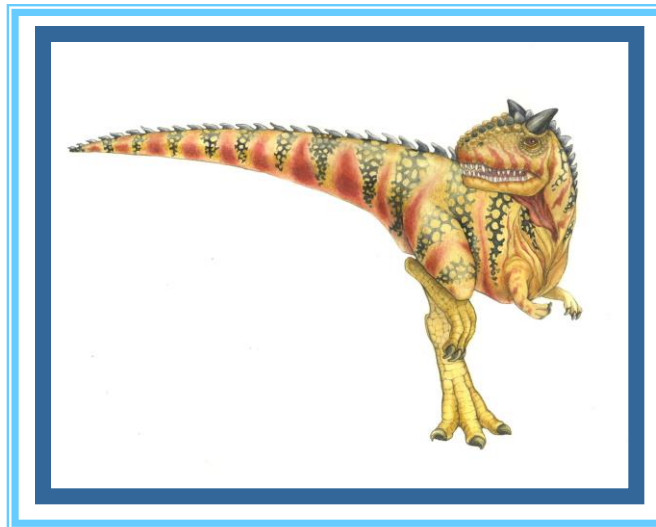


Chapter 3: Process Concept





Chapter 3: Process Concept

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

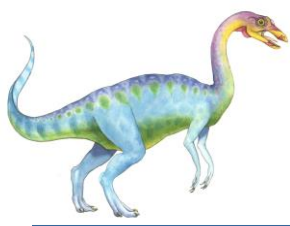




Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing
- To describe communication in client-server systems





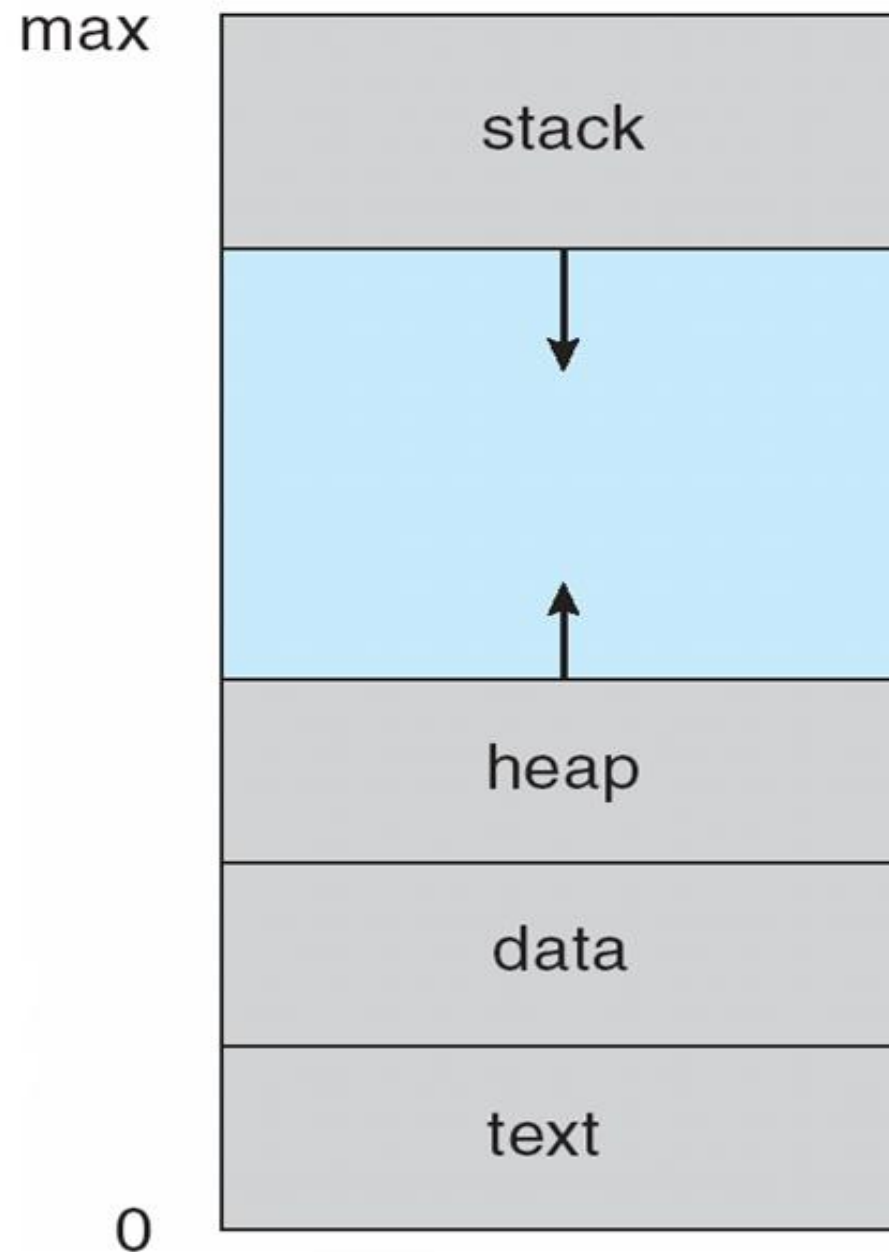
Process Concept

- 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
- 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
- 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





