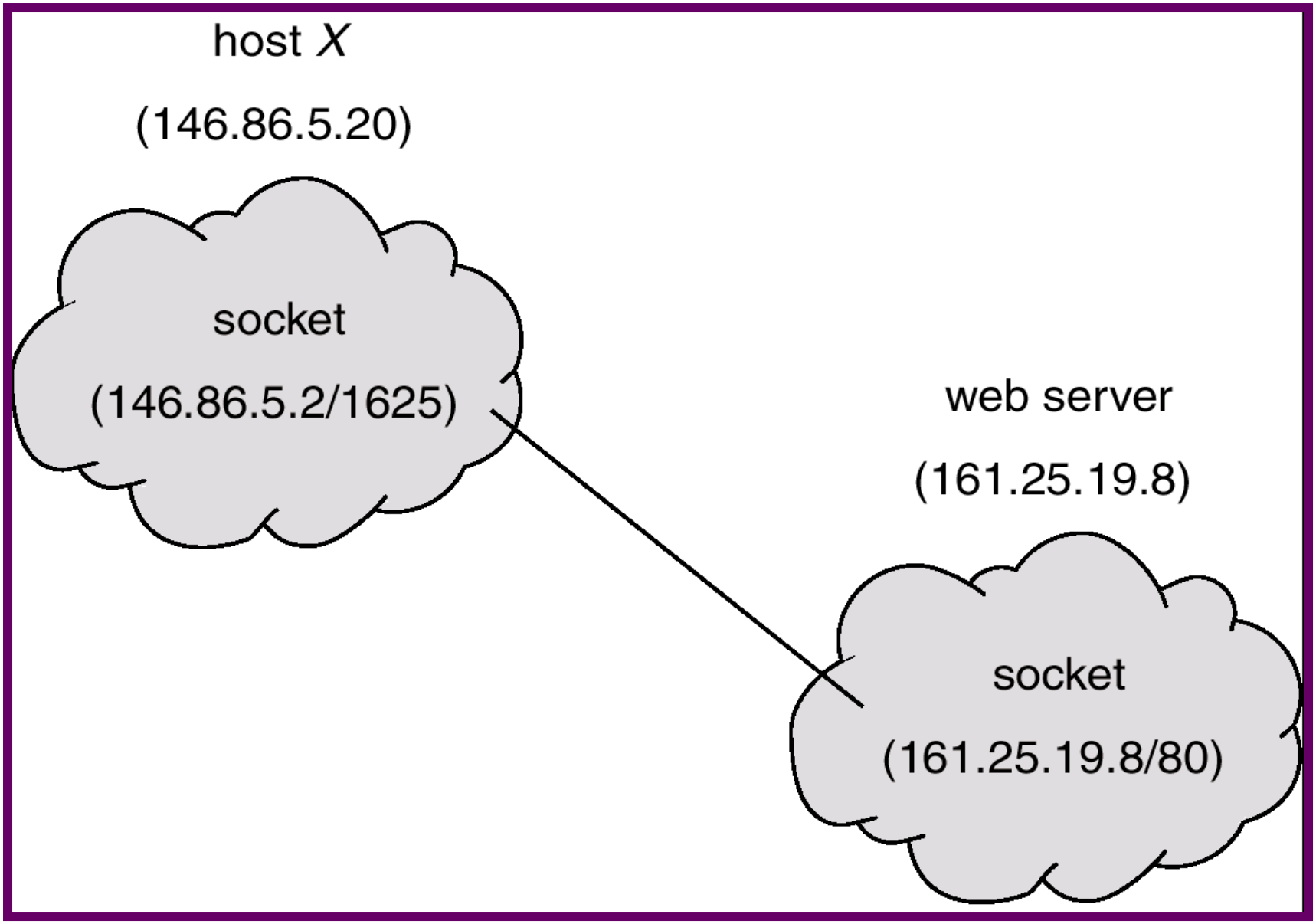
Client-Server Communication

- Sockets
- Remote Procedure Calls
- Pipes

Sockets

- A socket is defined as an *endpoint for communication*.
- A pair of processes communicating over a network employees a pair of sockets– one for each process.
- Socket: Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- **Servers implementing specific services listen to well-known ports**
  - **telnet server listens to port 80**
  - **ftp server listens to port 21**
  - **http server listens to port 80.**
- **The ports less than 1024 are used for standard services.**
- **The port for socket is an arbitrary number greater than1024.**
- Communication consists between a pair of sockets.

