PROCESS CONCEPT

REFERENCES:

1. “OPERATING SYSTEM CONCEPTS” 9TH EDITION BY ABRAHAM SILBERSCHATZ, PETER BAER GALVIN AND GREG GAGNE
2. “OPERATING SYSTEMS: INTERNALS AND DESIGN PRINCIPLES”, 7TH EDITION BY WILLIAM STALLINGS
3. NPTEL lectures on “Introduction to Operating Systems” by Prof. Chester Rebeiro, IIT Madras https://nptel.ac.in/courses/106106144/

WHY WE NEED OPERATING SYSTEMS

- Programmers develop applications to perform some task over computer.
- It is not efficient to write applications for a particular hardware platform.
- OS act as secure and efficient interface between the applications and the hardware resources.
- OS provides a uniform access to resources when requested by applications

WHAT DOES OS MANAGES?

- It provides resources to multiple applications
- It helps multiple applications to use processor simultaneously by switching processor among applications.
- All applications make progress concurrently.
- It helps in efficiently sharing Processor and I/O devices among applications.

Define PROCESS?

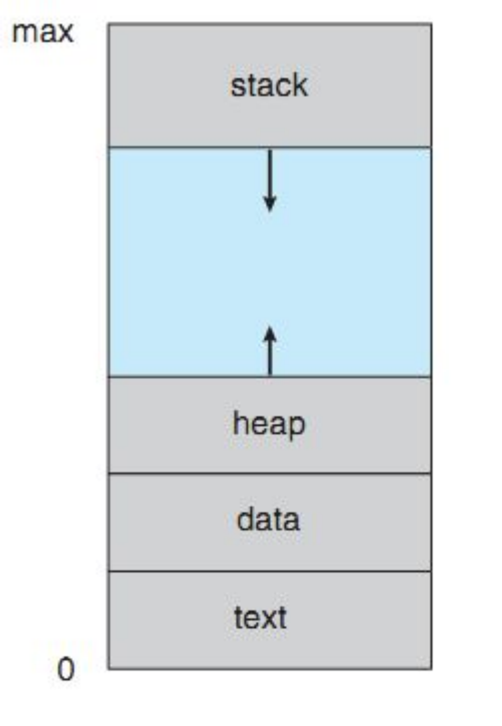
- It is an instance or a portion of a program which is getting executed in the processor at the moment
- Executing a process means executing instructions in a sequence.
- Each process has a current state, and a set of system resources acquired by it

PROCESS CONCEPT

- Types of systems
  - Batch system: Where computational tasks are completed by processors in the form of **Jobs.**
  - Time-shared systems: Processor is shared among applications.

- ***process*** can also be termed as ***job***
- A program is not a process.
- A program is a passive entity, for example a file with a list of instructions stored on disk.
- Whereas, a process is an active part of a process.
- A program counter specifies the next instruction to execute and a set of associated resources.
- When an executable file is loaded into RAM, the program becomes a process.

A Process in Memory

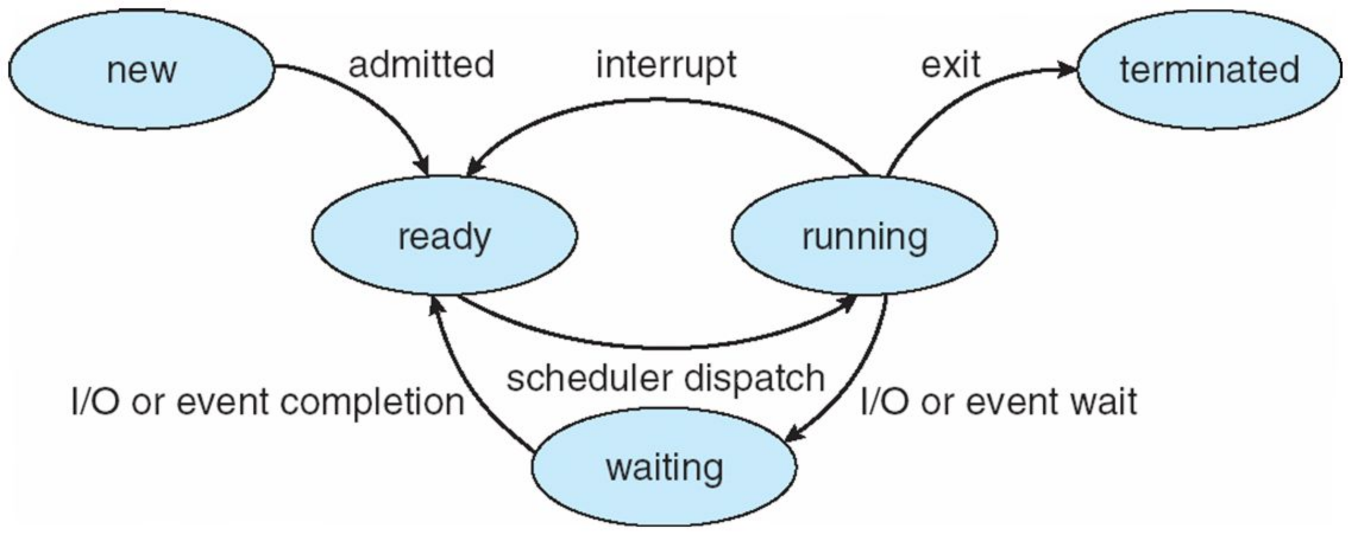


A process in memory
Figure 3.1 in Ref.1

- A Process consists of following sections in the memory:
  - The code or instructions, also called **text section**
  - **Program counter and registers** maintain current activities of a process
  - **Stack** stores temporary data of a process.
  - For example local variables, function arguments or return addresses
  - **Data section** stores global variables
  - Memory allocated dynamically during run time is stored in **Heap**

STATES OF A PROCESS

- During execution, a process is in one of the mentioned states**:**
  - **New**: When a process is being created.
  - **Waiting**: When the process is waiting for some I/O resource or an event to occur like getting input from keyboard, a message from another process, to access disk storage or an another process to finish a task, and so on.
  - **Ready**: The process is in queue having all the required resources and waiting to get a processor
  - **Running**: When CPU is executing process instructions
  - **Terminated**: When process finishes execution of all its instructions



WHAT IS A PROCESS CONTROL BLOCK (PCB)

- Each process has a **PCB** or a **Task Control Block.**
- Each process can be uniquely identified by a no. of parameters
- A process execution can be interrupted and later resumed
- It stores process specific information, like
  - State: State of a process can be ready, waiting, running, etc
  - Program counter: It stores location of next instruction to execute
  - CPU scheduling: it provides relative priorities of process and pointers to queue
  - CPU registers: It stores information/ data associated with process

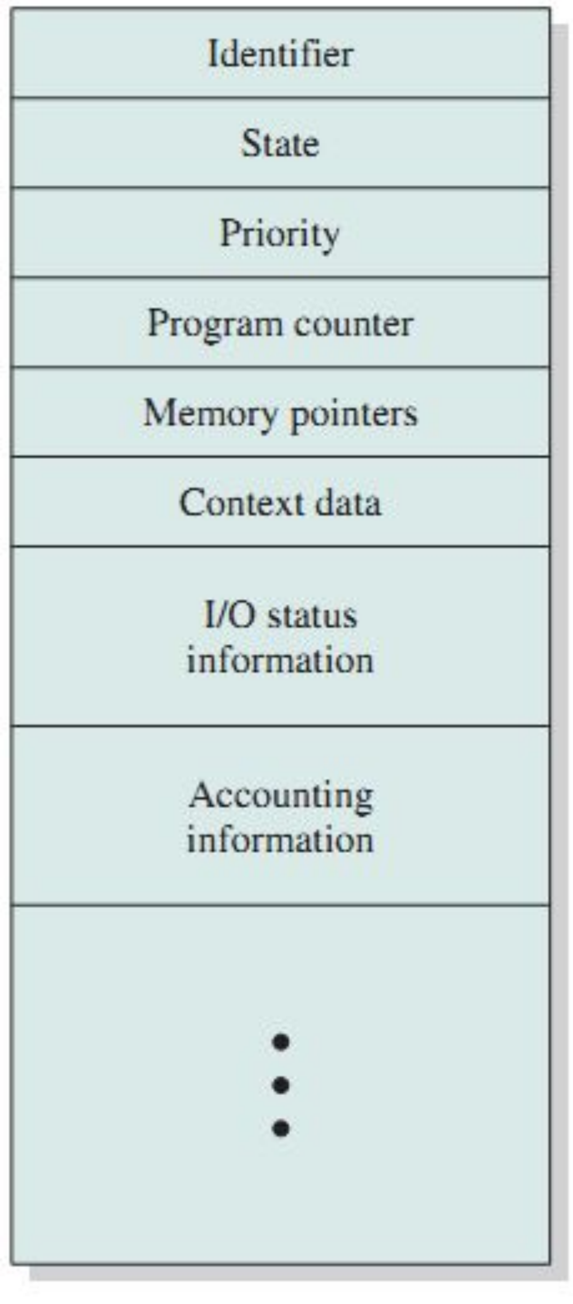


Fig. 3.1 from Ref. 2

PROCESS CONTROL BLOCK (CONTD.)