Process in Memory





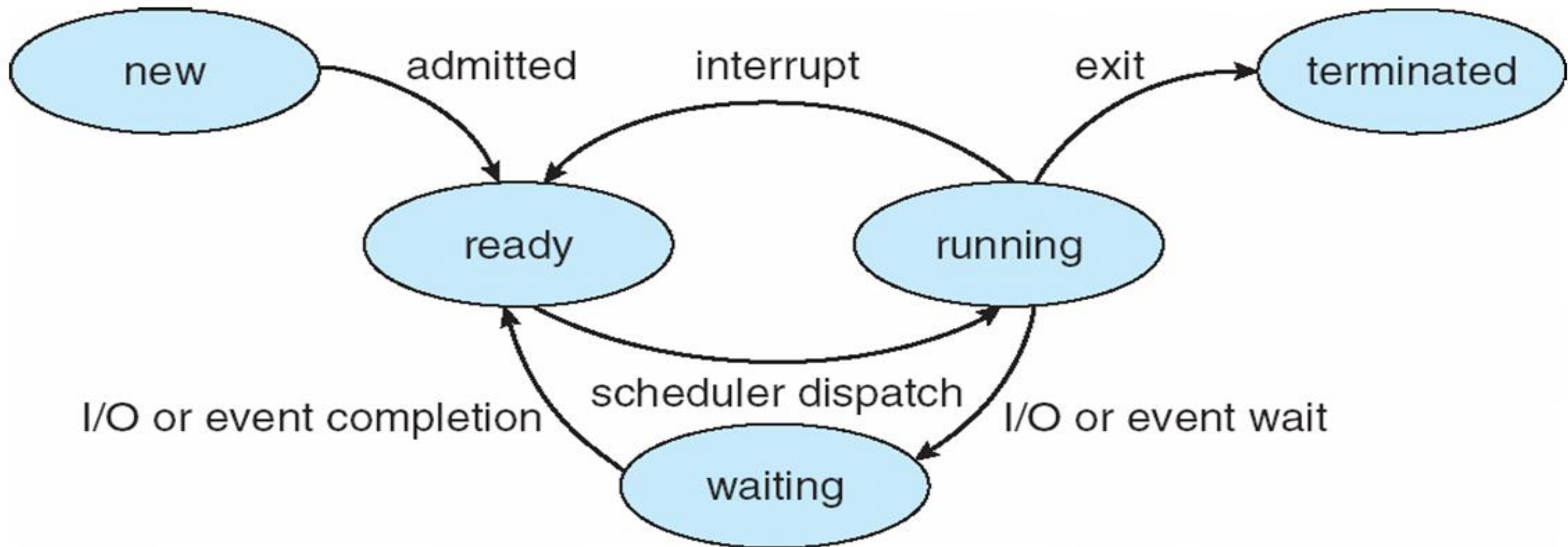
Process State

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





Diagram of Process State



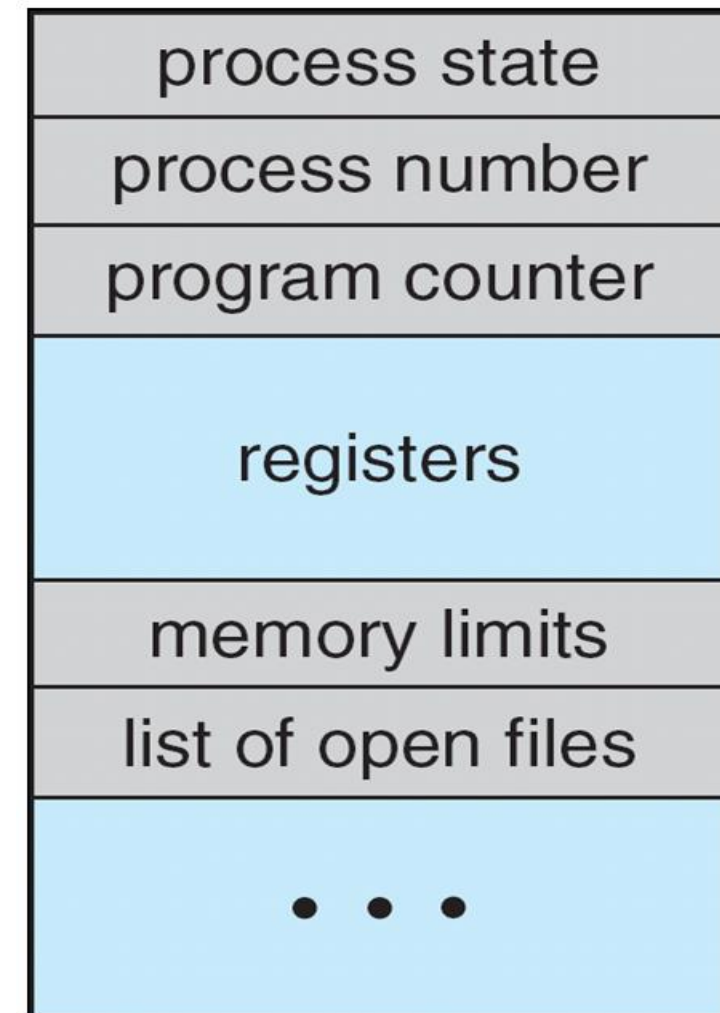


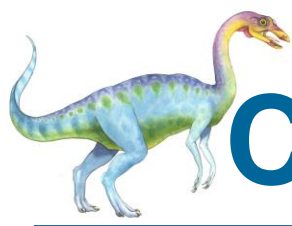
Process Control Block (PCB)

Information associated with each process

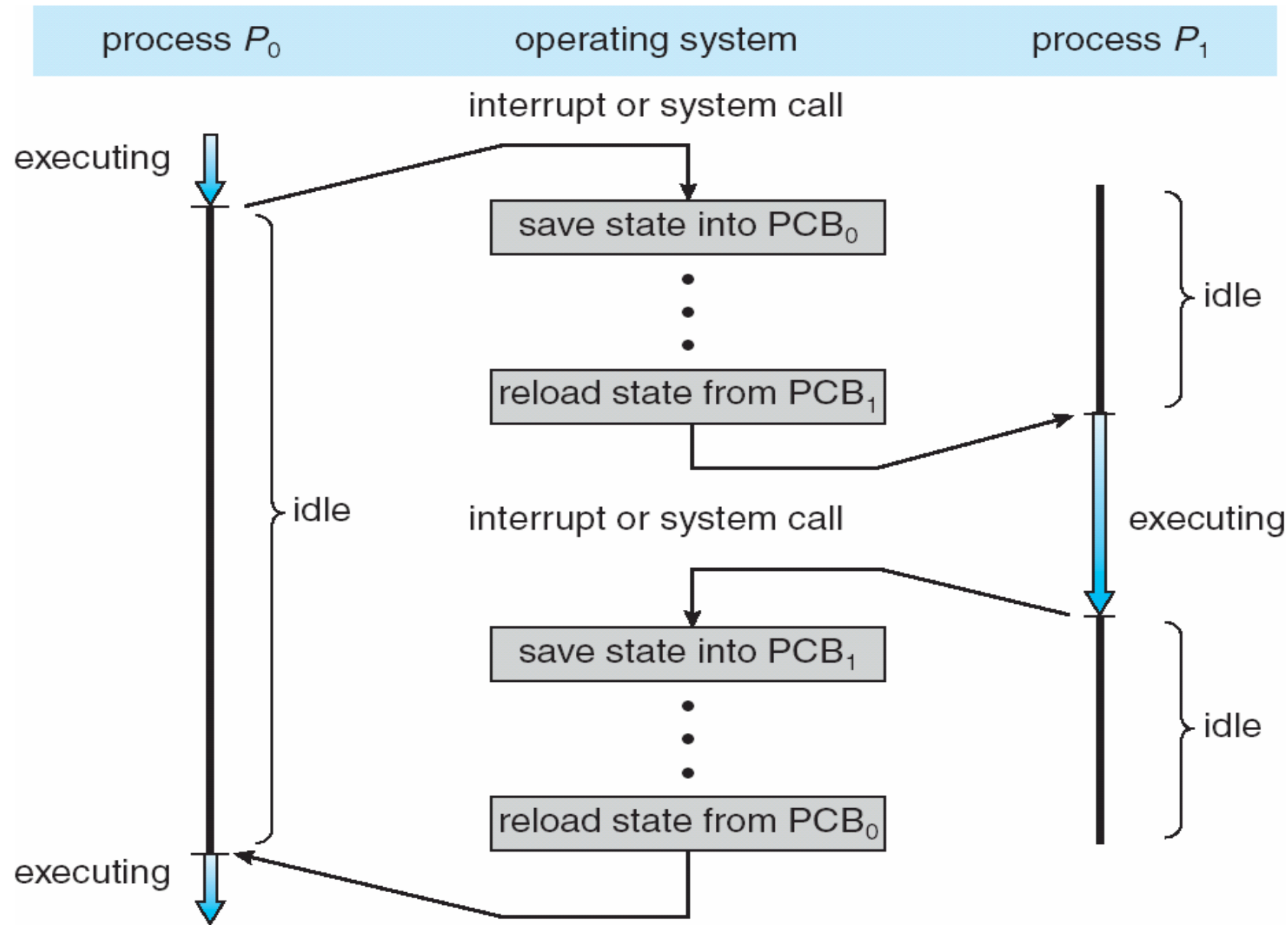
(also called **task control block**)

