BITS Pilani
Hyderabad Campus

OPERATING SYSTEMS (CS F372)

Processes

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus

Process Concept

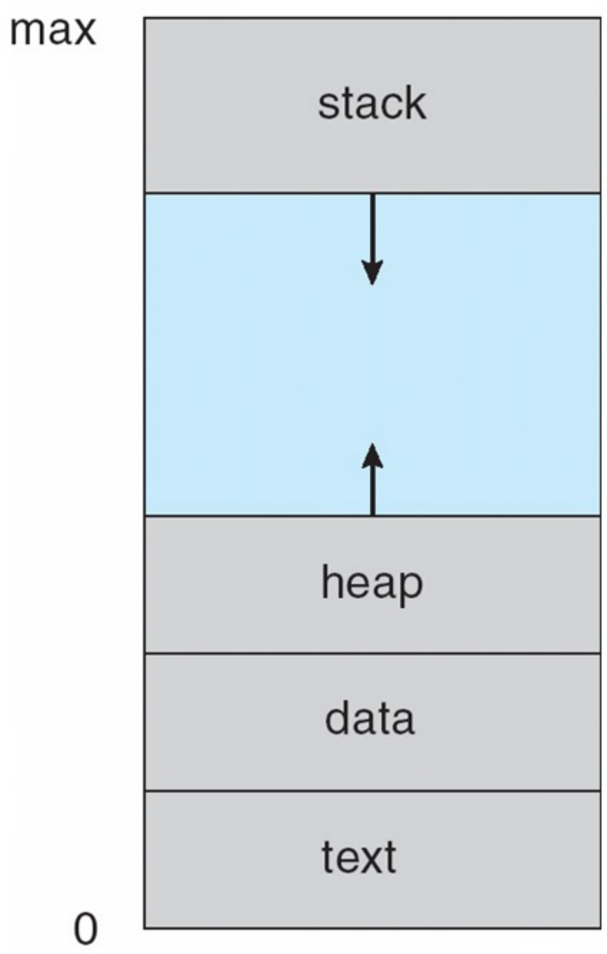
- An operating system executes a variety of programs
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Terms ***job*** and ***process*** used interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion

Process Concept

- 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

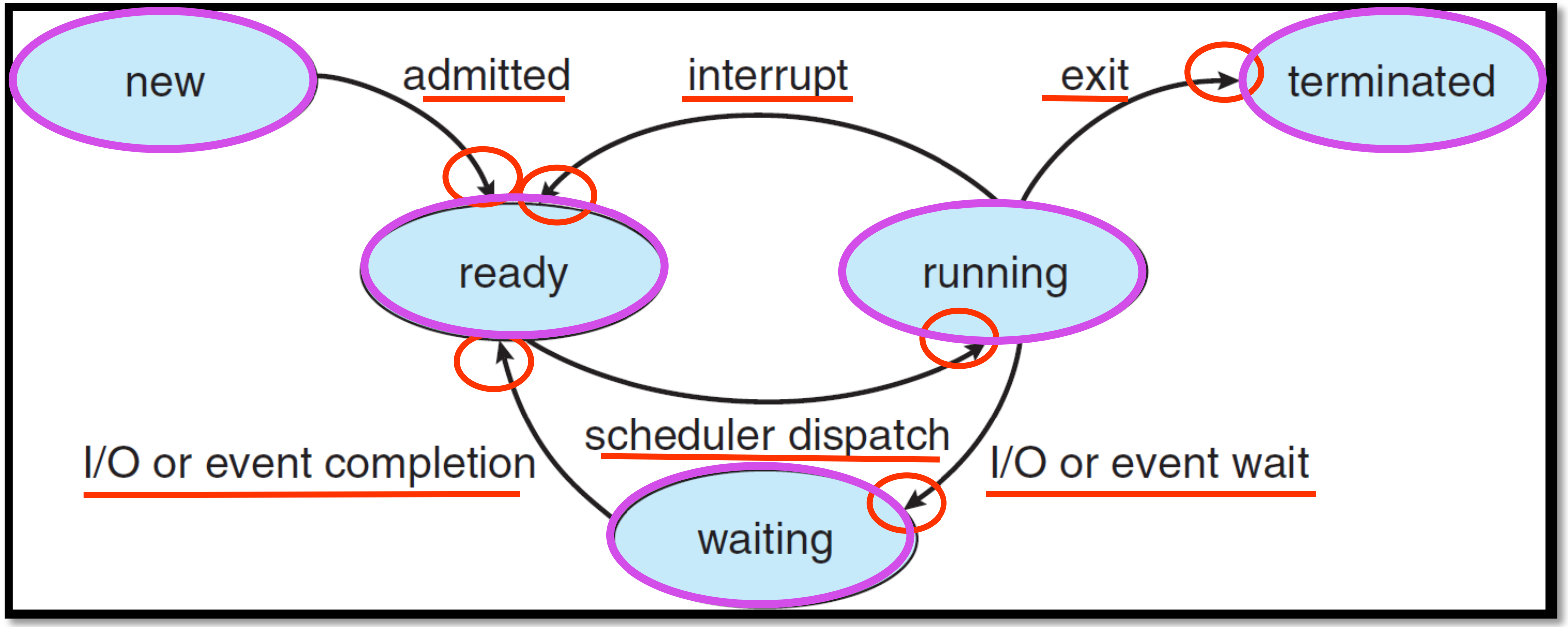
Process Concept

- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file 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



