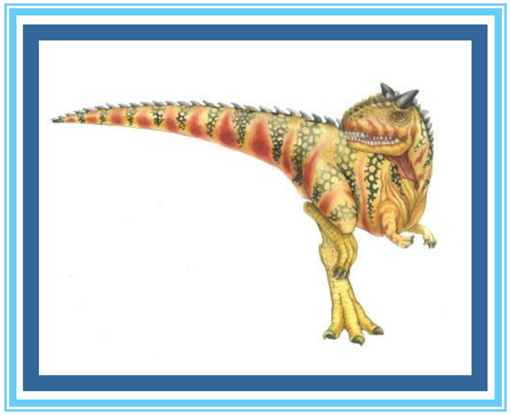
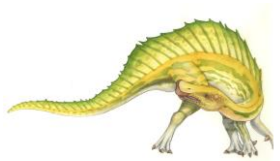
Chapter 3: Processes



Chapter 3: Processes

CO2 : Illustrate the concepts of process management and process scheduling mechanisms employed in Operating Systems. (**Cognitive knowledge: Understand**)

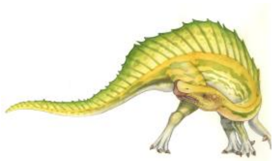
- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing



Processes and programs

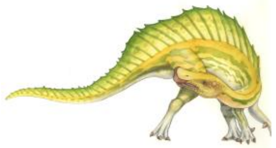
The difference between a process and a program:

- ❑ Baking analogy:
  - Recipe = Program
  - Baker = Processor
  - Ingredients = data
  - Baking the cake = Process
- ❑ Interrupt analogy
  - The baker's son runs in with a wounded hand
  - First aid guide = interrupt code



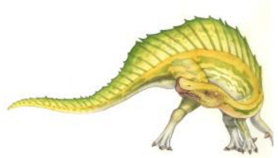
Main OS Process-related Goals

- Interleave the execution of existing processes to maximize processor utilization
- Provide reasonable response times
- Allocate resources to processes
- Support inter-process communication (and synchronization) and user creation of processes



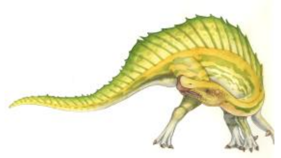
How are these goals achieved?

- Schedule and dispatch processes for execution by the processor
- Implement a safe and fair policy for resource allocation to processes
- Respond to requests by user programs
- Construct and maintain tables for each process managed by the operating system



Process Concept

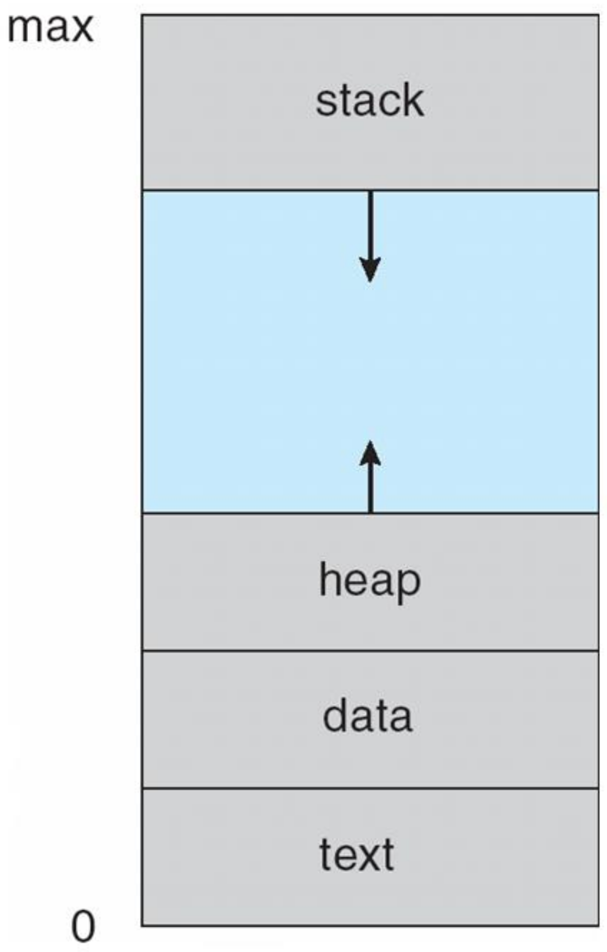
- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential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



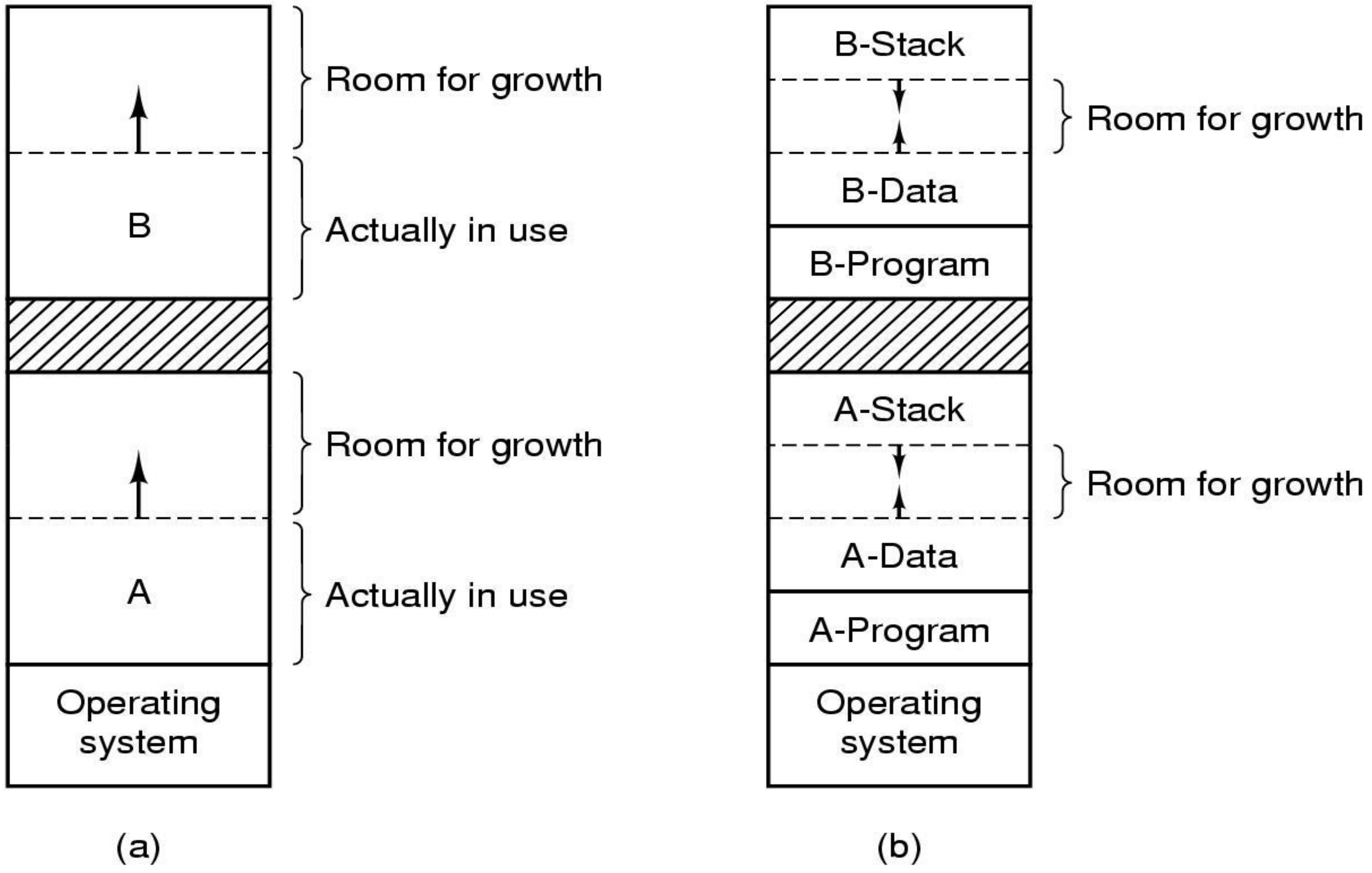
Process Concept

- 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 in Memory



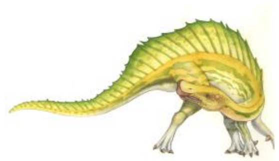
Processes in Memory (2)



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

  - A process can ***block itself***, but not “run” itself



Process State Transitions

When do these transitions occur?