- ❑ 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





CPU Switch From Process to Process





Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
 - Multiple locations can execute at once
 - ▶ Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- See next chapter

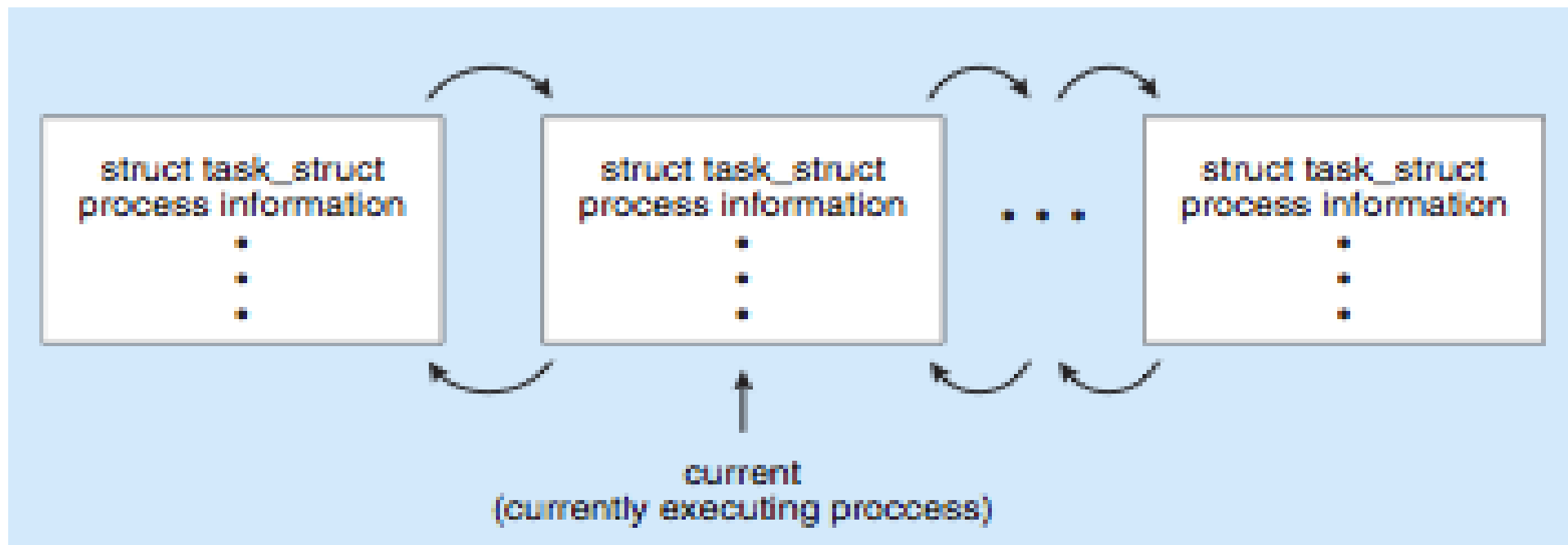


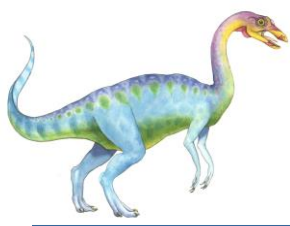


Process Representation in Linux

- 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 */
```





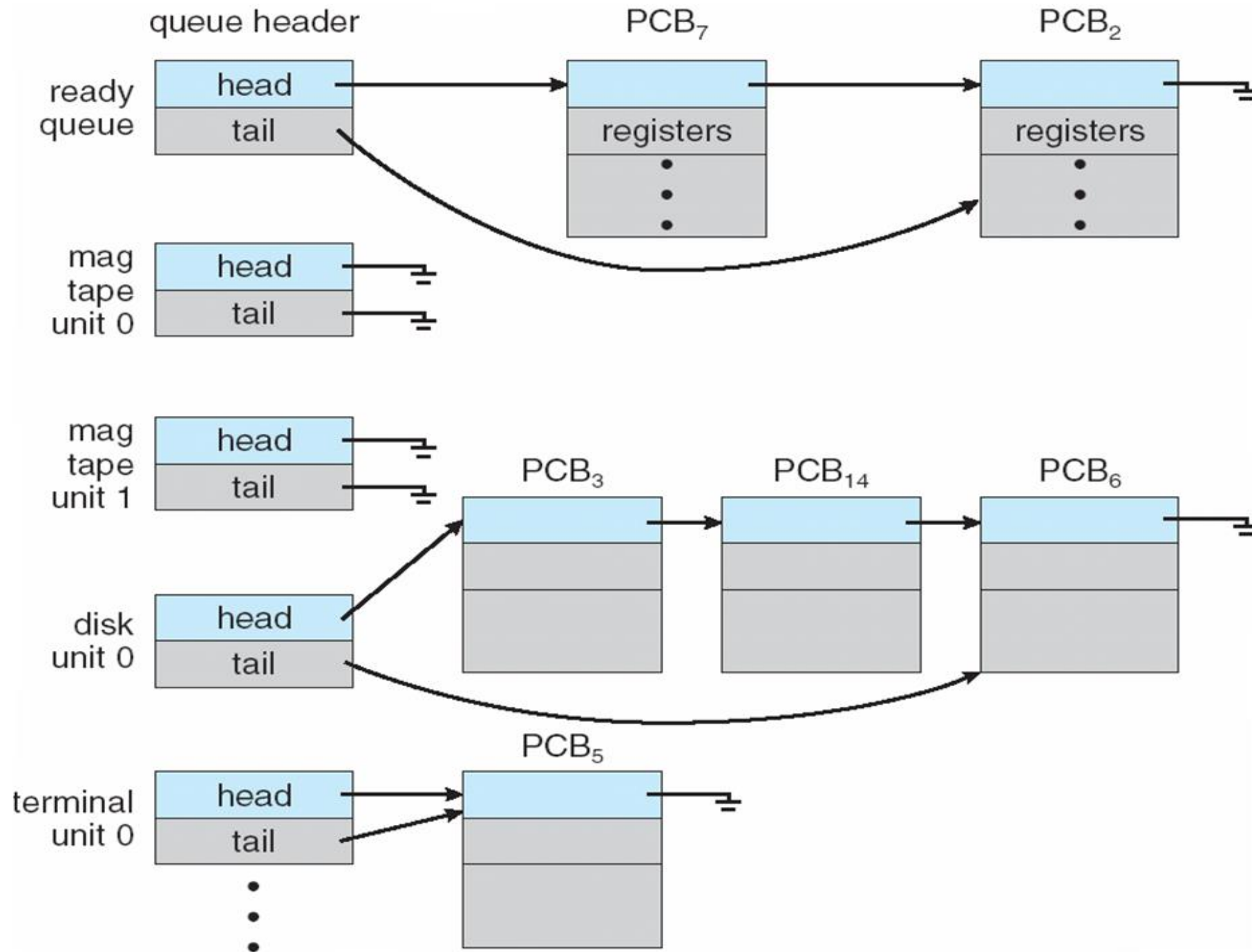
Process Scheduling

- ❑ 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
 - ❑ **Device queues** – set of processes waiting for an I/O device
 - ❑ Processes migrate among the various queues