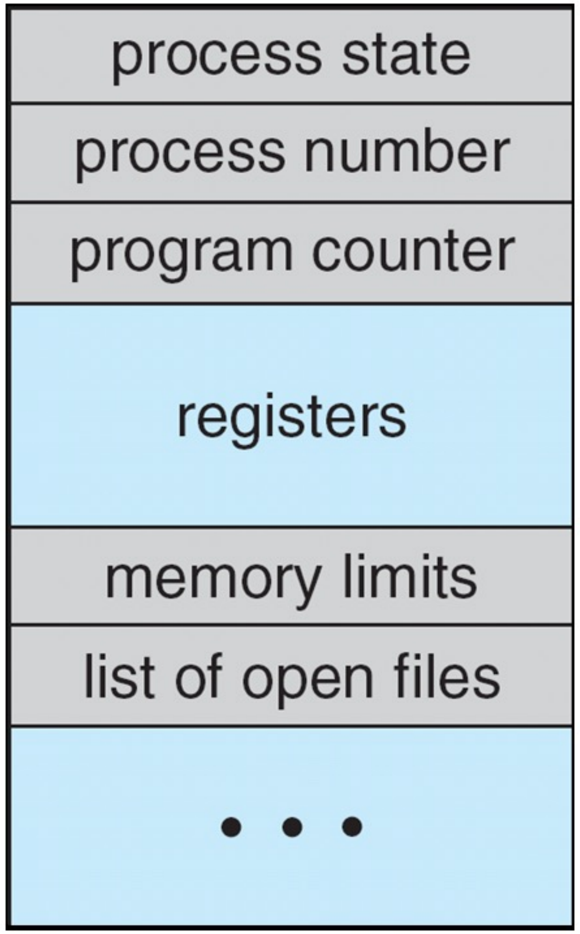
States of Process

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



Process Control Block

- **Process state** – new, ready, 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, scheduling parameters
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits, account nos., process nos.
- **I/O status information** – I/O devices allocated to process, list of open files



Process Creation

- **Parent** process creates **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**), integer number
- Resource sharing options (CPU time, memory, files, I/O devices)
  - 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

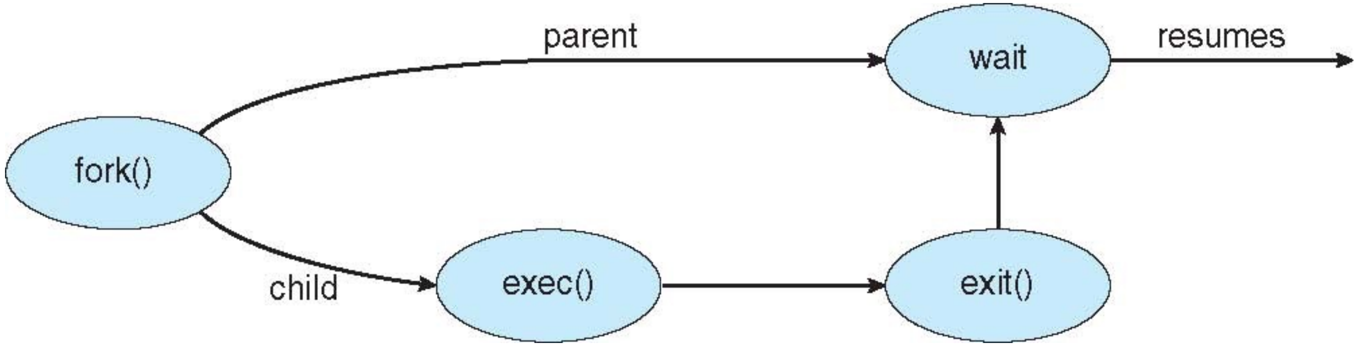
- Address space
  - Child duplicate of parent (same program and data)
  - Child has a program loaded into it
- UNIX examples
  - fork() system call creates new process
  - exec() system call used after a fork() to replace the process’s memory space with a new program

Process Creation

- **fork()**
  - address space of child process is a copy of parent process
  - both child and parent continue execution at the instruction after fork()
  - return code for fork() is 0 for child
  - return code for fork() is non-zero (child *pid*) for parent
- **exec()**
  - loads a binary file into memory and starts execution
  - destroys previous memory image
  - call to exec() does not return unless an error occurs
- **wait()**
  - parent can issue wait() to move out of ready queue until the child is done