- Memory-management information: Size of memory and starting address allocated to the process
- Usage/Accounting information: CPU utilization, turn around time, etc
- I/O status: resources or devices allocated to a process
- Identifier: Each process has a unique id
- Memory pointers: It stores the address of the program code and data related to process
- Priority: Priority of a process compared to other

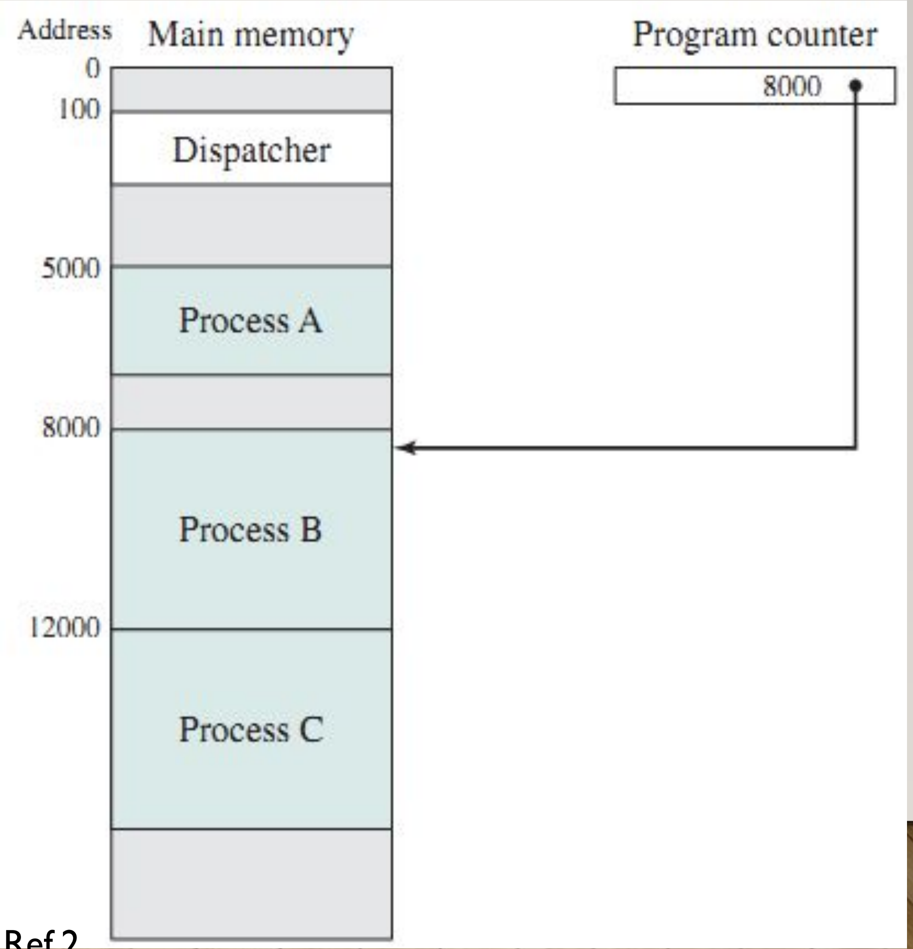
TRACE AND DISPATCHER

- TRACE monitors and notes execution sequence of instructions. It also keeps record of process interleaving while executing.

- DISPATCHER is a small program in operating system that helps in switching processor the from one process to another

(a) Trace of process A	(b) Trace of process B	(c) Trace of process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000 = Starting address of program of process A
8000 = Starting address of program of process B
12000 = Starting address of program of process C



- Fig. 3.2 and 3.3 from Ref.2

SWITCHING Between PROCESSES

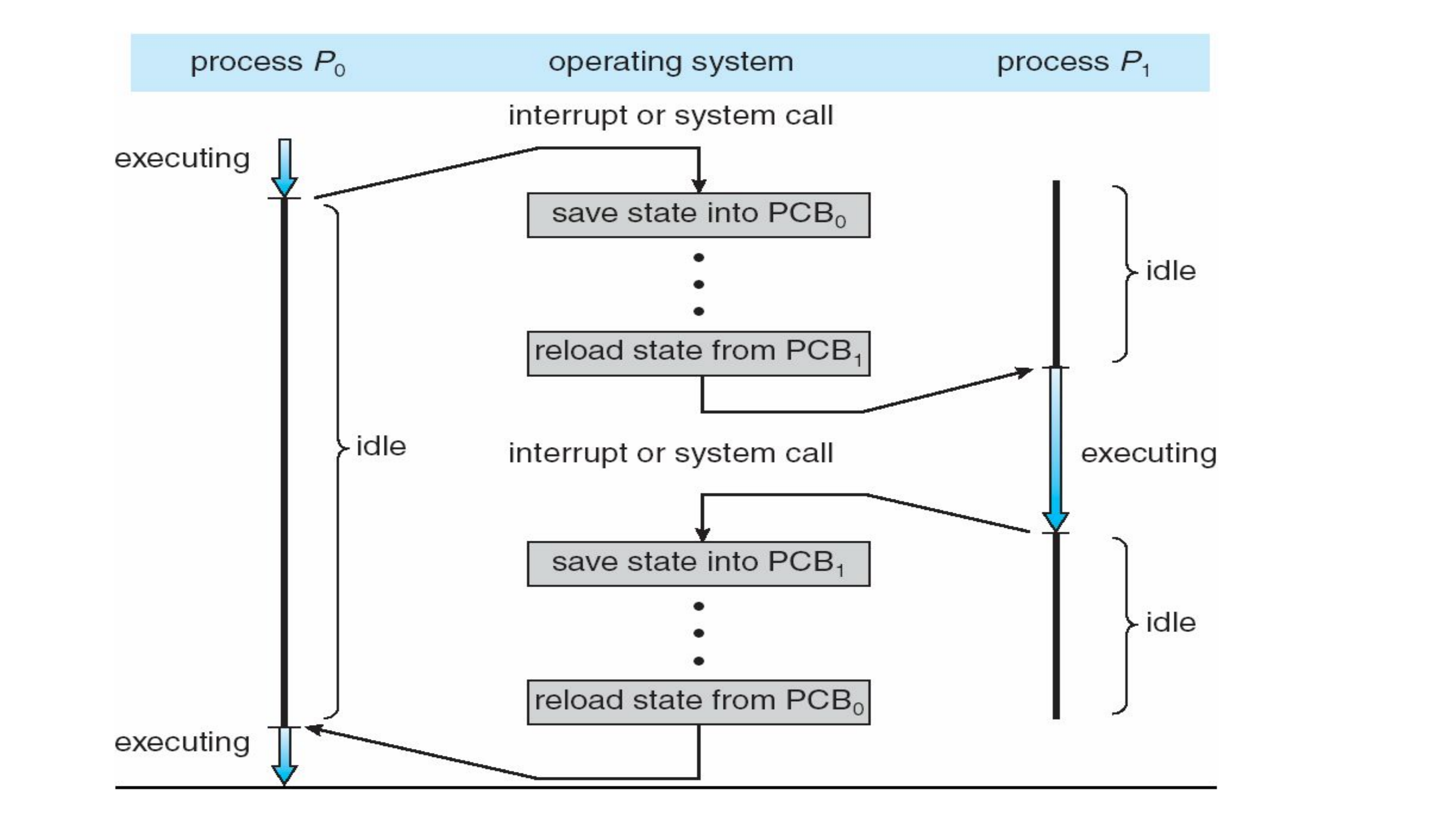


Fig. 3.4 from Ref.1

1	5000
2	5001
3	5002
4	5003
5	5004
6	5005
---------------------Time-out	
7	100
8	101
9	102
10	103
11	104
12	105
13	8000
14	8001
15	8002
16	8003
---------------------I/O request	
17	100
18	101
19	102
20	103
21	104
22	105
23	12000
24	12001
25	12002
26	12003
27	12004
28	12005
---------------------Time-out	
29	100
30	101
31	102
32	103
33	104
34	105
35	5006
36	5007
37	5008
38	5009
39	5010
40	5011
---------------------Time-out	
41	100
42	101
43	102
44	103
45	104
46	105
47	12006
48	12007
49	12008
50	12009
51	12010
52	12011
---------------------Time-out	

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Fig. 3.4 from Ref.2

THREADS

- A process can be divided into multiple light weight executable entities to parallelize the task.
- These are called Threads.
- Suppose a process consists of multiple program counters (PC)
- Code of a process can be executed from multiple locations or instructions

PROCESS SCHEDULING

- Multiple processes share same CPU in time sharing systems
- It is required to maximize CPU utilization
- A process should not hold processor while it is busy some other tasks like I/O activity.
- A quick processes switching is required onto CPU

PROCESS SCHEDULING

- **A Process scheduler** selects the next process for execution on CPU while multiple are available to run
- It maintains scheduling queues of processes
  - **I/O or Device queues:** consist of processes waiting to get access to an I/O device
  - **Ready queue:** It is a queue of processes which are waiting in main memory to execute in CPU
  - Processes switches from one queue to another