1. Process blocks for input or waits for an event
2. End of time-slice, or preemption
3. Scheduler switches back to this process
4. Input becomes available, event arrives

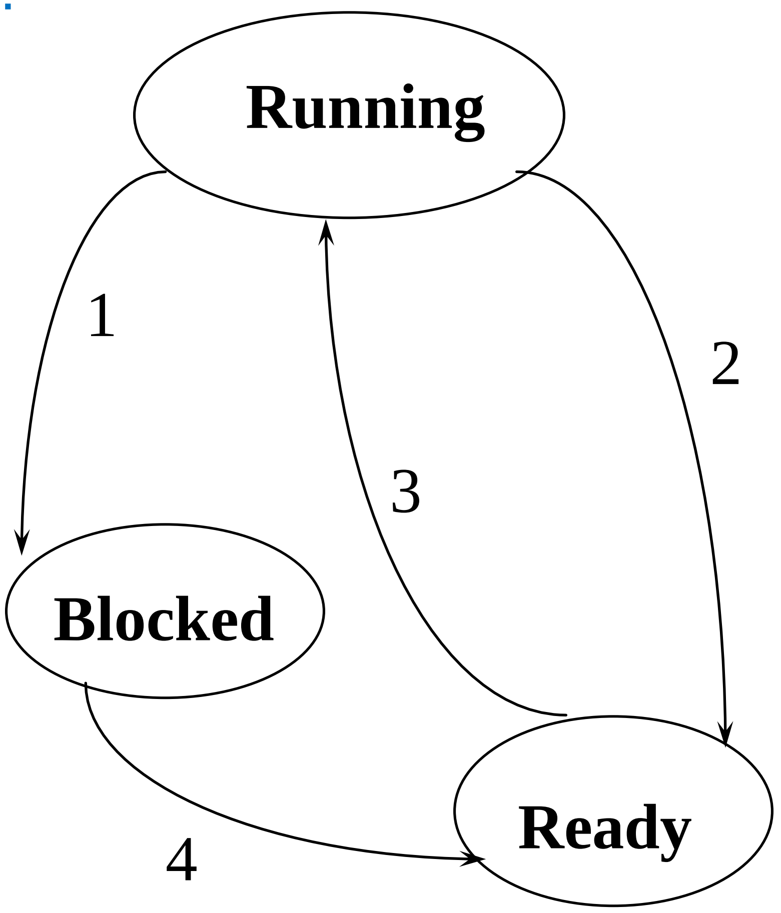
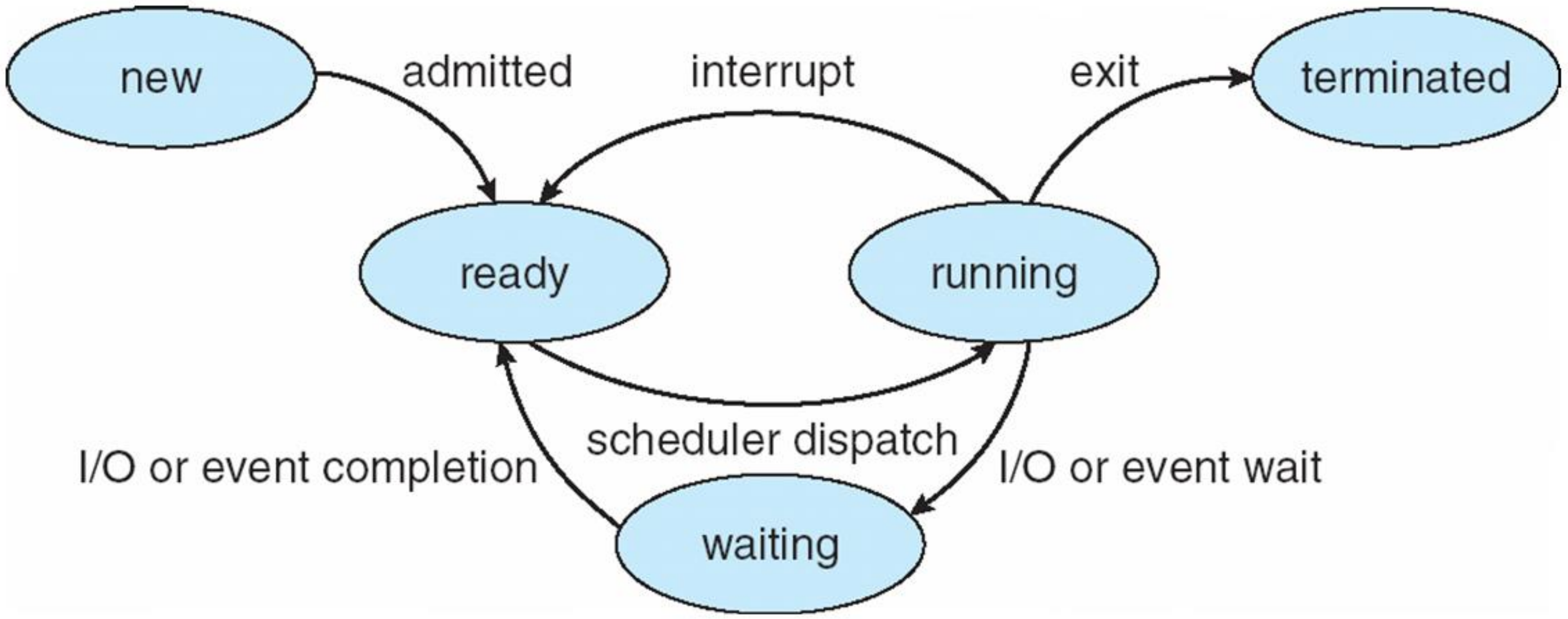
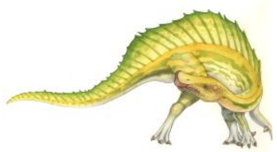


Diagram of Process State



Process Transitions (1)