Process Creation

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include<sys/wait.h>
int main()
{
		pid_t pid;
		pid = fork(); /* fork a child process */
		if (pid < 0) { /* error occurred */
				fprintf(stderr, "Fork Failed");
				return 1;
		}
		else if (pid == 0) { /* child process */
				printf("Child Process\n");
				execlp("/bin/ls","ls",NULL);
		}
		else { /* parent process */
				wait(NULL); /* parent waits for child to complete */
				printf("Child Complete");
		}
		return 0;
}
```



OUTPUT:

```
Child Process
a.out
Documents  examples.desktop  MyPrograms  Pictures  Templates
Desktop  Downloads  Music     parent.c  Public   Videos
Child Complete
```

Example 1

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
        fork();
        printf ("hello\n");
        return (0);
}
```

hello
hello

Example 2

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
        int x;
        x = fork();
        if (x == 0 )
            printf ("Child Process :%d", x);
        else
            printf ("I am parent : %d", x);
        return (0);
}
```

Child Process : 0
I am Parent : 1234

Example 3

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{

        fork();
        fork();
        fork();
        printf("Hello");
        return (0);
}
```

Process Creation

```
int main()
{
        pid_t pid;
        pid = fork();
        if (pid < 0) {
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        else if (pid == 0) {
                printf("Child Process\n");
                printf("child pid = %d\n", getpid());
        }
        else {
                printf("parent pid = %d\n", getpid());
                wait(NULL);
                printf("Child Complete");
        }
        return 0;
}
```

OUTPUT:

```
parent pid = 6597
Child Process
child pid = 6598
Child Complete
```

Process Creation

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include<sys/wait.h>
int main()
{
        pid_t pid;   int x = 10;
        pid = fork();
                if (pid < 0) {
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        else if (pid == 0) {
                printf("Child Process: x = %d\n", x);
                execlp("/bin/ls","ls",NULL);
        }
        else {
                printf("Parent Process: x = %d\n", x);
                wait(NULL);
                printf("Child Complete");
        }
        return 0;
}
```

OUTPUT:

Parent Process: x = 10
Child Process: x = 10
a.out Documents examples.desktop MyPrograms Pictures Templates
Desktop Downloads Music parent.c Public Videos
Child Complete

Process Creation

```
int main()
{
        pid_t pid;
        int x = 10;
        pid = fork();
        if (pid < 0) {
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        else if (pid == 0) {
                x = x + 10;
                printf("Child Process: x = %d\n", x);
        }
        else {
                wait(NULL);
                printf("Parent Process: x = %d\n", x);
                printf("Child Complete");
        }
        return 0;
}
```

OUTPUT:

Child Process: x = 20

Parent Process: x = 10

Child Complete

Process Creation

```
int main()
{
        pid_t pid;
        int x = 10;
        pid = fork();
        if (pid < 0) {
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        else if (pid == 0) {
                for(long i = 0; i < 50000000000; i++);
                printf("Child Process: x = %d\n", x);
        }
        else {
                x = x + 10;
                wait(NULL);
                printf("Parent Process: x = %d\n", x);
                printf("Child Complete");
        }
        return 0;
}
```

OUTPUT:

Child Process: x = 10

Parent Process: x = 20

Child Complete

Process Termination

- Process executes last statement and then asks the operating system to delete it using the exit() system call.
- May return status data from child to parent (via wait())
- Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes because:
    - Child has exceeded allocated resources limit
    - Task assigned to child is no longer required
    - Parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exist if 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_t pid;
int status;
pid = wait(&status); //parent can tell which child has terminated
```

- If no parent waiting (did not invoke **wait()** till then) process is a **zombie**
- If parent terminated, process is an **orphan**