READY QUEUE AND VARIOUS I/O DEVICE QUEUES

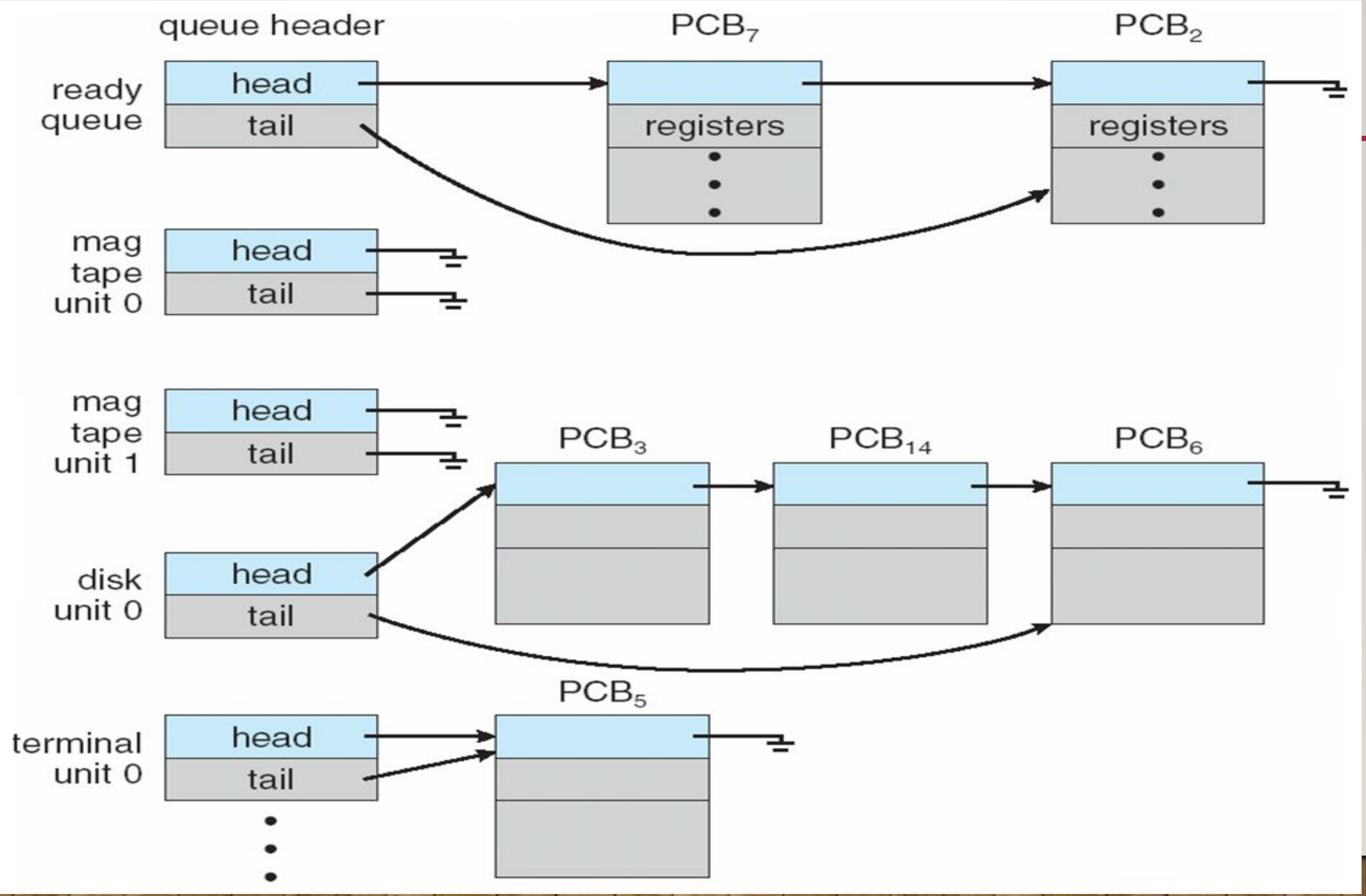


Fig. 3.5 from Ref.1

QUEUEING DIAGRAM

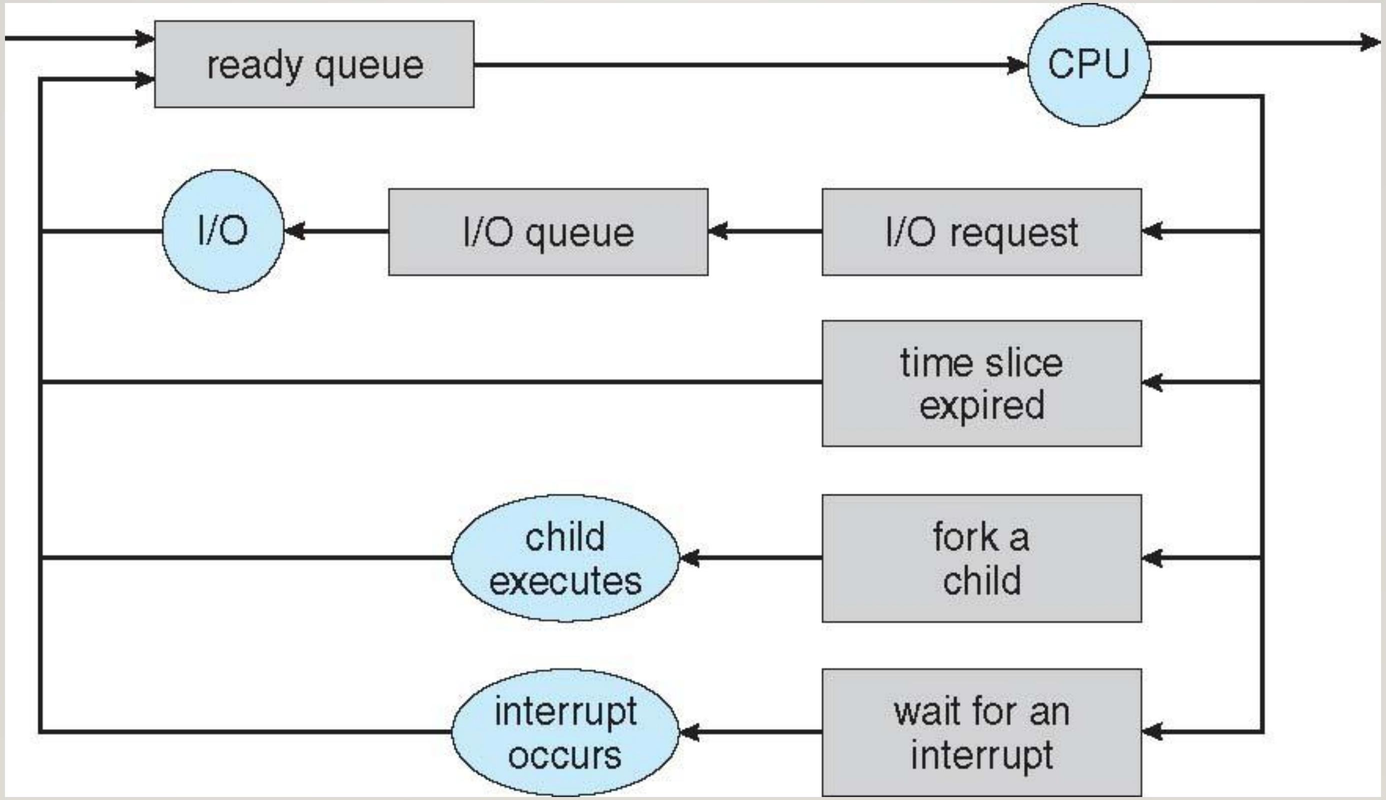


Fig. 3.6 from Ref.1

SCHEDULERS

- Processes are usually stored to a secondary storage device (like a disk) for later execution
- For execution OS brings them to main memory (RAM)

- Processes are classified as:
  - **I/O-bound process** – These processes majorly spends time performing I/O activities than doing computations using CPU
  - **CPU-bound process** – – These processes majorly perform computations using CPU

- The portion of the OS responsible for selecting processes to be brought to main memory/ ready queue is **Long-term scheduler** (or **job scheduler**).
  - It is called less-frequently (like in seconds, minutes)
  - It decides the **degree of multiprogramming**
  - It should select a proper combination of I/O bound and CPU bound processes

SCHEDULERS (CONTD.)

- The portion of the OS responsible for selecting processes which can be executed next in CPU is called **Short-term scheduler** (or **CPU scheduler**)
  - Few system have only this scheduler
  - It is called frequently (like in milliseconds)

- **Medium-term scheduler:** It can used to reduce the degree of multiple programming.
  - It performs **swapping of processes.** It swaps processes from memory to disk and vice-versa