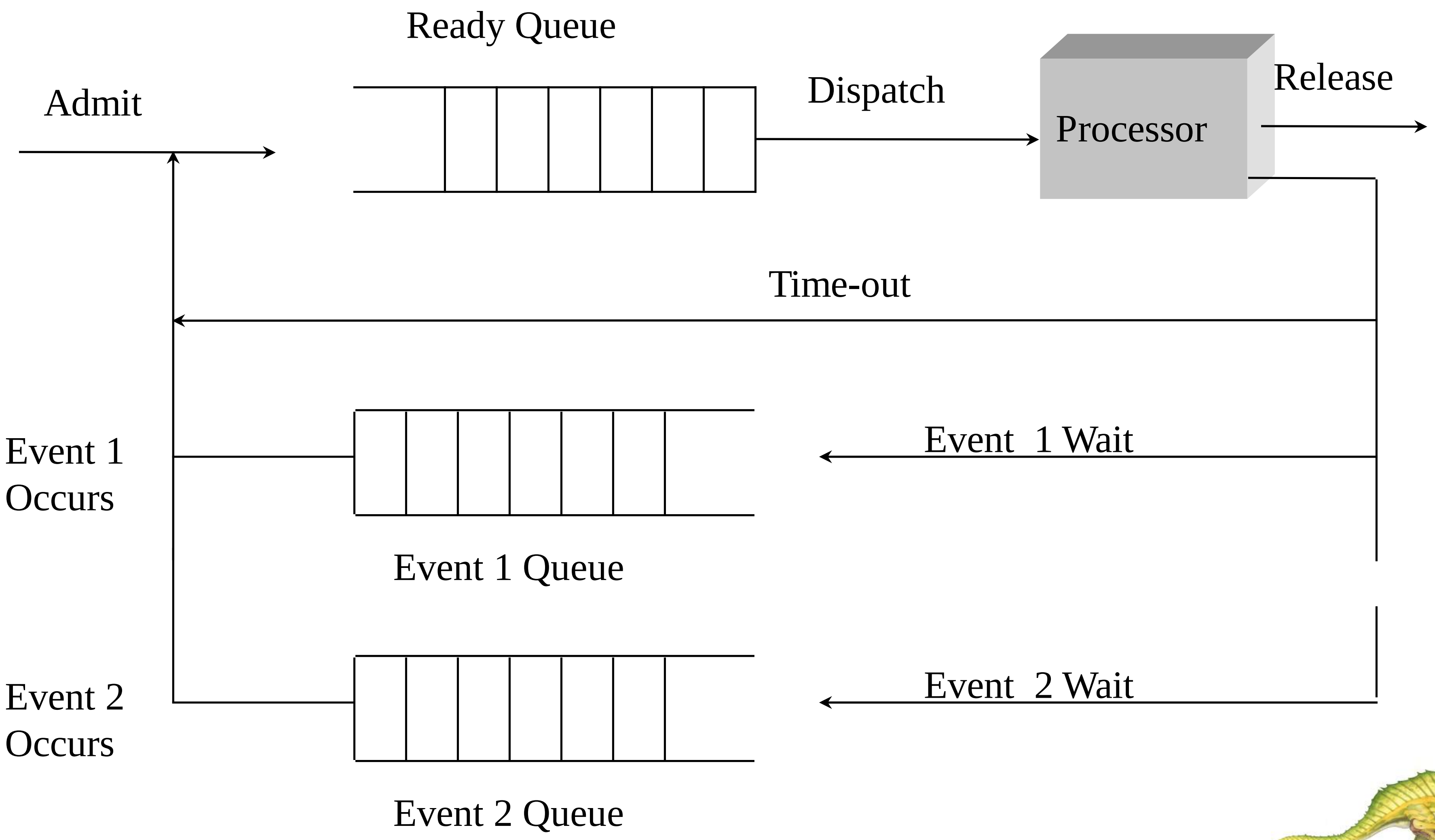
- Ready –> Running
  - When it is time, the dispatcher selects a new process to run.
- Running –> Ready
  - the running process has expired his time slot.
  - the running process gets interrupted because a higher priority process is in the ready state.



Process Transitions (2)

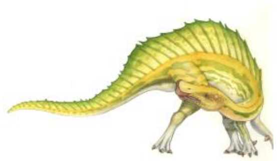
- Running –> Waiting
  - When a process requests something for which it must wait:
    - a service that the OS is not ready to perform.
    - an access to a resource not yet available.
    - initiates I/O and must wait for the result.
    - waiting for a process to provide input.
- Waiting –> Ready
  - When the event for which it was waiting occurs.