Zombie Process

```
int main()
{
    pid_t child_pid = fork();
    if (child_pid > 0) {
        sleep(10);          zombie
        wait(NULL);           no longer a zombie
        sleep(200);
    }
    else{
        printf("\n%d",getpid());
        exit(0);
     }
     return 0;
}
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
1	Z	1001	17404	17403	0	80	0	-	0	-	pts/0	00:00:00	a.out <defunct>

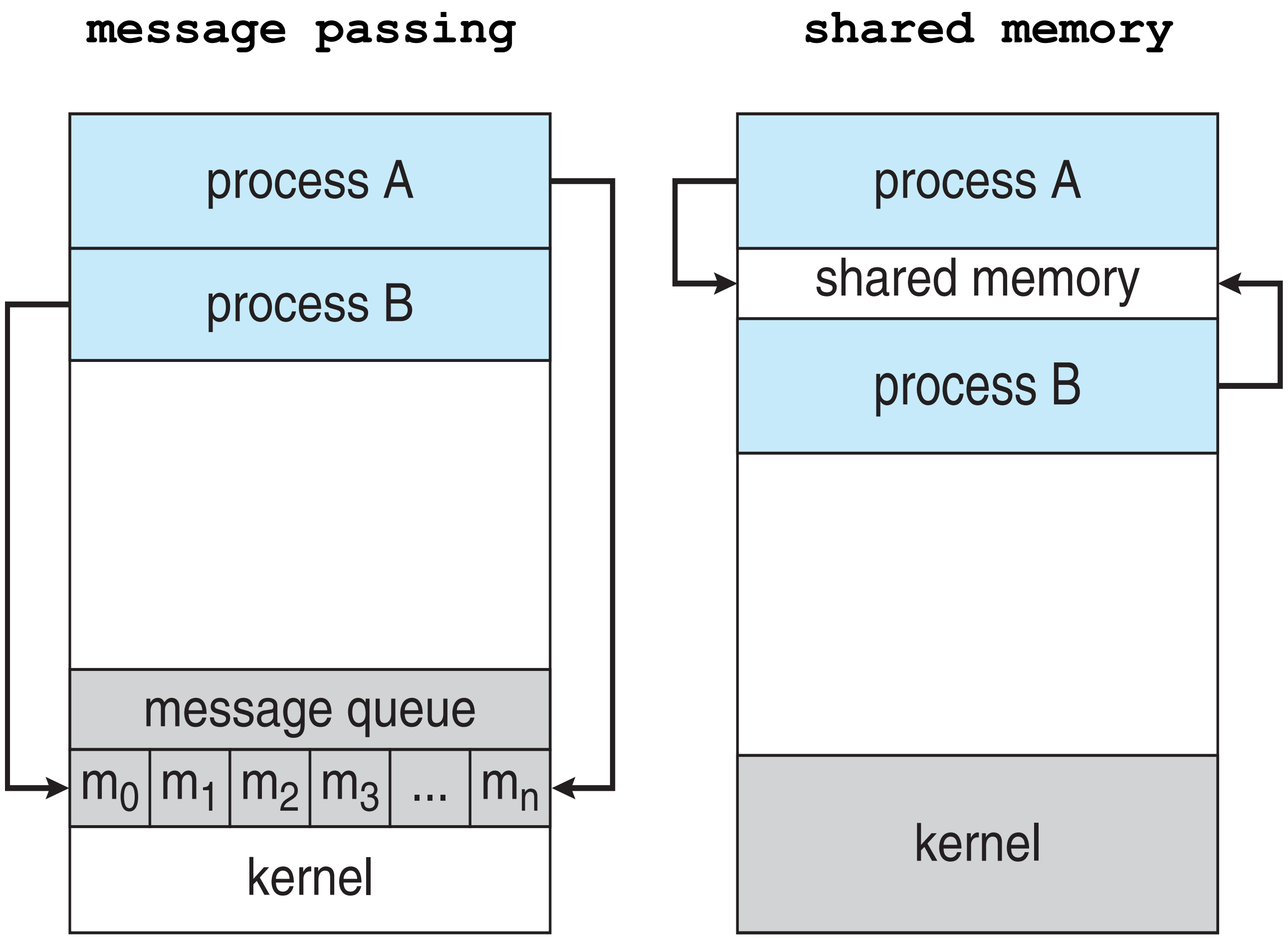
Orphan Process

```
int main()
{
    pid_t child_pid = fork();
    if (child_pid > 0){
        printf("\nParent process: %d\n",getpid());
        sleep(6);
     }
    else{
        printf("\nParent PID: %d\n",getppid());
        sleep(20);
        printf("\nChild Process: %d",getpid());
        printf("\nParent PID: %d",getppid());
        exit(0);
     }