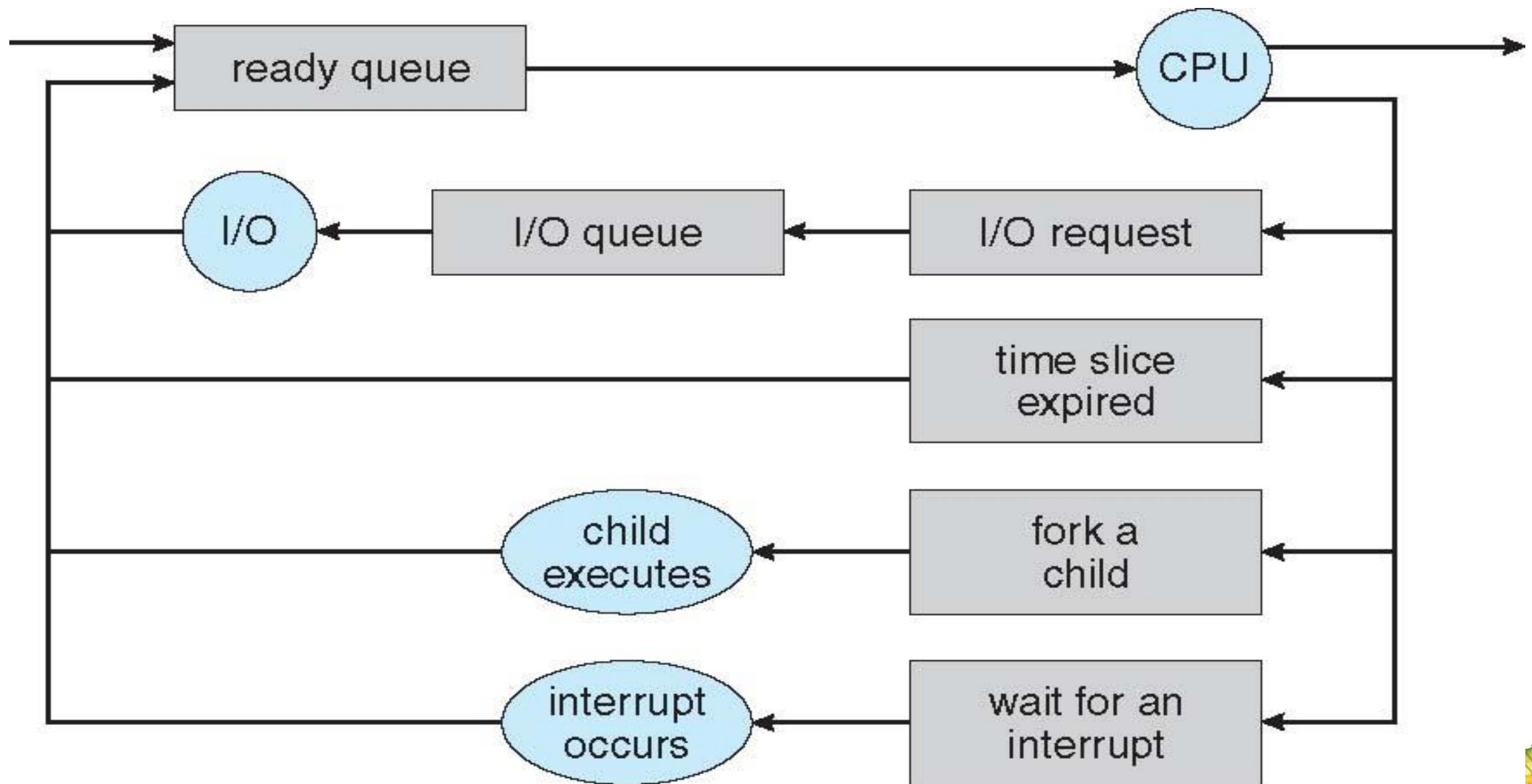
Ready Queue And Various I/O Device Queues





Representation of Process Scheduling

- **Queuing diagram** represents queues, resources, flows

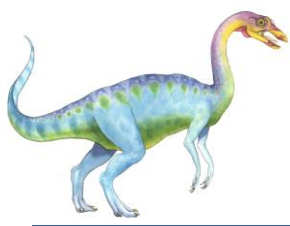




Schedulers

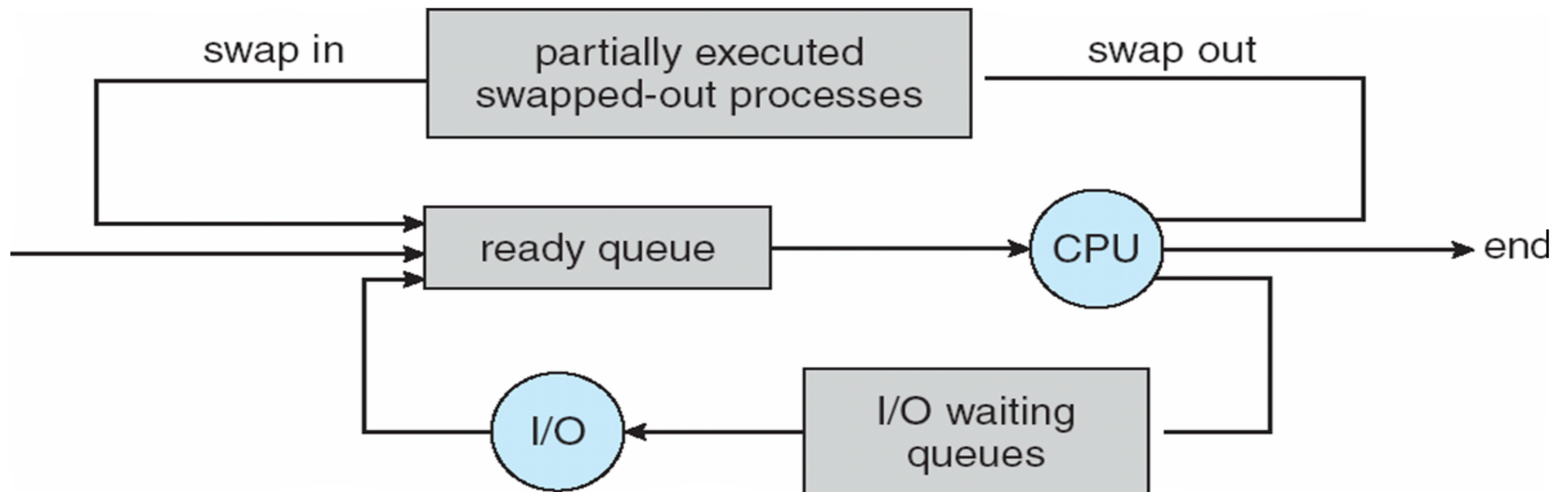
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
 - Sometimes the only scheduler in a system
 - Short-term scheduler is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





Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
 - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**





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 overhead; the system does no useful work while switching