Scheduling: Multiple Blocked Queues

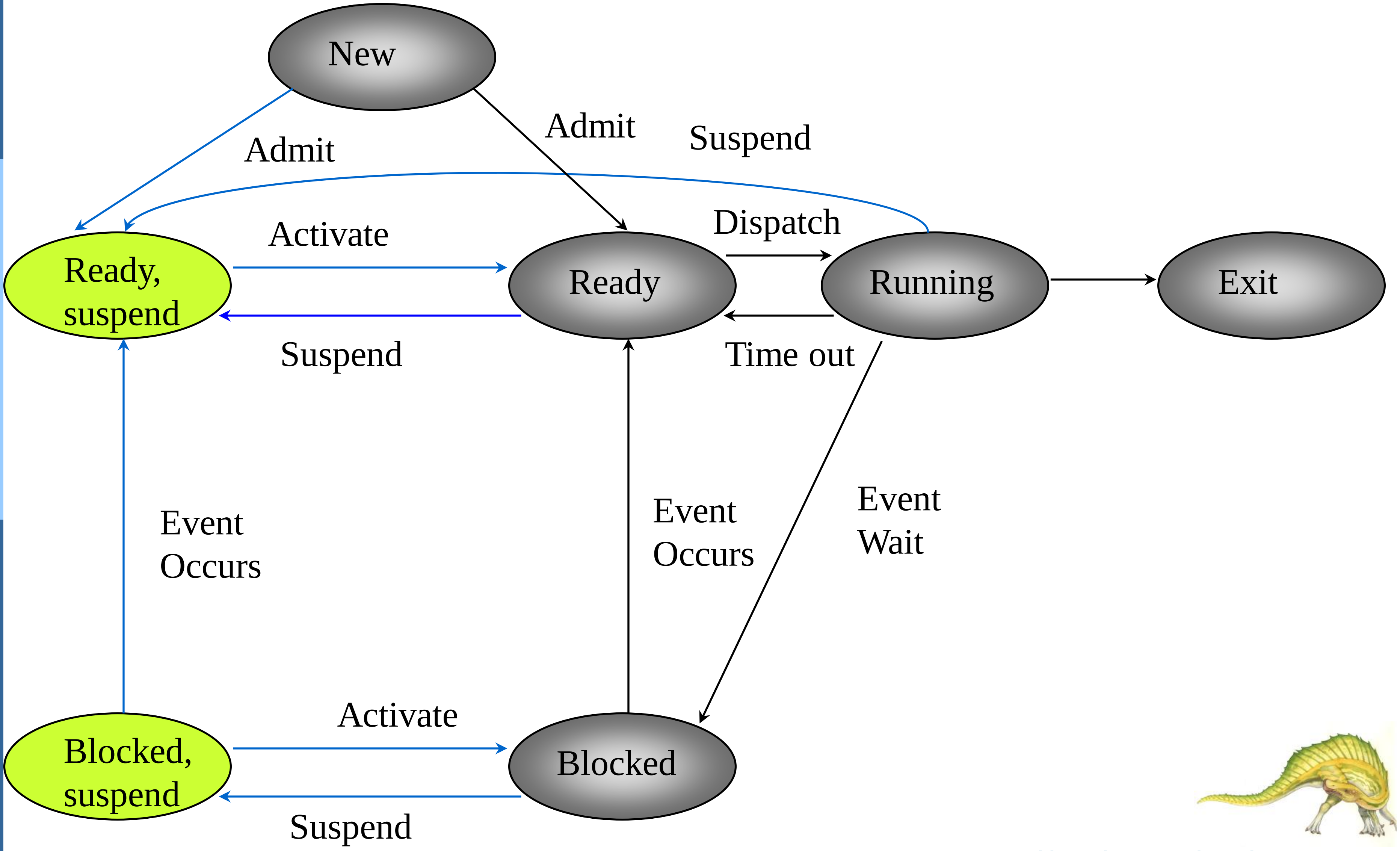


Suspended Processes

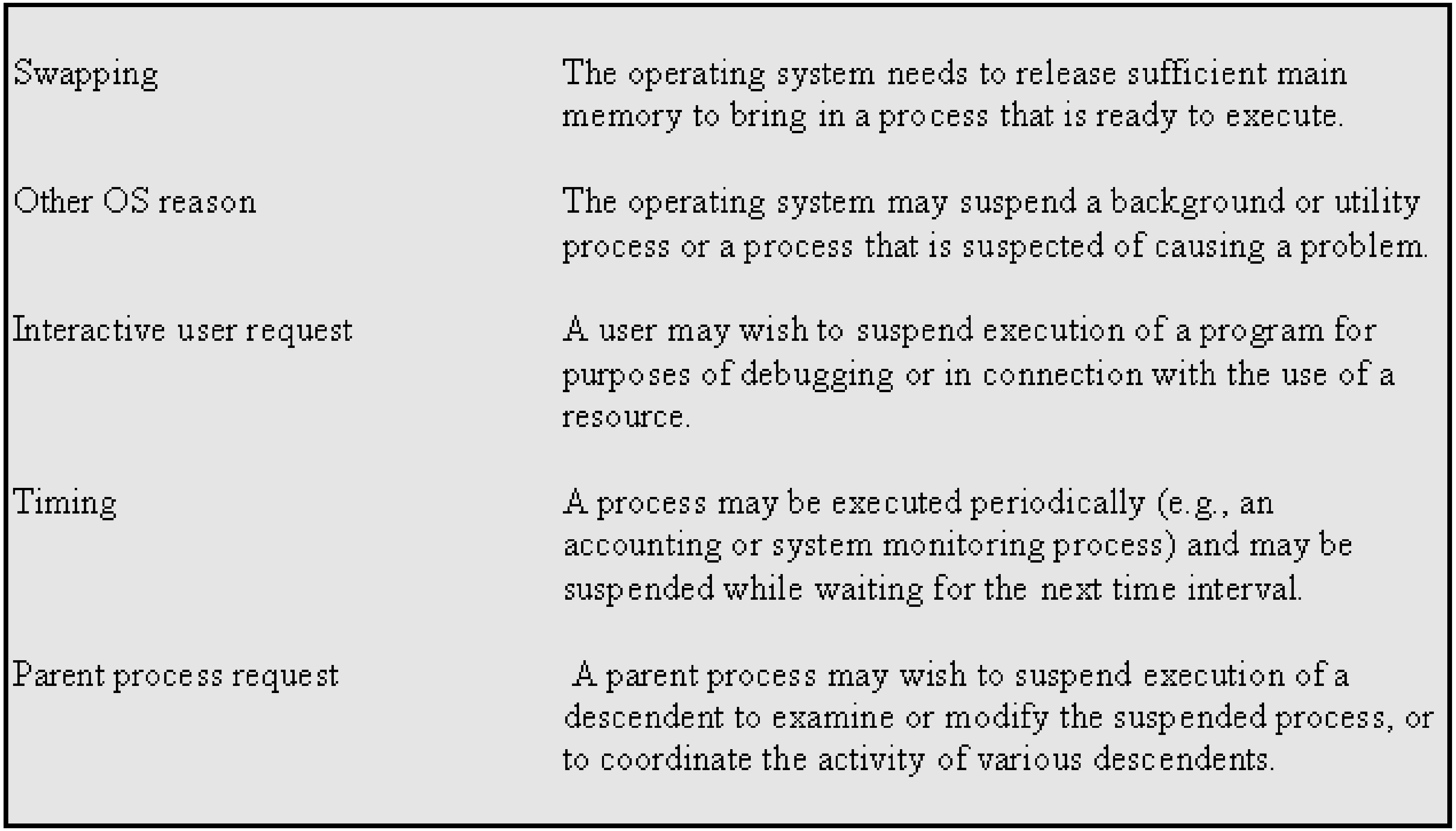
- Processor is much faster than I/O so many processes could be waiting for I/O
- Swap some of these processes to disk to free up more memory
- Blocked state becomes blocked-suspended state when swapped to disk, ready becomes ready - suspended
- Two new states
  - Blocked-suspended
  - Ready-suspended



Process State Transition Diagram with Two Suspend States



Reasons for Process Suspension



Swapping	The operating system needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The operating system may suspend a background or utility process or a process that is suspected of causing a problem.
Interactive user request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.
Parent process request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendents.

Process description

- OS constructs and maintains tables of information about each entity that it is managing : memory tables, IO tables, file tables, process tables.
- **Process control block:** Associated with each process are a number of attributes used by OS for process control. This collection is known as **PCB.**
- Process image: Collection of program, data, stack, and PCB together is known as **Process image**.

Process Control Block (1)

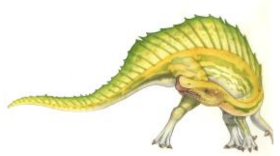
- Process identification
  - Identifiers

    Numeric identifiers that may be stored with the process control block include

    - Identifier of this process
    - Identifier of the process that created this process (parent process)
    - User identifier

Process Control Block (2)

- Processor State Information
  - User-Visible Registers
    - A user-visible register is one that may be referenced by means of the machine language that the processor executes.
    - Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.



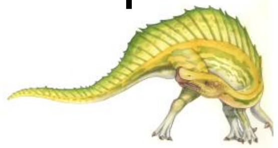
Process Control Block (3)

- Processor State Information
  - Control and Status Registers

    These are a variety of processor registers that are employed to control the operation of the processor. These include

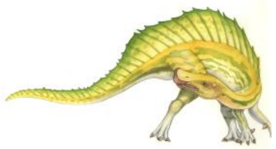
    - •*Program counter:* Contains the address of the next instruction to be fetched
    - •*Condition codes:* Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)

    •*Status information:* Includes interrupt enabled/disabled flags, execution mode



Process Control Block (4)

- Processor State Information
  - Stack Pointers
    - Each process has one or more last-in-first-out (LIFO) system stacks associated with it.
    - A stack is used to store parameters and calling addresses for procedure and system calls.
    - The stack pointer points to the top of the stack.



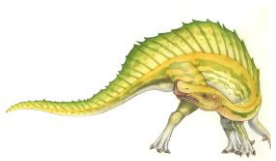
Process Control Block (5)

- Process Control Information
  - Scheduling and State
    - *Process state:* defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).
    - *Priority:* One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable)
    - *Scheduling-related information:* This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.

    •*Event:* Identity of event the process is awaiting before it can be resumed

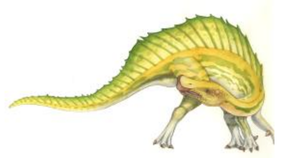
Process Control Block (6)

- Process Control Information
  - Data Structuring
    - A process may be linked to other process in a queue, ring, or some other structure.
    - For example, all processes in a waiting state for a particular priority level may be linked in a queue.
    - A process may exhibit a parent-child (creator-created) relationship with another process.
    - The process control block may contain pointers to other processes to support these structures.