    return 0;
}
```

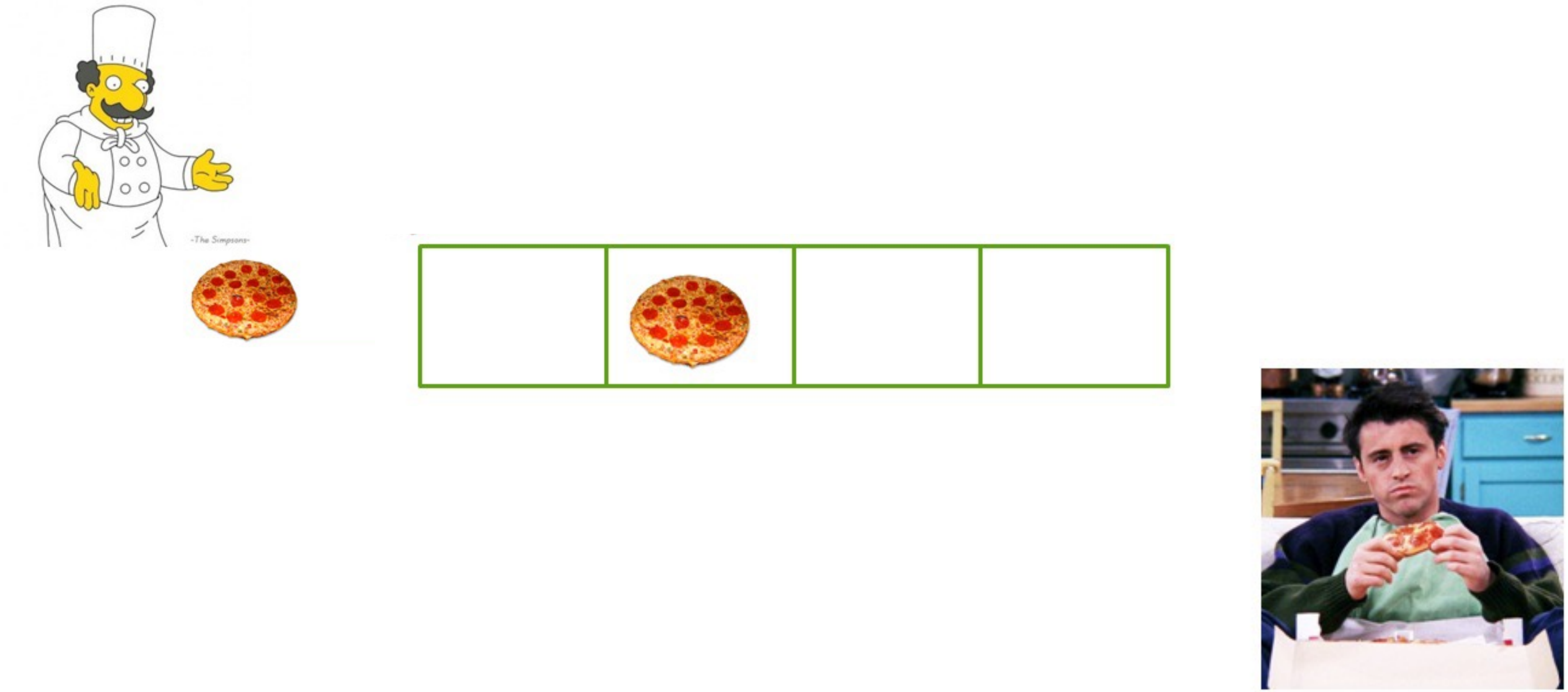
Interprocess Communication

- 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**

Interprocess Communication



Producer-Consumer Problem



IPC – Shared Memory

- An area of memory shared among the processes that wish to communicate

- The communication is under the control of the users processes not the operating system

- Provide mechanism that will allow the user processes to synchronize their actions when they access shared memory

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

- Shared data, reside in a region of memory shared by producer & consumer

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

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

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

Bounded Buffer - Producer

```
item next_produced;
while (true) {
	/* produce an item in next produced */
	while (((in + 1) % BUFFER_SIZE) == out) //buffer is full
		; /* do nothing */
	buffer[in] = next_produced;
	in = (in + 1) % BUFFER_SIZE;
}
```

Producer

Consumer