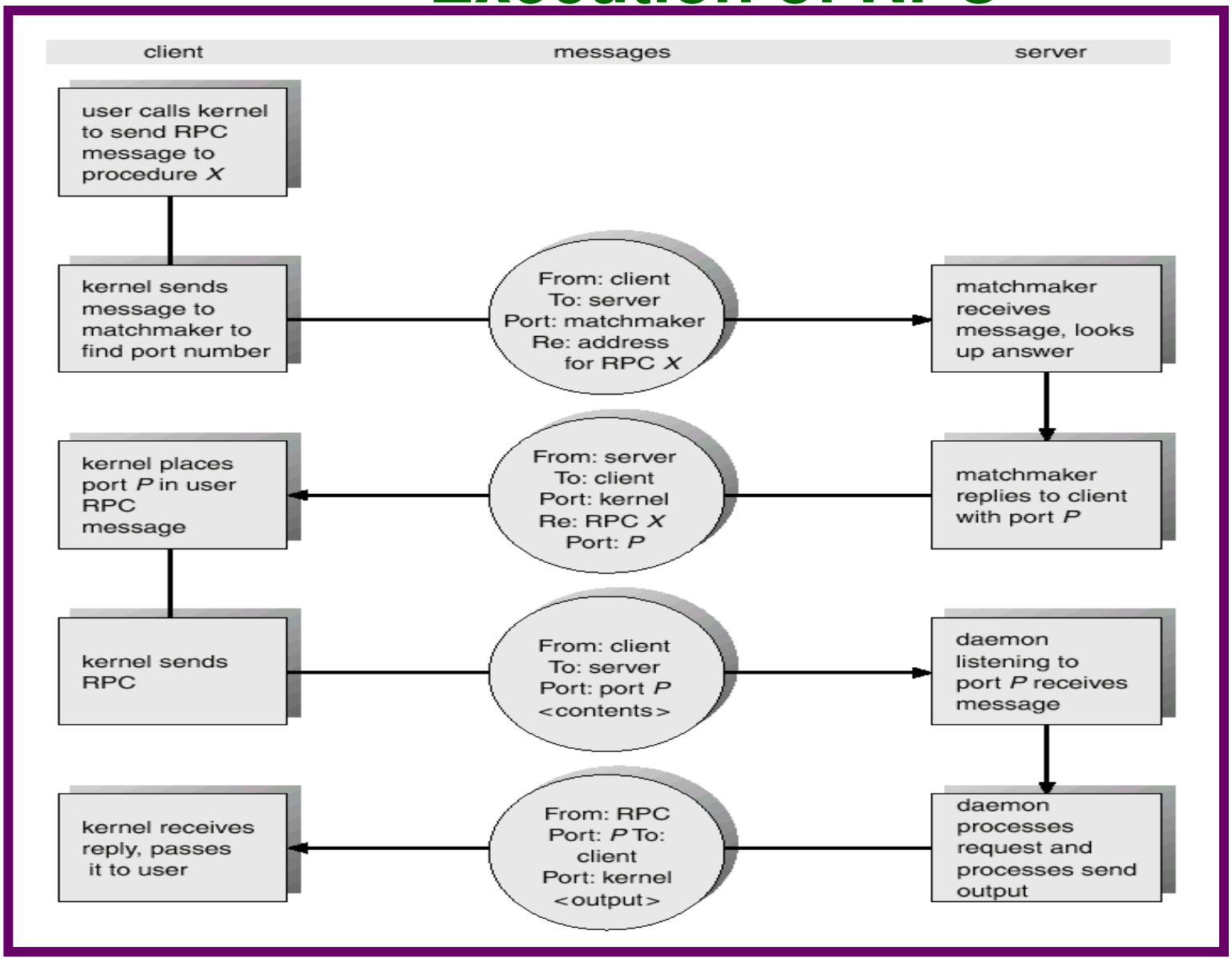
Socket Communication



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 *marshals* the parameters.
  - Marshalling: Parameters must be marshaled into a standard representation.
  - The server-side stub receives this message, unpacks the marshaled parameters, and performs the procedure on the server.
- There are several issues
  - Local procedure call can fail and RPCs can fail and re-executed
  - One way to address this problem
    - ✔ Exactly once semantics (Local procedure calls)
      - that processing of message will happen only **once.** It is difficult to implement in distributed system.
    - ✔ At most once semantics (Remote procedure calls)
      - when sending a message from a sender to a receiver there is no guarantee that a given message will be delivered.
      - Server will act and send ack and identifies repeated messages. Client may send messages more than once until it receives ack.
  - Binding issues: linking loading and execution
    - ✔ Fixed or rendezvous
- Applications: distributed file systems