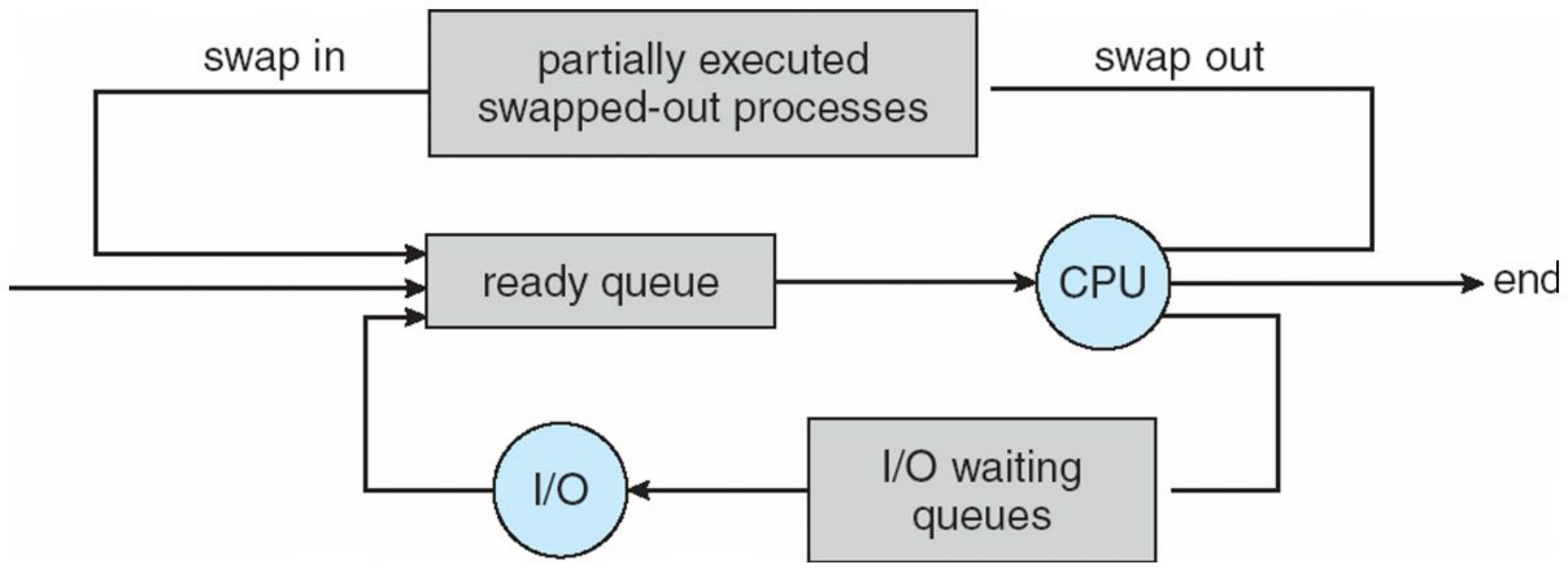


Fig. 3.7 from Ref.1

CONTEXT SWITCH

- Sometimes OS interrupts CPU from performing its current task to run a kernel routine
- Or CPU switches to another process when CPU usage of one process is either complete or interrupted
- **Context switching** means saving the current state of a running process so that it can be restored.
- It helps in saving the state of the old process and loading the saved state for the new process
- **PCB** provides the **Context/state** of a process
- Context-switch time is an overhead; not a useful work for the system
- Complex OS and PCB increases the switching time
- Time also depends on hardware support

OPERATIONS ON PROCESSES

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

TWO-STATE DIAGRAM

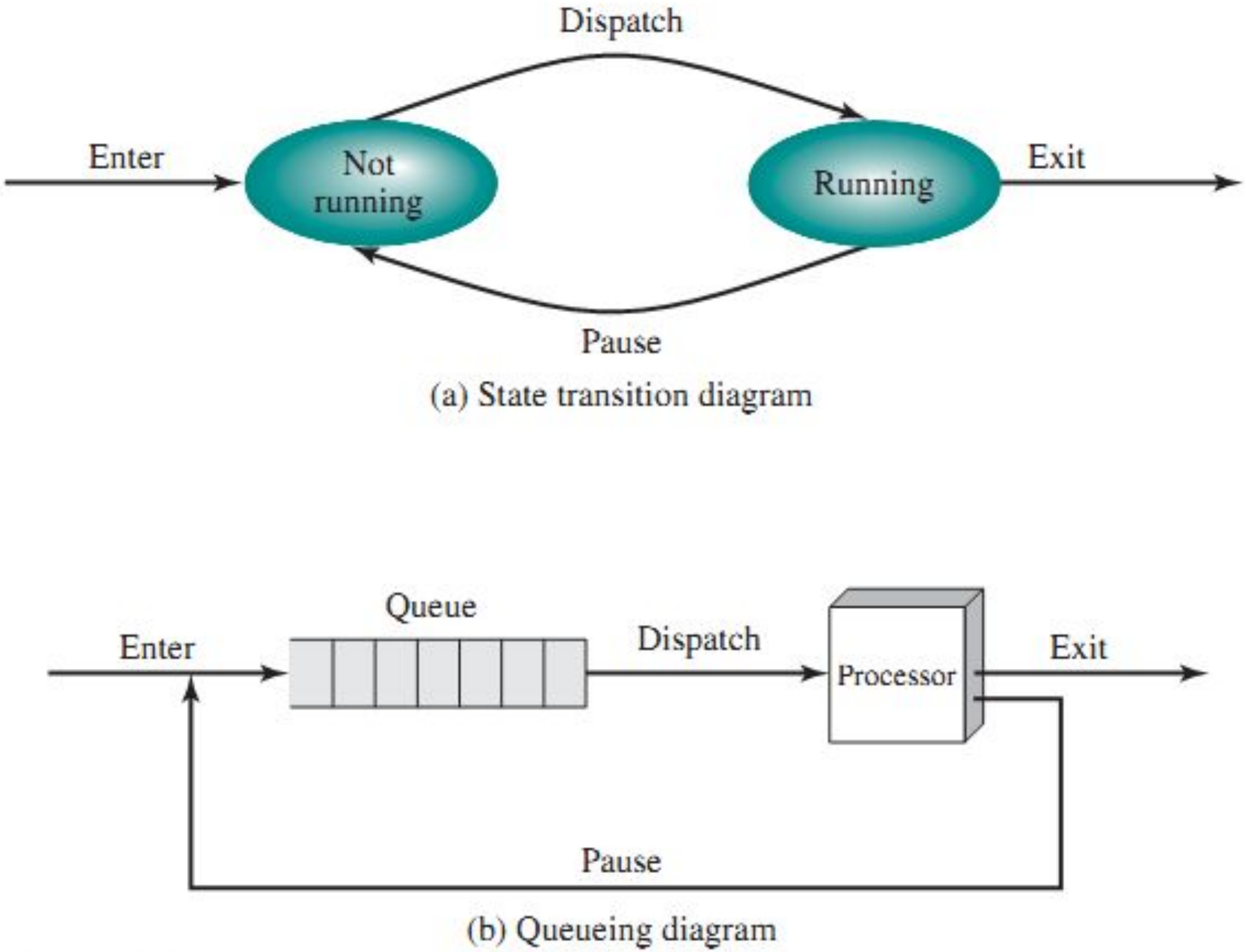


Figure 3.5 Two-State Process Model

Ref.2

REASONS FOR PROCESS CREATION

New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive logon	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

Or when a process requests the OS to create another process

Table 3.1 from Ref.2

PROCESS CREATION

- A process can create another process by requesting OS.
- These processes are called **Parent** process and **children** processes, respectively.
- This may result in creating a **tree** of processes
- Each process has a unique **process identifier** (**pid**)

Resource Sharing

- A Parent process has to distribute its resources among its children
- It prevents any process from creating too many child processes as it may overload the system
- A child process can acquire its resources either from its parent or directly from the operating system

PROCESS CREATION

Process Execution

- Parent and children processes may execute simultaneously
- Parent waits to terminate until all its children terminate

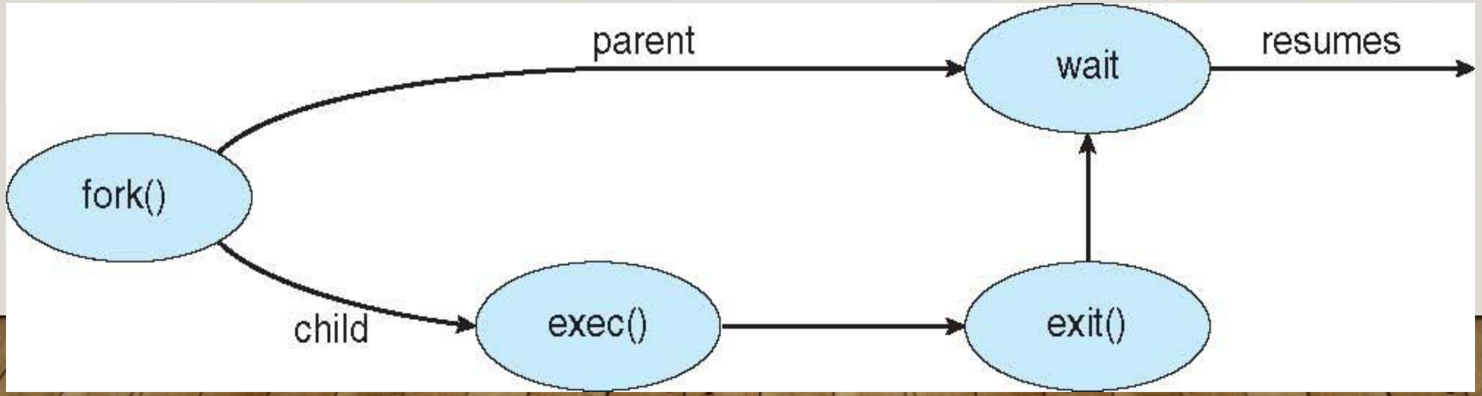
Address-space options for the new process

- A child process is a copy of its parent process.
- They both have same program and data

CREATING PROCESS IN UNIX

- **fork():** This system call creates new process
- The parent and the child both continue the execution starting from the instruction just after the fork()
- Fork returns 0 to the child process and pid of child to the parent