```
item next_consumed;
while(true){
	while (in == out) //buffer is empty
		; /*do nothing*/
	next_consumed = buffer[out];
	out = (out + 1) % BUFFER_SIZE;
	/* consume the item in next consumed */
}
```

IPC – Message Passing

- Mechanism for processes to communicate and to synchronize their actions

- processes communicate with each other without resorting to shared variables, no sharing of address space

- IPC facility provides two operations:
  - **`send(`*`message`*`)`**
  - **`receive(`*`message`*`)`**

- The *message* size is either fixed or variable

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via `send()` / `receive()`

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
  - Processes only need to know each other's identity
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link

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
- Properties of communication link
  - Link is 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, each link corresponds to one mailbox

Message Passing – Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
- `send(A, message)` – send a message to mailbox A
- `receive(A, message)` – receive a message from mailbox A

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received by the receiving process or mailbox
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continues
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message

Buffering

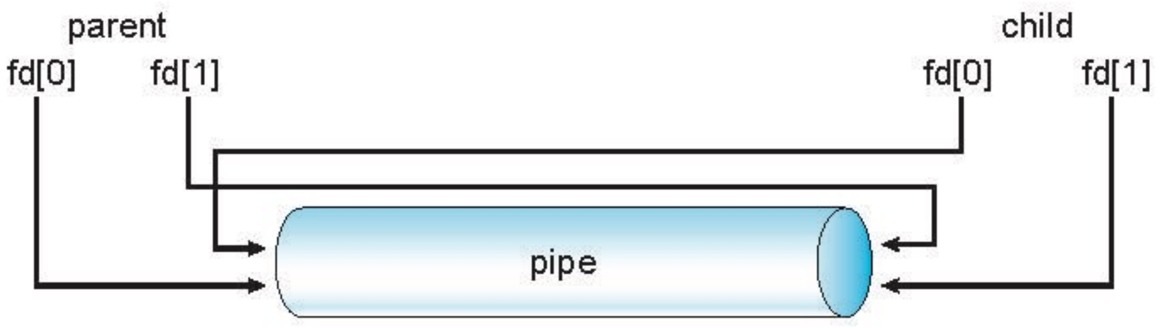
- messages exchanged by communicating processes reside in a temporary queue
- implemented in one of three ways
  - **Zero capacity** – no messages are queued, link can't have any waiting messages, sender must block until receiver receives message
  - **Bounded capacity** – queue is of finite length of *n* messages, sender need not block if queue is not full, sender must wait if queue full
  - **Unbounded capacity** – infinite length queue, sender never blocks

Pipe

- 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** –
  - cannot be accessed from outside the process that created it
  - parent process creates a pipe and uses it to communicate with a child process that it created
- **Named pipes** – can be accessed without a parent-child relationship

Ordinary Pipe

- 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 unidirectional
- Require parent-child relationship between communicating processes
- Windows calls these **anonymous pipes**



Ordinary Pipe

- ordinary pipe can’t be accessed from outside the process that created it
- parent process creates a pipe and uses it to communicate with a child process that it creates via fork()
- child inherits the pipe from its parent process like any other file

Ordinary Pipe

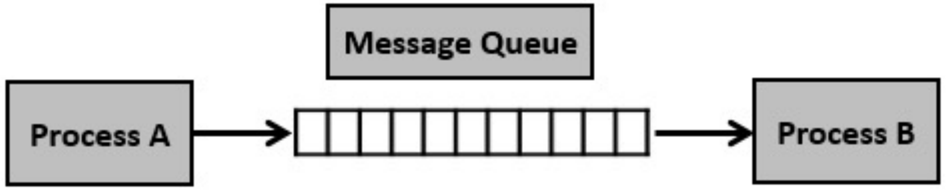
```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;

 /* create the pipe */
 if (pipe(fd) == -1) {
   fprintf(stderr,"Pipe failed");
   return 1;
 }

 /* fork a child process */
 pid = fork();

 if (pid < 0) { /* error occurred */
   fprintf(stderr, "Fork Failed");
   return 1;
 }

if (pid > 0) { /* parent process */
   /* close the unused end of the pipe */
   close(fd[READ_END]);

   /* write to the pipe */
   write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

   /* close the write end of the pipe */
   close(fd[WRITE_END]);
}
else { /* child process */
   /* close the unused end of the pipe */
   close(fd[WRITE_END]);

   /* read from the pipe */
   read(fd[READ_END], read_msg, BUFFER_SIZE);
   printf("read %s",read_msg);