 - The more complex the OS and the PCB -> longer the context switch
- Time dependent on hardware support
 - Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once





Operations on Processes

- System must provide mechanisms for process creation, termination, and so on as detailed next

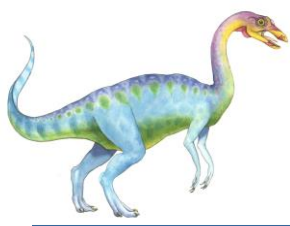




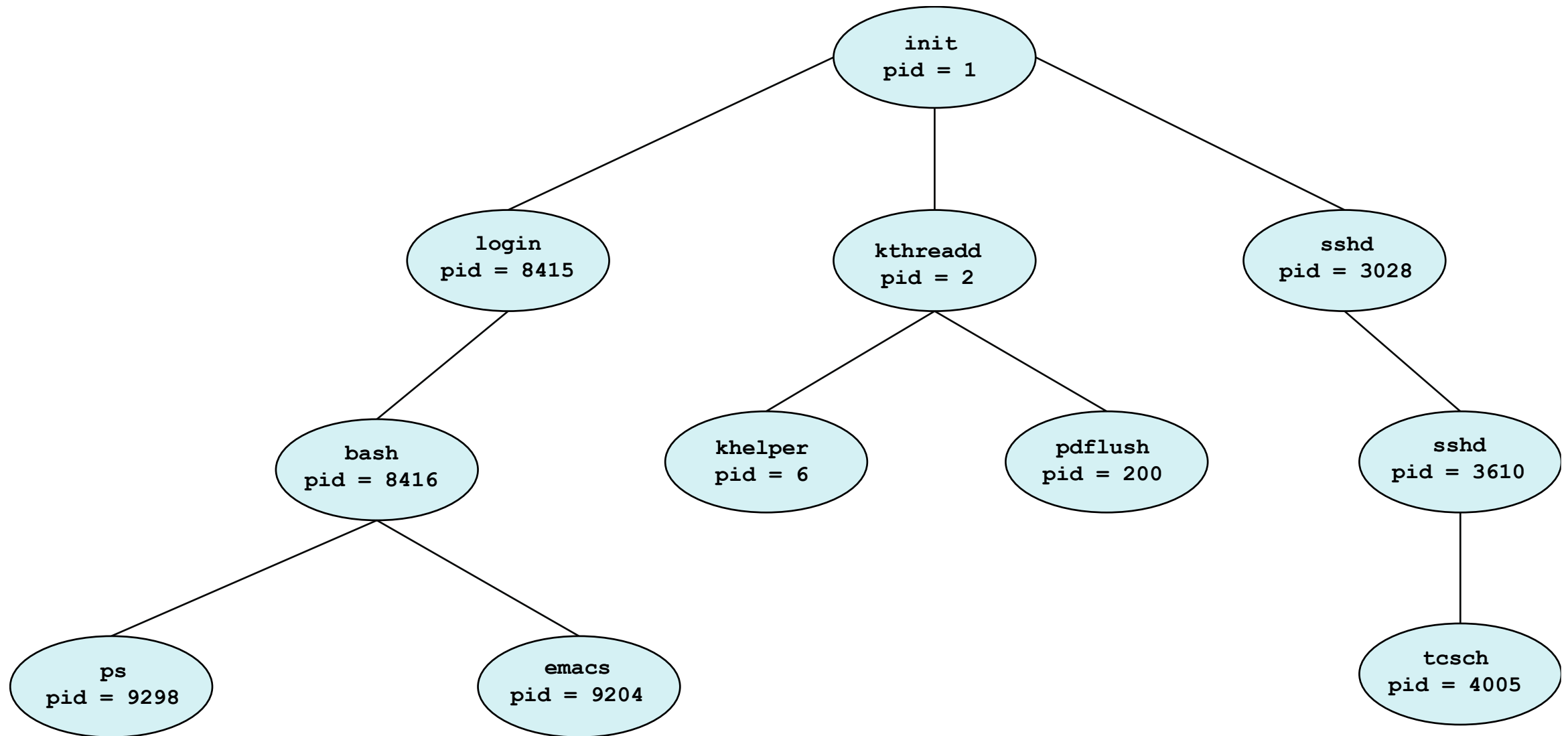
Process Creation

- **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





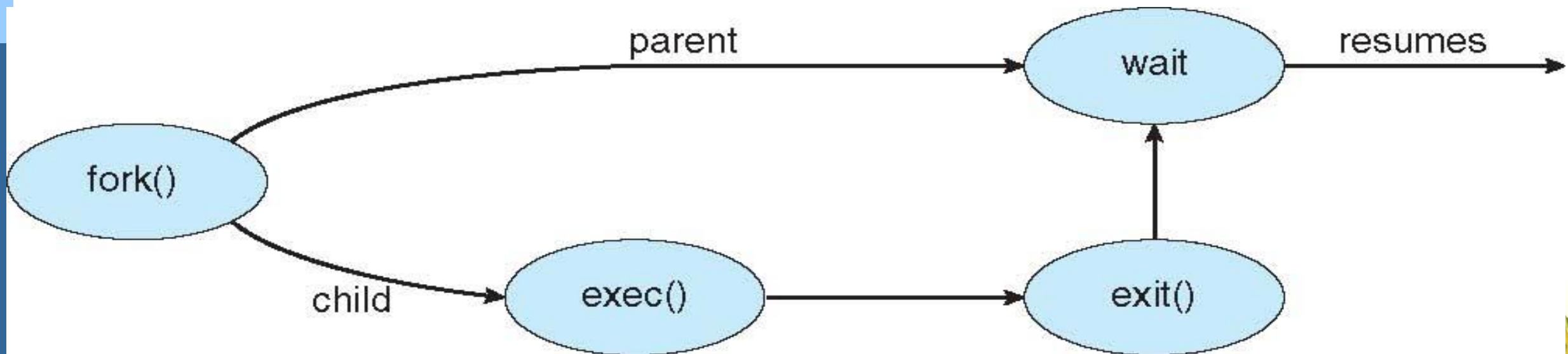
A Tree of Processes in Linux





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





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;
}
```