Process Control Block (7)

- Process Control Information
  - Interprocess Communication
    - Various flags, signals, and messages may be associated with communication between two independent processes.
  - Process Privileges
    - Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed.
    - In addition, privileges may apply to the use of system utilities and services.



Process Control Block (8)

- Process Control Information
  - Memory Management
    - This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.
  - Resource Ownership and Utilization
    - Resources controlled by the process may be indicated, such as opened files.
    - A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

Process Control Block (9)

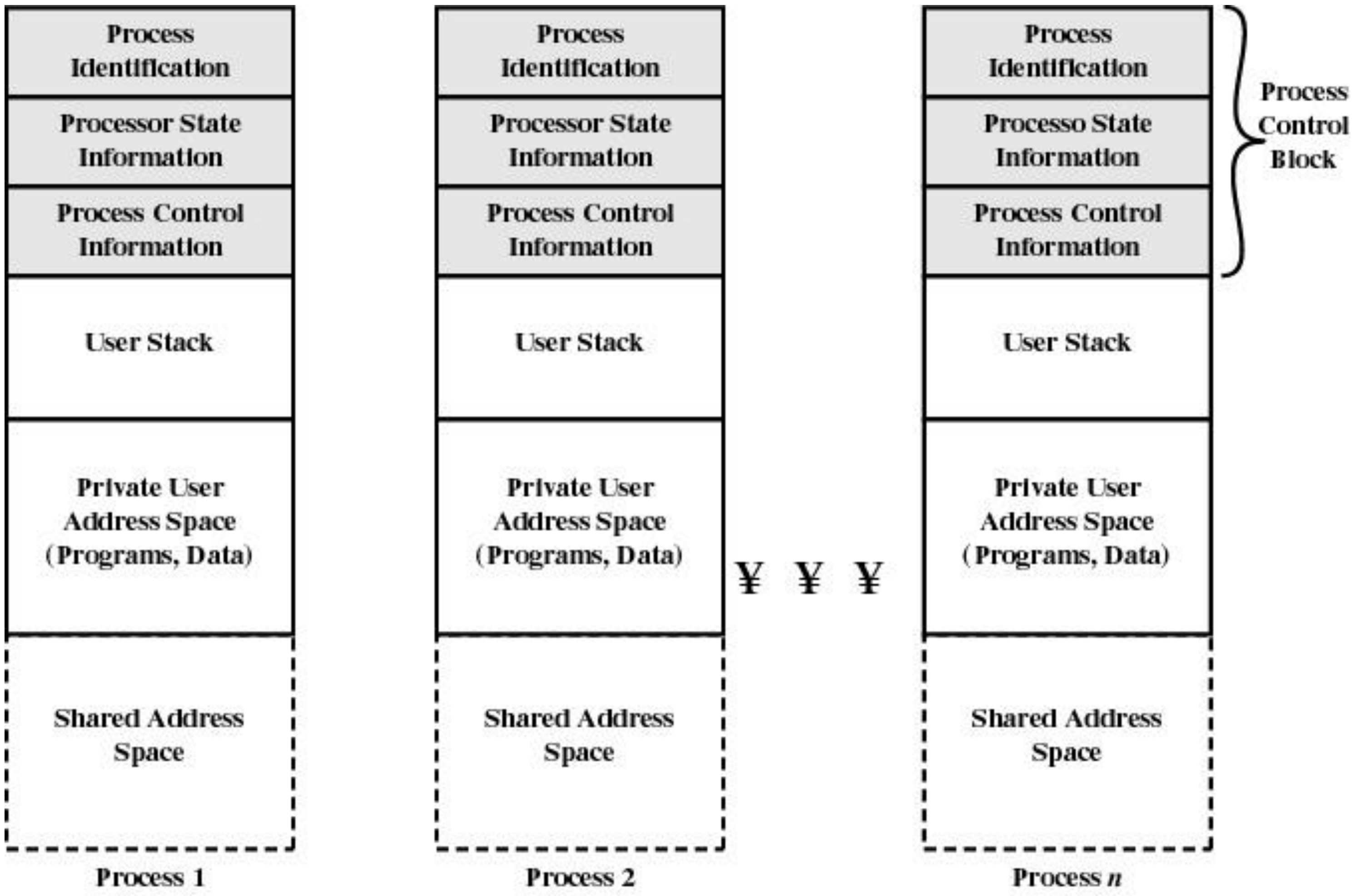

Process Identification
Processor State Information
Process Control Information
User Stack
Private User Address Space (Programs, Data)
Shared Address Space
Process 1
Process 2
Processo State Information
Process n
Process Control Block
¥ ¥ ¥

Figure 3.12 User Processes in Virtual Memory

Process control block

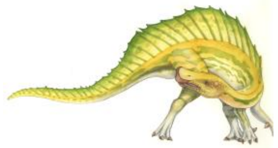
- Contains three categories of information:

1) Process identification

2) Process state information

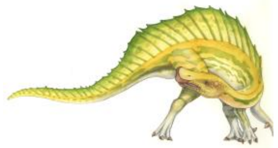
3) Process control information

- **Process identification**:
  - numeric identifier for the process (pid)
  - identifier of the parent (ppid)
  - user identifier (uid) - id of the usr responsible for the process.



Process control block (contd.)

- **Process state information:**
  - User visible registers
  - Control and status registers : PC, IR, PSW, interrupt related bits, execution mode.
  - Stack pointers

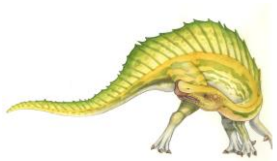


Process control block (contd.)

- **Process control information:**
  - Scheduling and state information : Process state, priority, scheduling-related info., event awaited.
  - Data structuring : pointers to other processes (PCBs): belong to the same queue, parent of process, child of process or some other relationship.
  - Interprocess comm: Various flags, signals, messages may be maintained in PCBs.

Process control block (contd.)

- Process control information (contd.)
  - Process privileges: access privileges to certain memory area, critical structures etc.
  - Memory management: pointer to the various memory management data structures.
  - Resource ownership : Pointer to resources such as opened files. Info may be used by scheduler.
- PCBs need to be protected from inadvertent destruction by any routine. So protection of PCBs is a critical issue in the design of an OS.