   /* close the write end of the pipe */
   close(fd[READ_END]);
}

return 0;
```

Named Pipe

- 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
- Do not cease to exist if the communicating processes have terminated
- Provided on both UNIX and Windows systems
- Referred to as FIFOs in UNIX systems

Message Queue

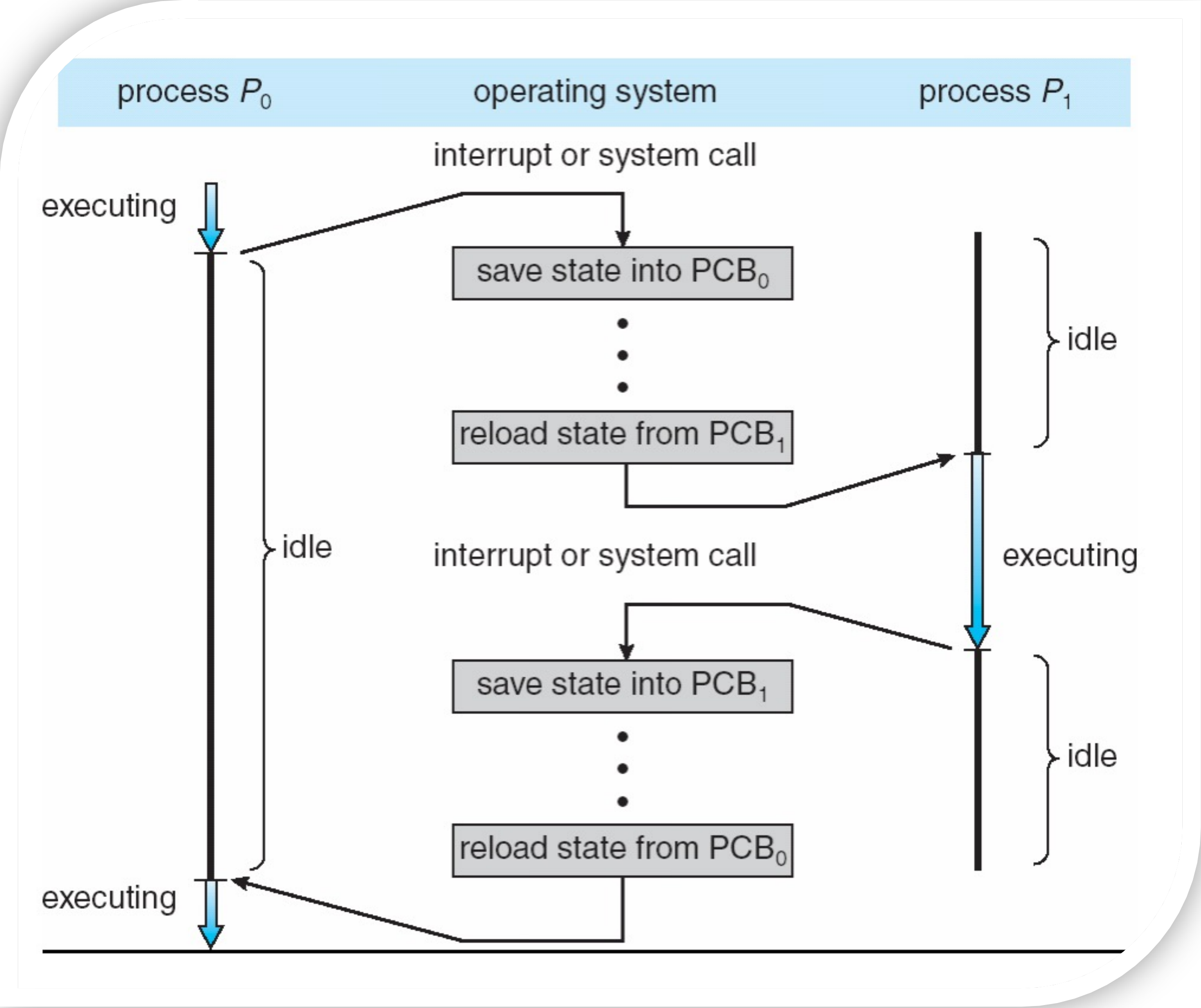
- asynchronous communication
- messages placed onto the queue are stored until the recipient retrieves them



- **Step 1** – Create a message queue or connect to an already existing message queue (msgget())
- **Step 2** – Write into message queue (msgsnd())
- **Step 3** – Read from the message queue (msgrcv())
- **Step 4** – Perform control operations on the message queue (msgctl())

Context Switch

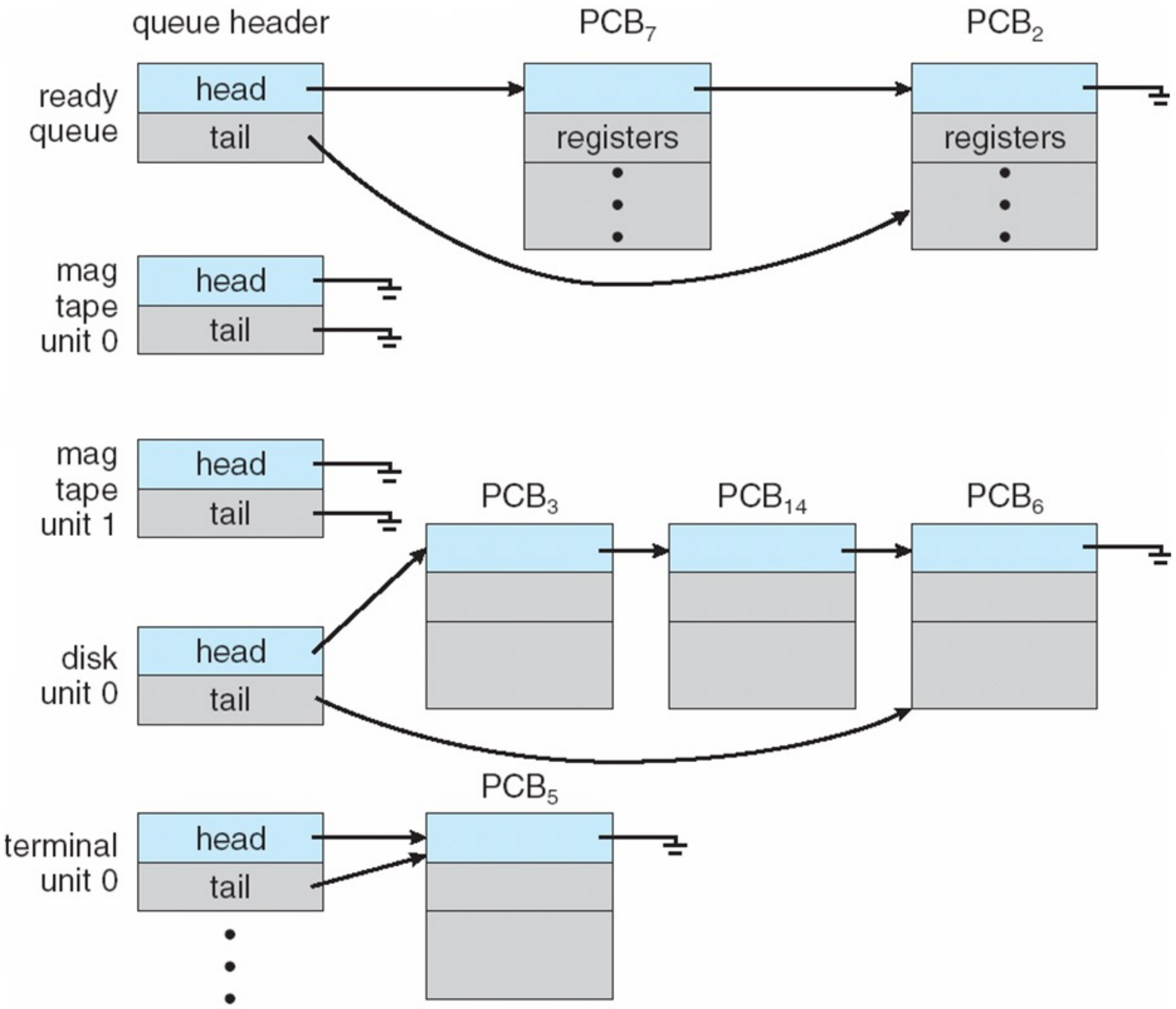
- When CPU switches to another process, system must **save state** of the old process and load the **state** for the new process via a **context switch**
- **Context** of a process represented in the PCB (CPU registers contents, process state, memory management info.)