- **exec():** After a fork(), the parent or the child process uses the exec() system call to replace the memory space of process with a new program
- Parent can use wait() to leave the ready queue until its children terminate



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;
}
```

Figure 3.9 Creating a separate process using the UNIX fork() system call
Ref.1

PROCESS TERMINATION

Methods available to terminate a process:

- HALT instruction or an explicit OS service call can be used for termination
- Close or Logoff application
- Unix uses **exit()** system call
- Parent process receives a status value from child process using wait() system call
- Resources of the terminating process are deallocated like I/O buffers, files, physical and virtual memory

PROCESS TERMINATION

`abort()`: Parents can use it to terminate the execution of children processes

Reasons:

- When child exceeds allocated resources
- No utility of child and it assigned task
- Parent wants to terminate and the operating systems does not allow a child to continue if its parent terminates

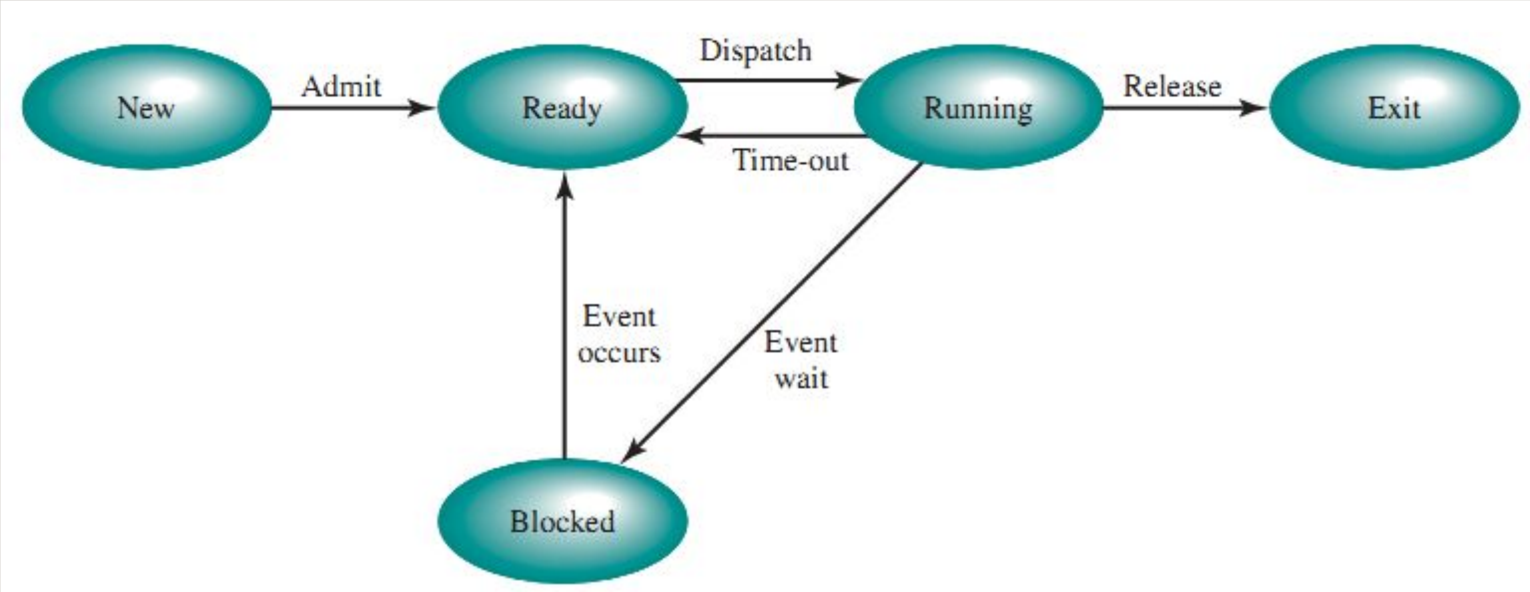
Cascading termination: Termination of a process leads to the termination of all its children and grandchildren

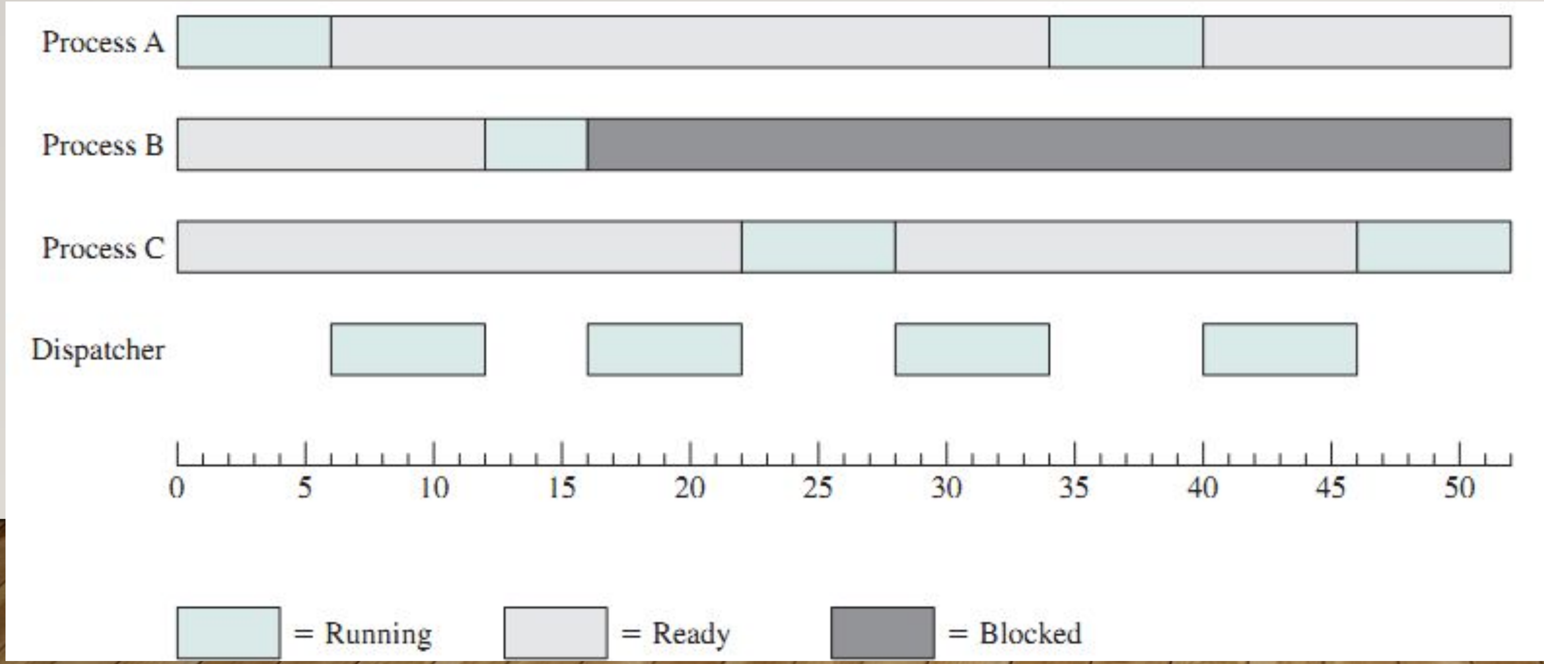
- **Exit()** returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- If no parent waiting (i.e. did not invoke `wait()`) then child process becomes **Zombie**
- If parent terminated without invoking `wait`, process becomes **Orphan**

FIVE-STATE DIAGRAM





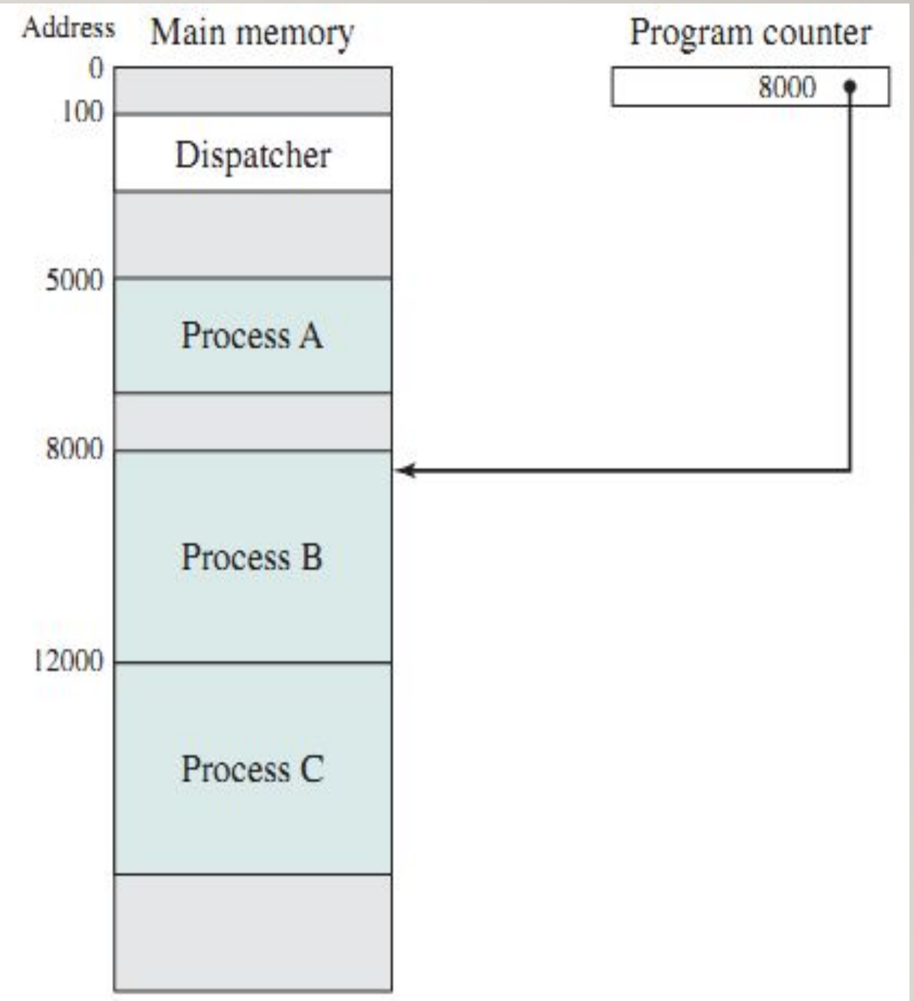


Figure 3.6 and 3.7 from Ref.2

SUSPENDED PROCESSES

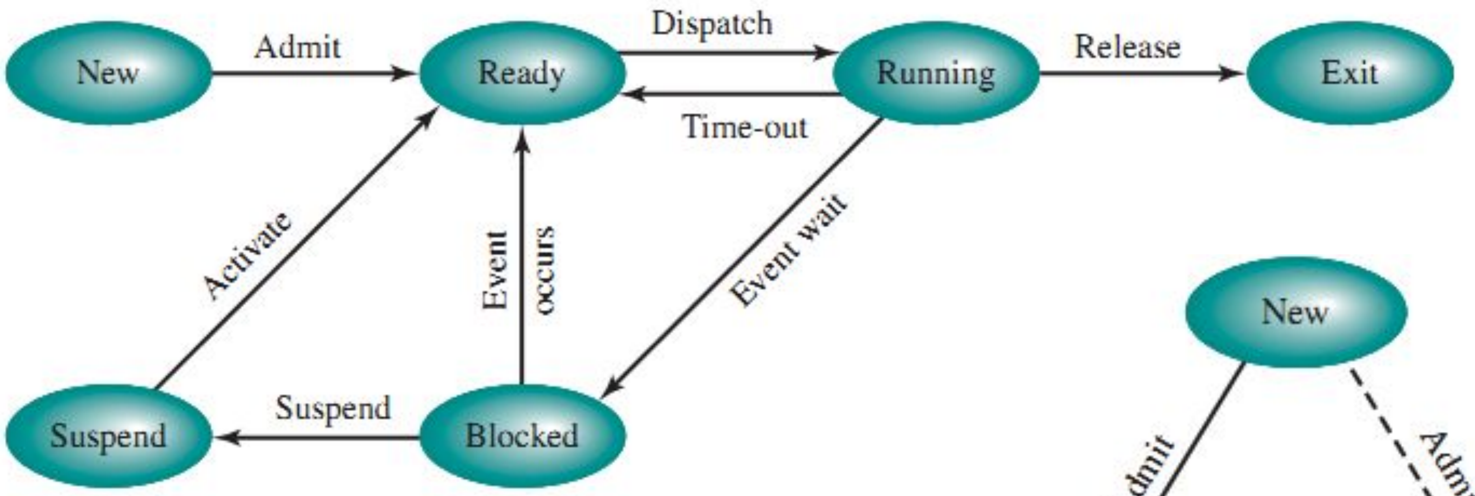
Swapping

Scenario: Ready queue is full. Processes in ready queue are blocked and waiting to complete an I/O activity. Processor is idle.

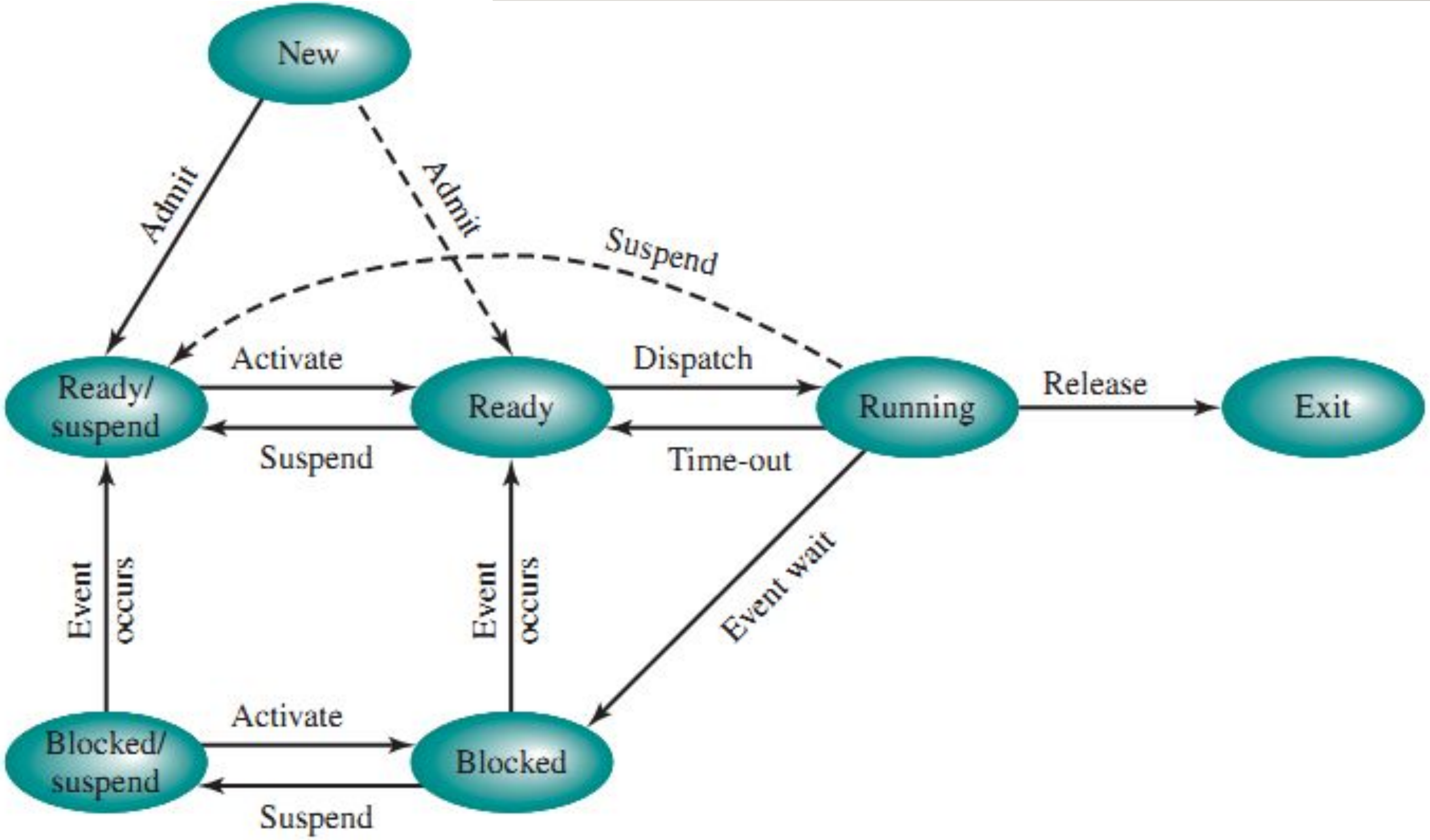
Solution: No processes in main memory are in the Ready state, the OS swaps one of the blocked processes out on to disk into a suspend queue

After swapping-out operation, OS has two choices:

- Bring a newly created process into main memory
- Or bring in a previously suspended process.



(a) With one suspend state



(b) With two suspend states

Figure 3.9 Process State Transition Diagram with Suspend States from Ref.2

- **Running to Ready/Suspend:** Time allocation expires OR a higher-priority process on the Blocked/Suspend queue gets unblocked

- **Blocked to Blocked/Suspend**: No ready processes available OR currently running process or a ready process about to dispatch requires more main memory

- **Blocked/Suspend to Blocked**: A process in the (Blocked/Suspend) queue is having higher priority over the processes in the (Ready/Suspend) queue and OS assumes unblocking event will occur soon.

- **Blocked/Suspend to Ready/Suspend**: The event for which it has been waiting occurs