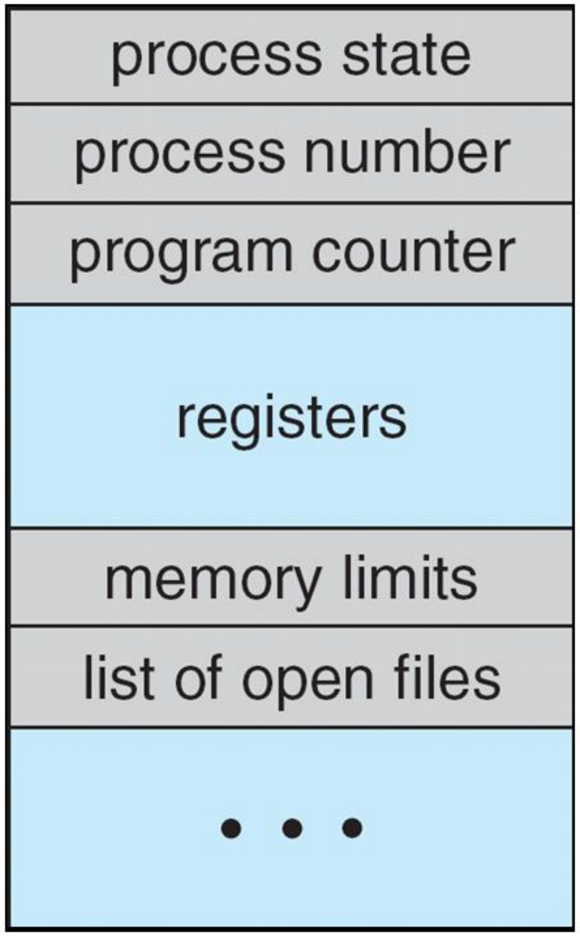


Process Control Block (PCB)

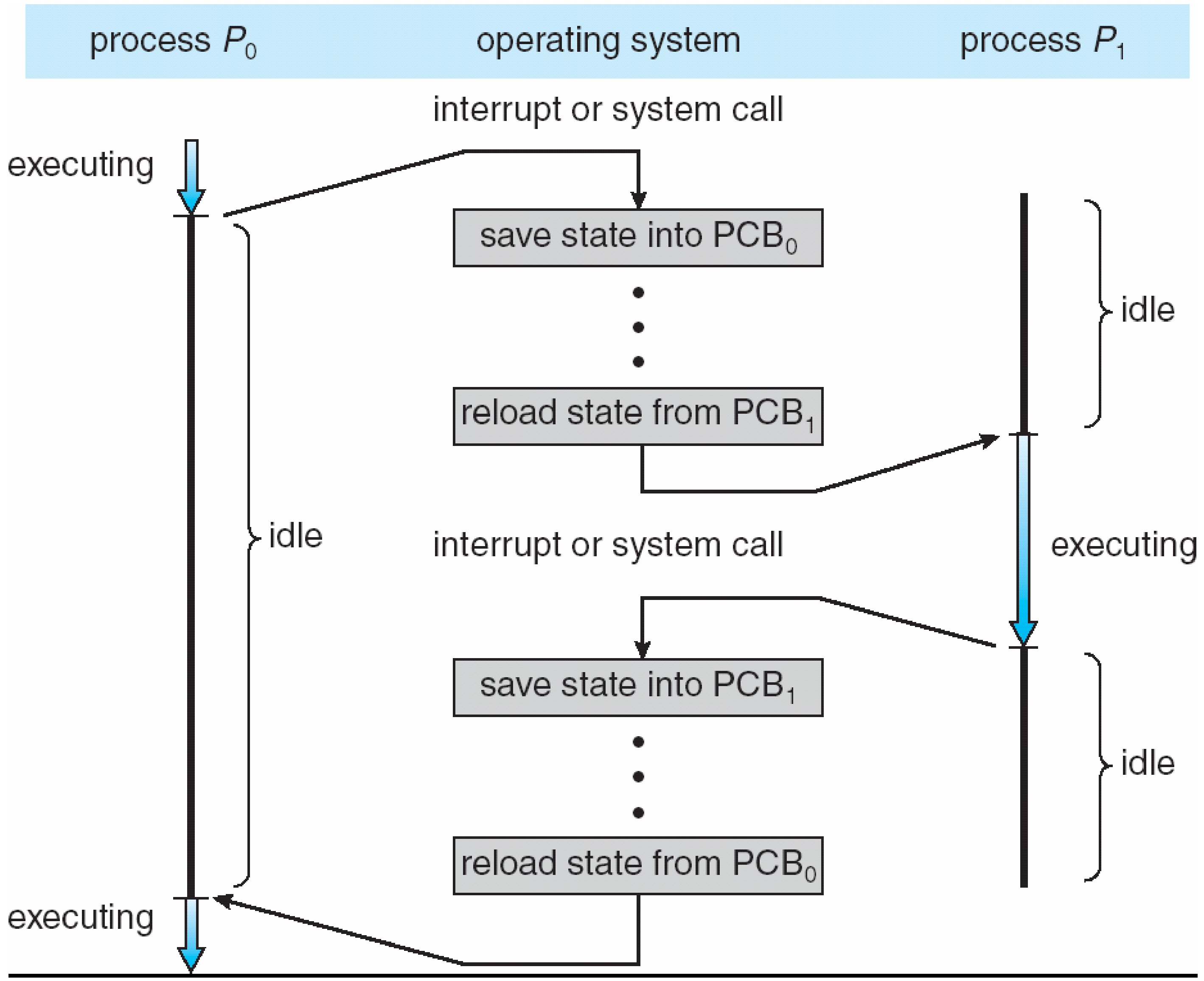
Information associated with each process