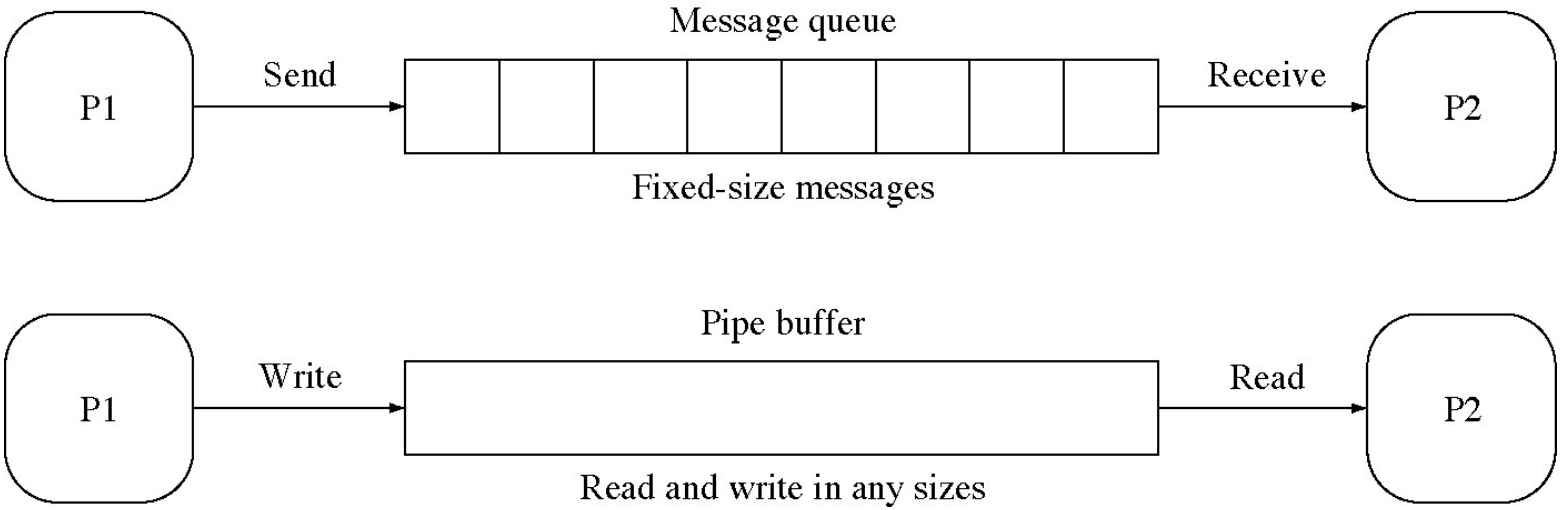
Execution of RPC



Pipes

- Pipe: another IPC mechanism
  - uses the familiar file interface
  - not a special interface (like messages)
- Connects an open file of one process to an open file of another process
  - Often used to connect the standard output of one process to the standard input of another process

Messages and pipes compared



More about Pipes

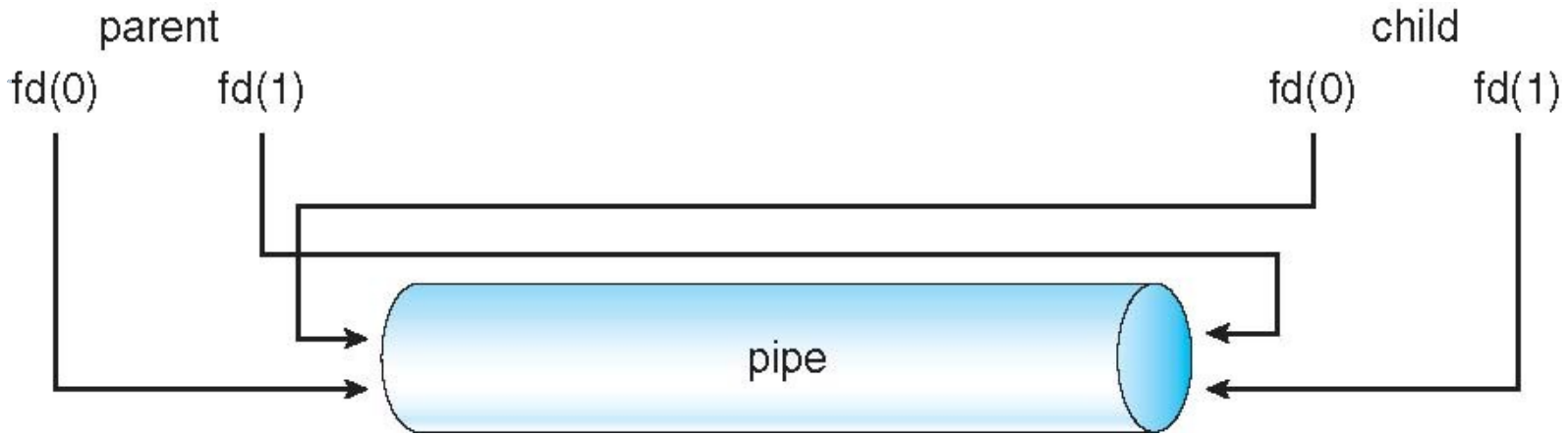
- Acts as a conduit allowing two processes to communicate

- **Issues**
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e. parent-child) between the communicating processes?
  - Can the pipes be used over a network?

Ordinary Pipes

- **Ordinary Pipes** allow communication in standard producer-consumer style
- Producer writes to one end (the *write-end* of the pipe)
- Consumer reads from the other end (the *read-end* of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes
- Ordinary pipe can not be accessed from outside the process that creates it.
- Pipe is a special type of file.

Ordinary Pipes



- Unix function to create pipes: pipe(int fd[])
- fd[0] is the read end. fd[1] is write end.

Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

Pipes in Practice

- ls | more