- Context-switch time is overhead; system does no useful work while switching
- Time is dependent on hardware support



Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute, generally stored as a linked list
  - **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Various Queues

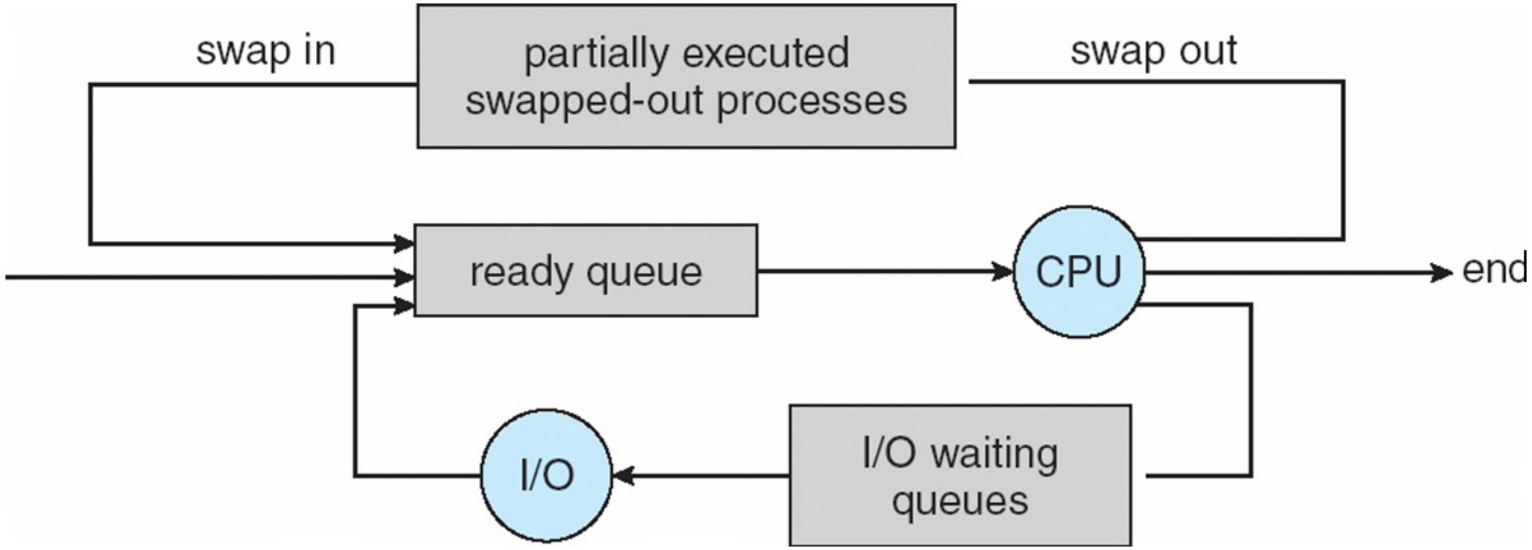


Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) $\Rightarrow$ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes from job queue should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming (number of processes in main memory)**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

Schedulers

- **Medium-term scheduler** can be added in time sharing systems if degree of multiprogramming needs to decrease
  - Intermediate level of scheduling
  - Remove process from memory, store on disk, bring back in from disk to continue execution from where it left off: **swapping**
  - **Required for improving process mix or for freeing of memory**



Thank You