(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

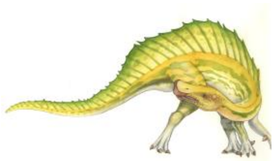


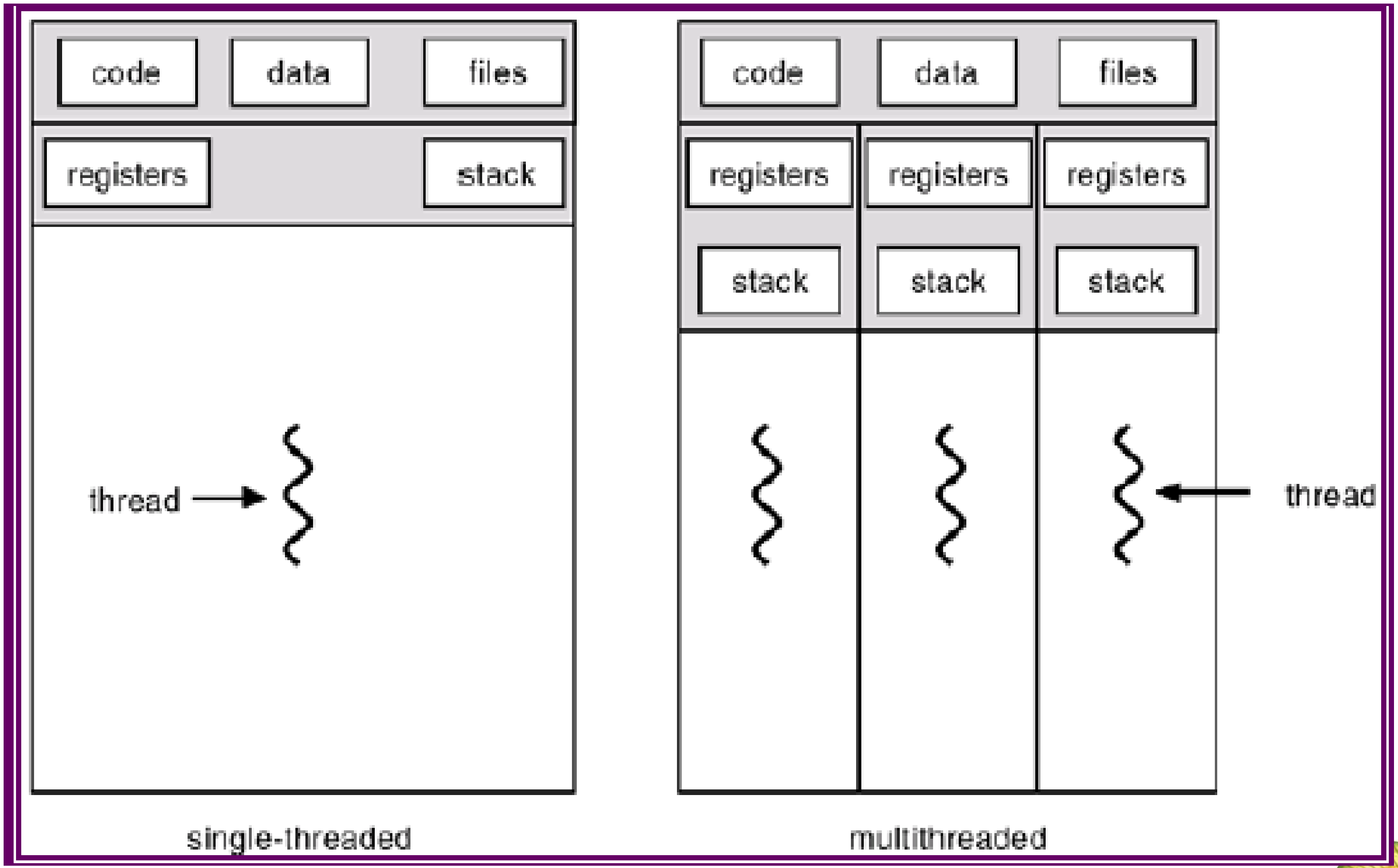
Threads

- A thread is **the smallest unit of processing that can be performed in an OS**.
- In most modern operating systems, a thread exists within a process - that is, a single process may contain multiple threads
- A thread is a path of execution within a process
- A thread is also known as lightweight process.
- The idea is to achieve parallelism by dividing a process into multiple threads.
- For example, in a browser, multiple tabs can be different threads.
- MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc.

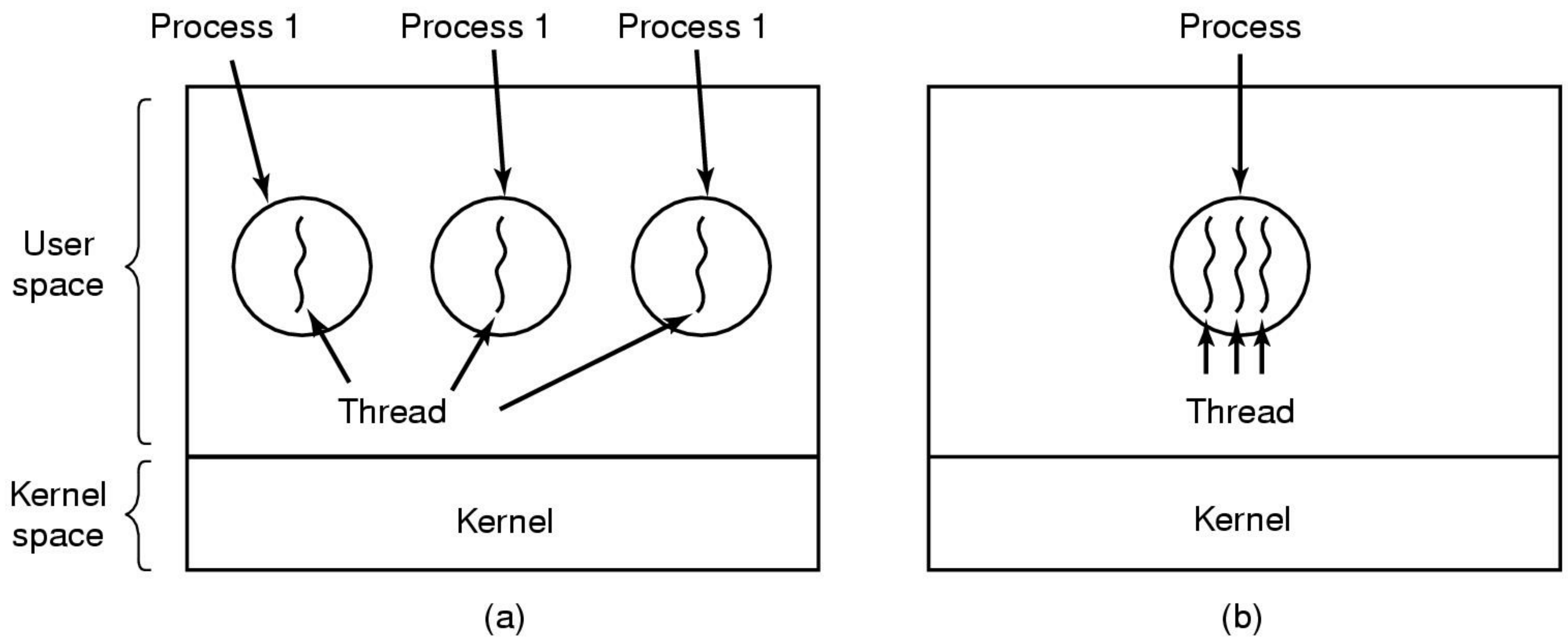
Threads

- In *traditional* OS, each process has
  - An address space
  - A single thread of control
- In *modern* OS, each process may have
  - Multiple threads of control
  - Same address spare
  - Running in quasi-parallel
- Thread or a Lightweight Process has
  - a program counter, registers, stack




code
data
files
registers
stack
thread
single-threaded
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded

The Thread Model (1)

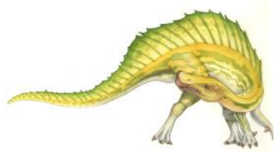


(a) Three processes each with one thread

(b) One process with three threads

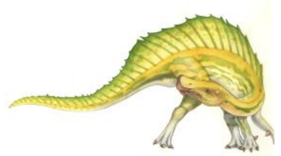
Concurrency vs Parallelism

- Concurrency : Multiple tasks start, run and complete in overlapping time periods, in no specific order
- Parallelism : Multiple tasks or subtasks of the same tasks run at the same time on a hardware with multiple computing resources like a multi core processor

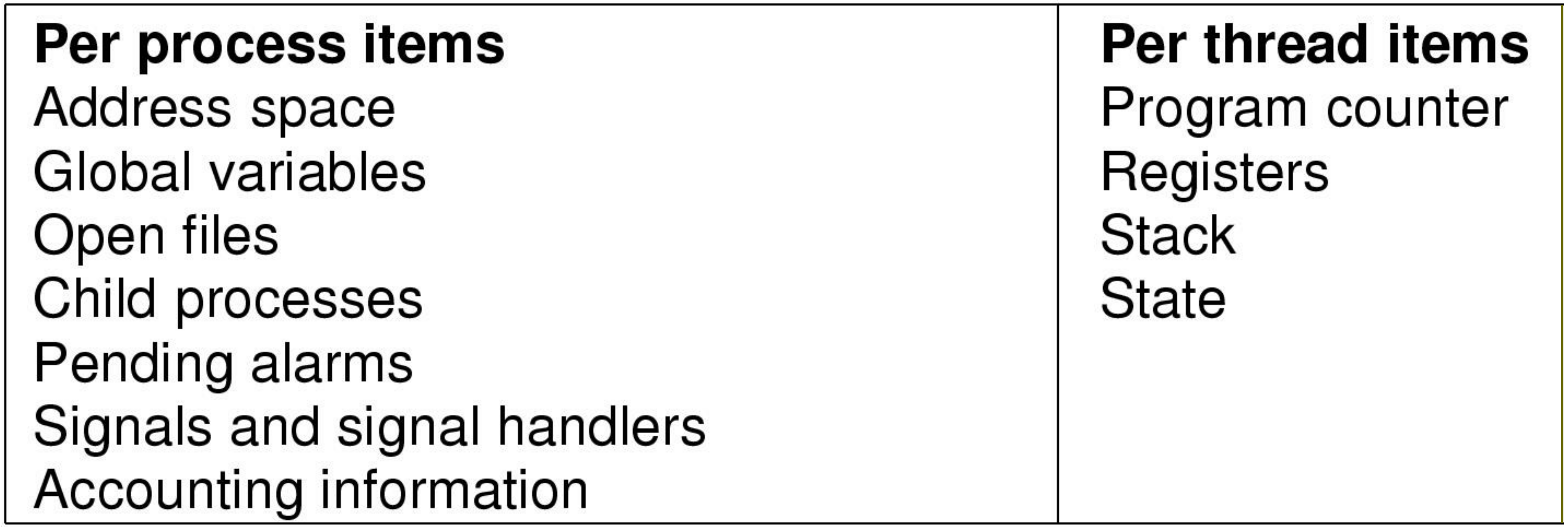


Process vs Thread?