Process Termination

- Process executes last statement and asks the operating system to delete it (**exit()**)
 - Output data from child to parent (via **wait()**)
 - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort()**)
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - If parent is exiting
 - ▶ Some operating systems do not allow child to continue if its parent terminates
 - All children terminated - **cascading termination**
- Wait for termination, returning the pid:

```
pid_t pid; int status;  
pid = wait(&status);
```
- If no parent waiting, then terminated process is a **zombie**
- If parent terminated, processes are **orphans**

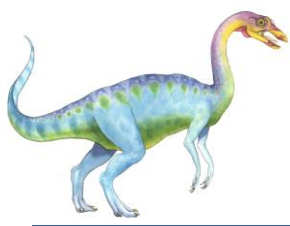




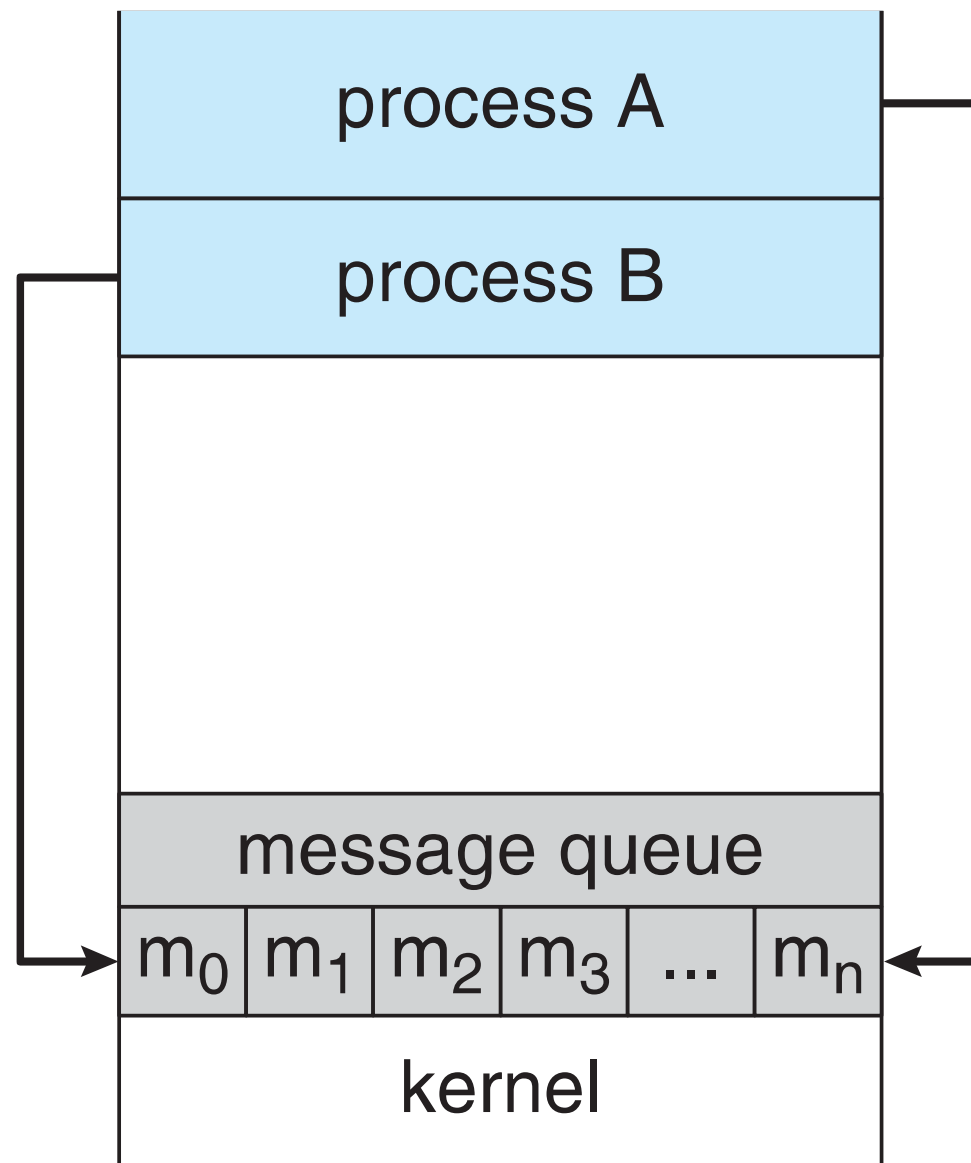
Interprocess Communication

- ❑ 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**

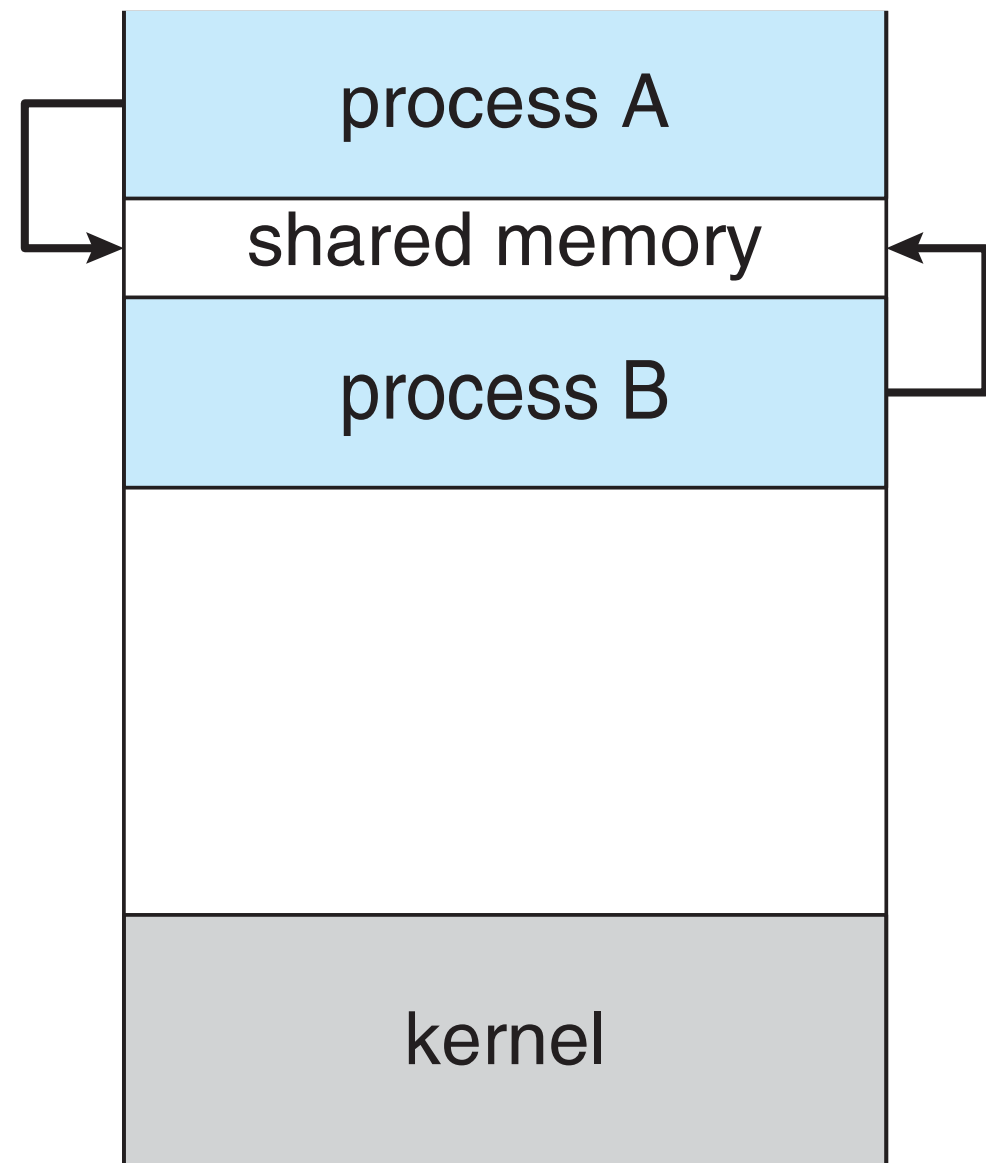




Communications Models



(a)



(b)





Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process

- **Cooperating** process can affect or be affected by the execution of another process

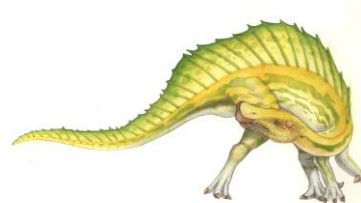
- Advantages of process cooperation
 - Information sharing
 - Computation speed-up
 - Modularity
 - Convenience

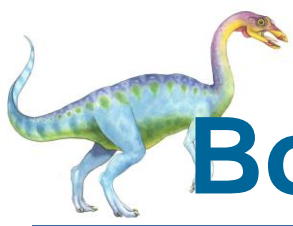




Producer-Consumer Problem

- 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





Bounded-Buffer – Shared-Memory Solution

□ Shared data

```
#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)
        ; /* do nothing */
    buffer[in] = next produced;
    in = (in + 1) % BUFFER SIZE;
}
```





Bounded Buffer – Consumer

```
item next consumed;
while (true) {
    while (in == out)
        ; /* do nothing */
    next consumed = buffer[out];
    out = (out + 1) % BUFFER SIZE;

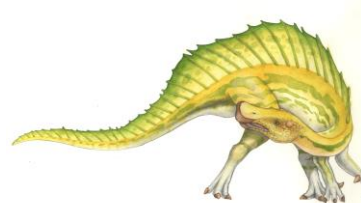
    /* consume the item in next consumed */
}
```





Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- 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., direct or indirect, synchronous or asynchronous, automatic or explicit buffering)

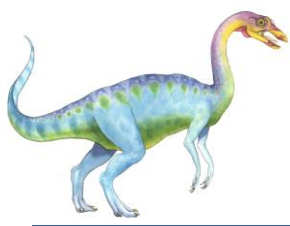




Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?





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





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 established only if processes share a common mailbox
 - A link may be associated with many processes
 - Each pair of processes may share several communication links
 - Link may be unidirectional or bi-directional





Indirect Communication

□ Operations

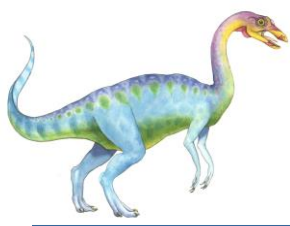
- create a new mailbox
- send and receive messages through mailbox
- destroy a mailbox

□ 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 the message?

- Solutions
 - Allow a link to be associated with at most two processes
 - Allow only one process at a time to execute a receive operation
 - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.





Synchronization

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





Synchronization (Cont.)

- Different combinations possible
 - If both send and receive are blocking, we have a **rendezvous**
- Producer-consumer becomes trivial

```
message next produced;
```

```
while (true) {  
    /* produce an item in next produced */  
    send(next produced);  
}
```

```
message next consumed;
```

```
while (true) {  
    receive(next consumed);  
  
    /* consume the item in next consumed */  
}
```

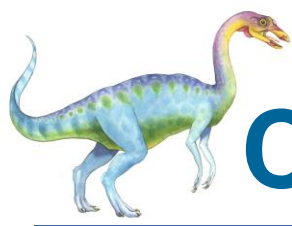




Buffering

- Queue of messages attached to the link;
implemented in one of three ways
 1. Zero capacity – 0 messages
Sender must wait for receiver (rendezvous)
 2. Bounded capacity – finite length of n
messages
Sender must wait if link full
 3. Unbounded capacity – infinite length
Sender never waits