- **Ready/Suspend to Ready**: there is no ready process in main memory OR a process in the Ready/Suspend state has higher priority than the processes in the Ready state.

- **Ready to Ready/Suspend**: Normally, not preferred. Needed to free up a sufficiently large block of main memory. Might also suspend a lower-priority ready process rather than a higher priority blocked process
- **New to Ready/Suspend**: Insufficient room in main memory for a new process

INTERPROCESS COMMUNICATION

- Types of Processes: ***Independent*** or ***Cooperating***
- ***Cooperating:*** It can affect or be affected by the other processes and share data with other process

---

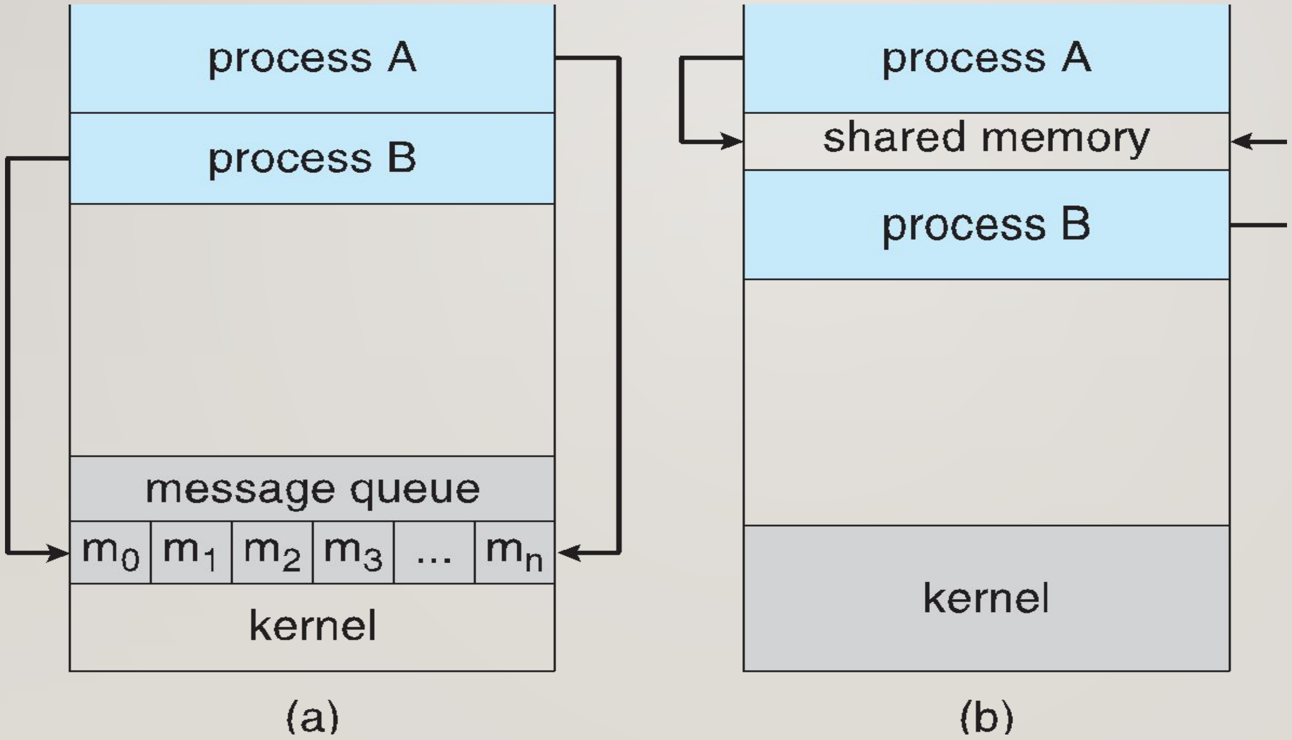
- Need for cooperating processes:
  - Modularity
  - Convenience
  - Computation speedup
  - Information sharing
- Cooperating processes need **Interprocess Communication** (**IPC**)

Why Cooperation?

- **Information sharing**: Several users may be interested in the same information
- **Computation speedup**
- **Modularity:** dividing the system functions into separate processes or threads
- **Convenience**

- IPC Mechanisms
  - **Shared memory**
  - **Message passing**
  - **Signals**

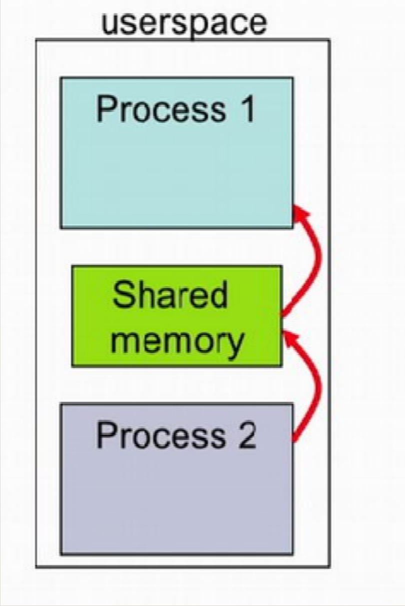
(a) Message passing (b) shared memory



Many systems implement both shared memory and message passing

SHARED MEMORY

- One process reserves an area in the RAM that is accessible by another process.
- Advantage: Fast access- similar to regular reading and writing in the memory
- Disadvantage: Needs process synchronization



Shared Memory in Linux

- **int shmget (key, size, flags)**
  - Create a shared memory segment;
  - Returns ID of segment : shmid
  - key : unique identifier of the shared memory segment
  - size : size of the shared memory (rounded up to the PAGE_SIZE)
- **int shmat(shmid, addr, flags)**
  - **At**tach shmid shared memory to address space of the calling process
  - addr : pointer to the shared memory address space
- **int shmdt(shmid)**
  - **D**e**t**ach shared memory

Example

server.c

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define SHMSIZE     27 /* Size of shared memory */

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    key = 5678; /* some key to uniquely identifies the shared memory */

    /* Create the segment. */
    if ((shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /* Attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /* Now put some things into the shared memory */
    s = shm;
    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = 0; /* end with a NULL termination */

    /* Wait until the other process changes the first character
     * to '*' the shared memory */
    while (*shm != '*')
        sleep(1);
    exit(0);
}
```