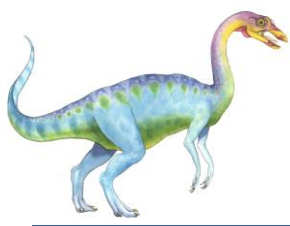


Communications in Client-Server Systems

□ Remote Procedure Calls

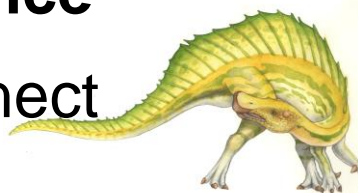
□ Pipes





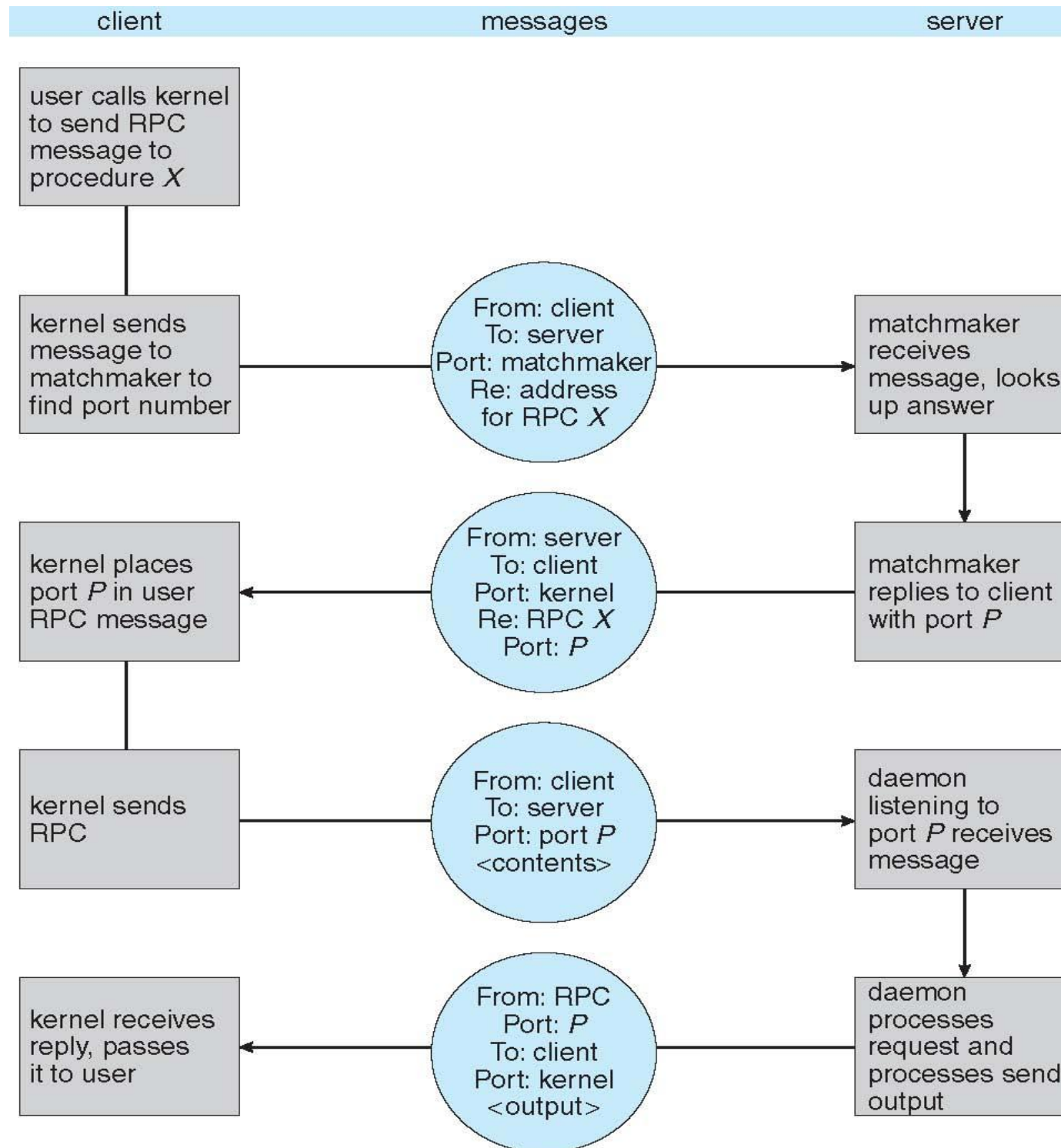
Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
 - Again uses ports for service differentiation
- **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 performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language (MIDL)**
- Data representation handled via **External Data Representation (XDL)** format to account for different architectures
 - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
 - Messages can be delivered **exactly once** rather than **at most once**
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server





Execution of RPC



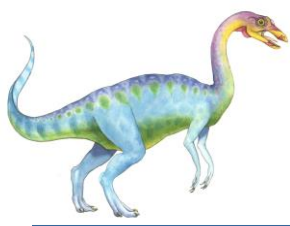


Pipes

- Acts as a conduit allowing two processes to communicate

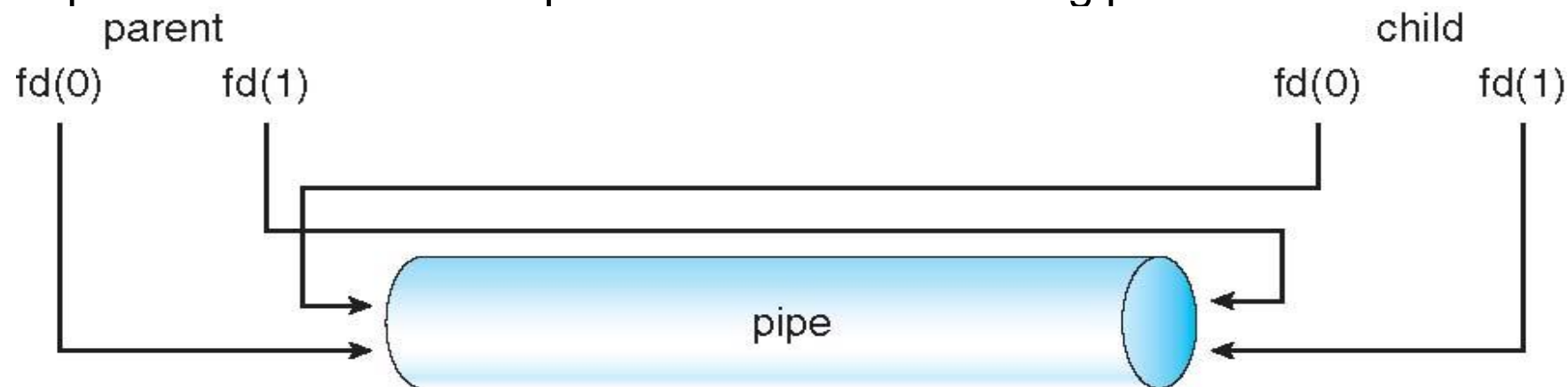
- **Issues**
 - Is communication unidirectional or bidirectional?
 - In the case of two-way communication, is it half or full-duplex?
 - Must there exist a relationship (i.e. **parent-child**) between the communicating processes?
 - Can the pipes be used over a network?



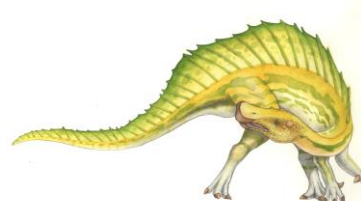


Ordinary Pipes

- 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



- Windows calls these **anonymous pipes**
- See Unix and Windows code samples in textbook





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