client.c

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define SHMSIZE     27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

     /* We need to get the segment named "5678", created by the server
    key = 5678;

    /* Locate the segment. */
    if ((shmid = shmget(key, SHMSIZE, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /* Attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /* read what the server put in the memory. */
    for (s = shm; *s != 0; s++)
        putchar(*s);
    putchar('\n');

    /*
     * Finally, change the first character of the
     * segment to '*', indicating we have read
     * the segment.
     */
    *shm = '*';

    exit(0);
```

Problems with Shared memory:

- Normally, OS tries to prevent one process from accessing another process's memory.
- Shared memory requires to remove this restriction.
- Processes are also responsible for ensuring that they are not writing to the same location, simultaneously.

Producer –Consumer Problem Example: A web server produces HTML files and images, which are consumed by the client browser.

Solution: A shared buffer that can be filled by the producer and emptied by the consumer

SHARED MEMORY: PRODUCER-CONSUMER PROBLEM

- A producer process produces information which is consumed by a consumer process.
- Both share a common buffer which can be of two types:
  - **unbounded-buffer:** Unlimited size of buffer
  - **bounded-buffer:** limited buffer size

SOLUTION WITH BOUNDED-BUFFER

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

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

Producer

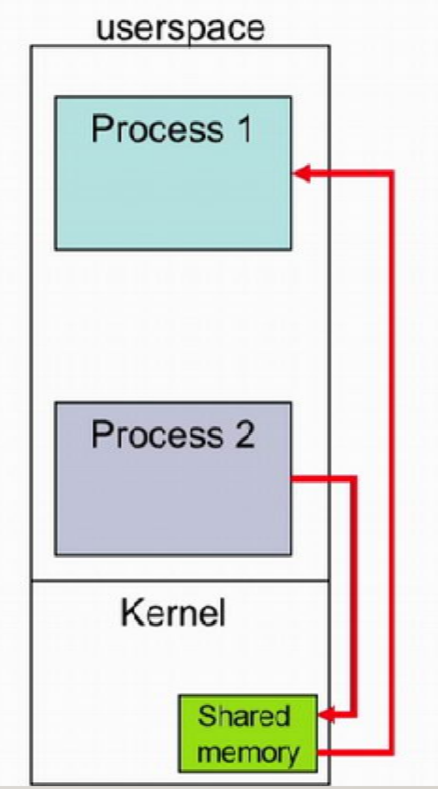
```
item next_produced;
while (true) {
     /* produce an item in next_produced */

     while (((in + 1) % BUFFER_SIZE) == out)
       ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

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 */
}
```

Figure 3.13 and 3.14 from Ref.1

MESSAGE PASSING



- A shared memory is created in the kernel
- System calls used for operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
  - Each send should have a corresponding receive: Process Cooperation
- *Message* can be of either fixed or variable size
- Processes need to establish a ***communication link*** between them and use provided operations
- Each call requires marshalling and de-marshalling of information.
- Message passing is useful for exchanging smaller amounts of data

MESSAGE PASSING

- Implementing communication link:
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

MESSAGE PASSING : DIRECT COMMUNICATION

- Each process must have a name and used in the following manner:
  - **`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
  - Exactly one link is associated with each pair of communicating processes
  - Usually the link is bi-directional, but can be made unidirectional

MESSAGE PASSING : INDIRECT COMMUNICATION

- Processes share a mailbox, also called as ports
- Each mailbox has a unique id
- mailboxes() are used to send or receive Messages
- Properties of communication link
  - Share a common mailbox between processes establishes a link
  - Many processes can share a link
  - Each pair of processes may share several communication links
  - It can be unidirectional or bi-directional

MESSAGE PASSING : INDIRECT COMMUNICATION

Procedure

- create a new mailbox (port)
- send and receive messages through mailbox
  - send(A, message)—Send a message to mailbox A.
  - receive(A, message)—Receive a message from mailbox A.
- destroy a mailbox

Example :

- *A, B,* and *C* share a mailbox M1
- *A* sends; *B* and *C* receive

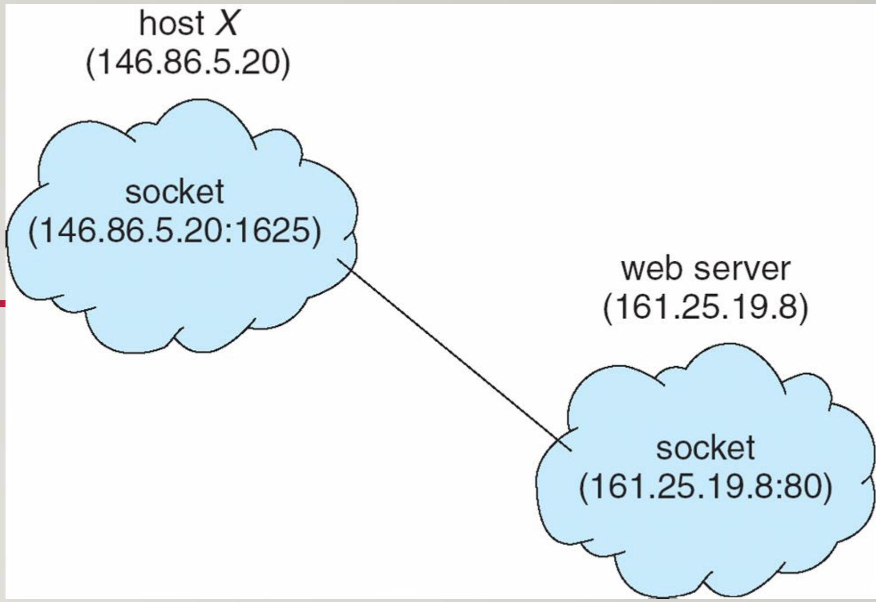
Signals

- Asynchronous unidirectional communication between processes
- Signals are a small integer
  - eg. 9: kill, 11: segmentation fault
- Send a signal to a process
  - kill(pid, signum)
- Process handler for a signal
  - sighandler_t signal(signum, handler);
  - Default if no handler defined

CLIENT-SERVER SYSTEMS: Communication

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

SOCKETS



- **Sockets:** an endpoint for communication
- One socket for each process
- Identified by an IP address concatenated with a port number
- Port no. helps in differentiating applications on a host
- The socket **121.65.12.8:1428** refers to port **1428** on host **121.65.12.8**
- Port no.s below 1024 are ***well known*** *and* used for standard services
- 127.0.0.1 is a Special IP address (**loopback**) used to refer to the system on which process is running

REMOTE PROCEDURE CALLS

- The calls between processes which are separated by networked systems
- Uses ports to differentiate between services
- Each message is sent to a port on the remote system.
- Each message contains an identifier specifying the function to execute and the parameters to pass to that function.
- The function is executed and output is sent back in a separate message.
- It allows a client to invoke a procedure on a remote host and gives an illusion of calling the procedure locally

PIPES

Following issues need to be considered while creating pipes:

- There exists a relationship (i.e., ***parent-child***) between the communicating processes?
- Pipes are going to be used within a machine or over a network?
- Unidirectional or bidirectional communication?
- Half or full-duplex communication?

- Ordinary pipes: it is created by a parent process to facilitate communication with its child process. It cannot be accessed from outside the process
- Named pipes: it can be accessed without a parent-child relationship.

ORDINARY PIPES

- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Unidirectional
- parent-child relationship is mandatory between communicating processes
- In Windows, these are called **anonymous pipes**

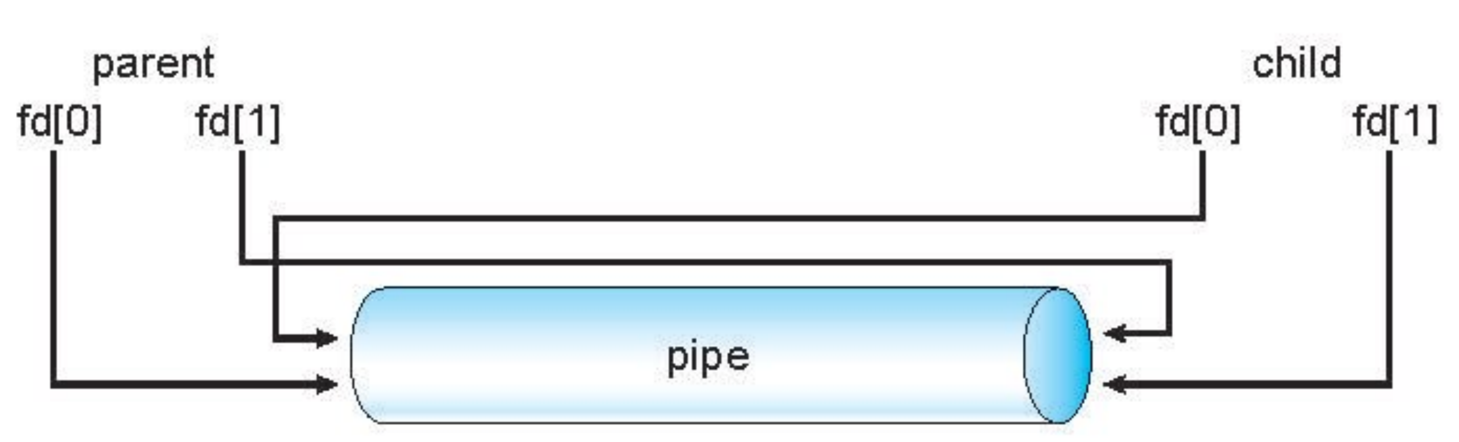
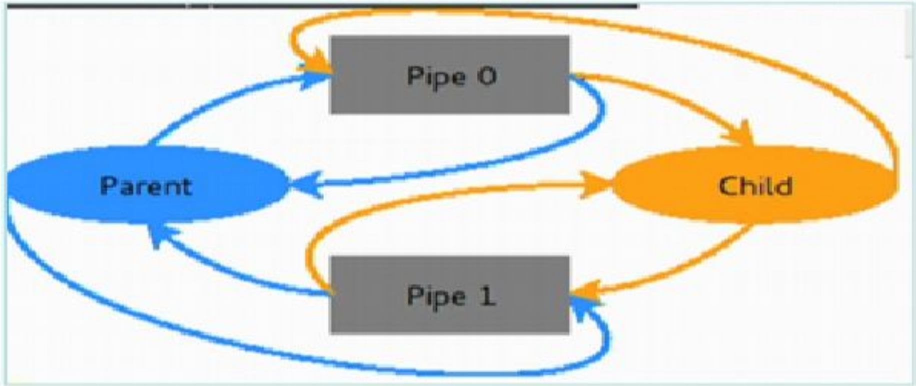


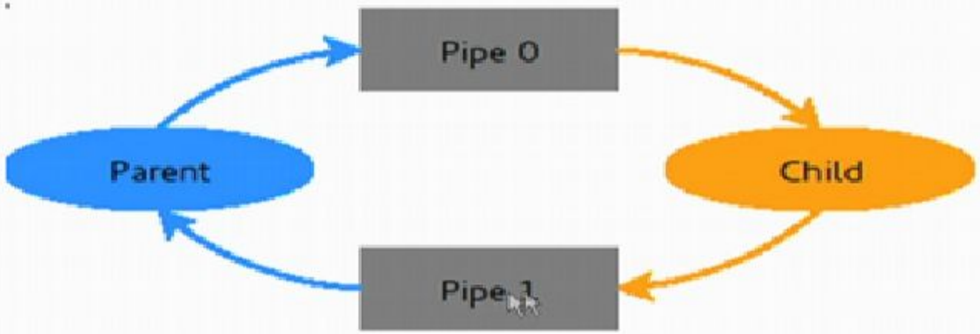
Figure 3.24 from Ref.1

NAMED PIPES

- Better than ordinary pipes
- Bidirectional
- Parent-child relationship is not necessary
- Many processes can use it for communication
- Available with both Windows systems and UNIX
- Continue to exist after communicating processes have finished.
- Named pipes are referred to as FIFOs in UNIX systems.
- A FIFO is created with the mkfifo() system call and manipulated with the ordinary open(), read(), write(), and close() system calls.
- UNIX use only half-duplex transmission
- Intermachine communication needs sockets
- Named pipes on Windows systems provide full-duplex communication and the communicating processes may reside on either the same or different machines.
- Only byte-oriented data transmitted across a UNIX FIFO, whereas Windows systems allow either byte- or message-oriented data.

Pipes for two way communication





- Two pipes opened
  pipe0 and pipe1
- Note the unnecessary pipes

- Close the unnecessary pipes

Example
(child process sending a string to parent)

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(){
  int pipefd[2];
  int pid;
  char recv[32];

  pipe(pipefd);

  switch(pid=fork()) {
  case -1: perror("fork");
           exit(1);
  case 0:                                        /* in child process */
        close(pipefd[0]);                        /* close unnecessary pipefd */
        FILE *out = fdopen(pipefd[1], "w"); /* open pipe descriptor as stream */
        fprintf(out, "Hello World\n");           /* write to out stream */
        break;
  default:                                       /* in parent process */
        close(pipefd[1]);                        /* close unnecessary pipefd */
        FILE *in = fdopen(pipefd[0], "r");       /* open descriptor as stream */
        fscanf(in, "%s", recv);                  /* read from in stream */
        printf("%s", recv);
        break;
  }
}
```