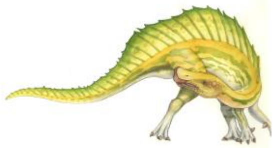
- The primary difference is that threads within the same process run in a **shared memory space**, while processes run in separate memory spaces.
- Threads are **not independent** of one another like processes are, and as a result threads share with other threads their code section, data section, and OS resources (like open files and signals).
- But, like process, a thread has its own program counter (PC), register set, and stack space.



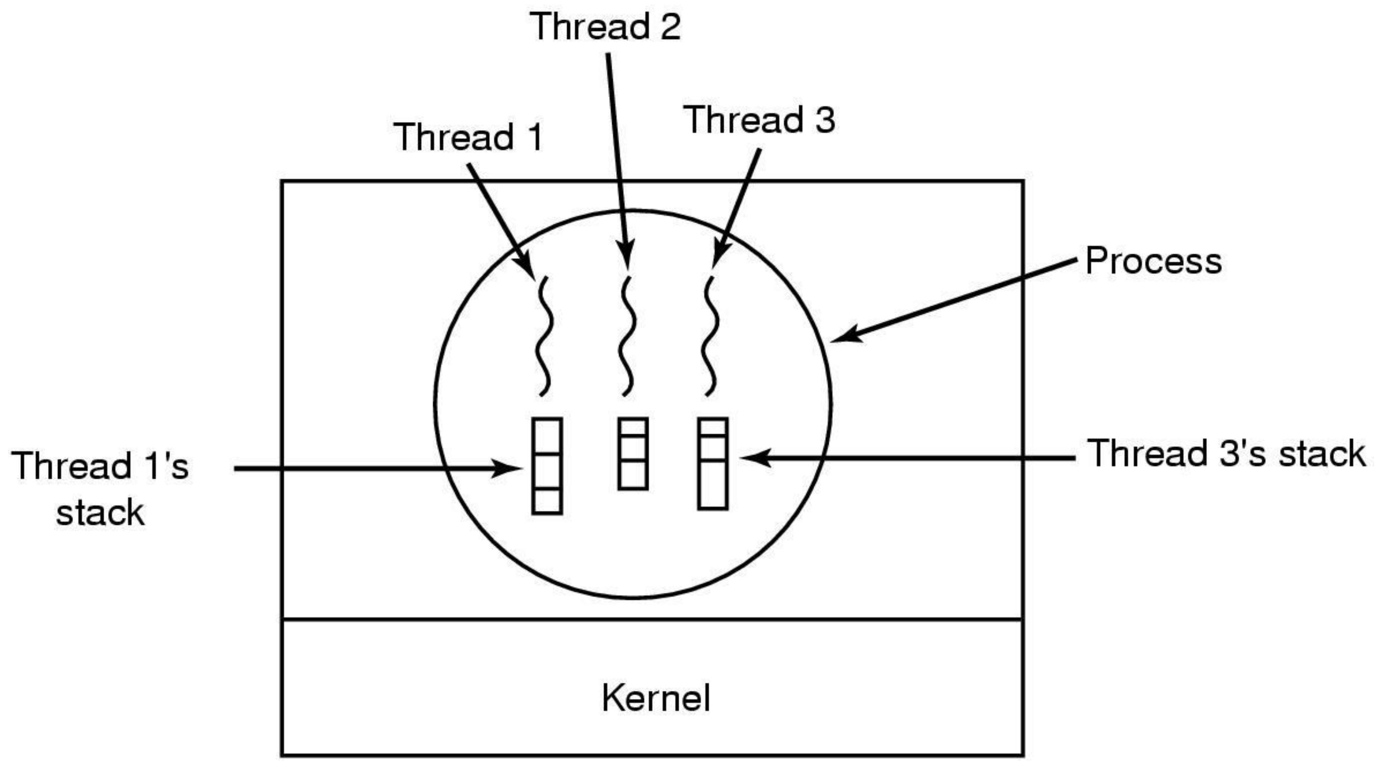
The Thread Model (2)



Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	



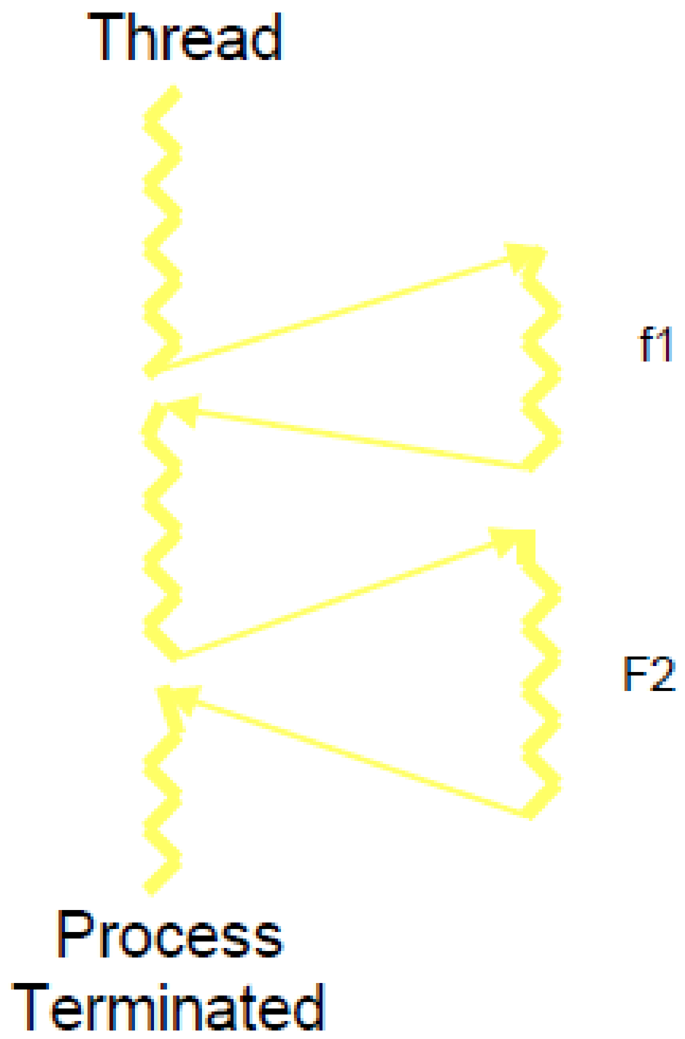
The Thread Model (3)



Each thread has its own stack

Code structure of a single-threaded (sequential) process

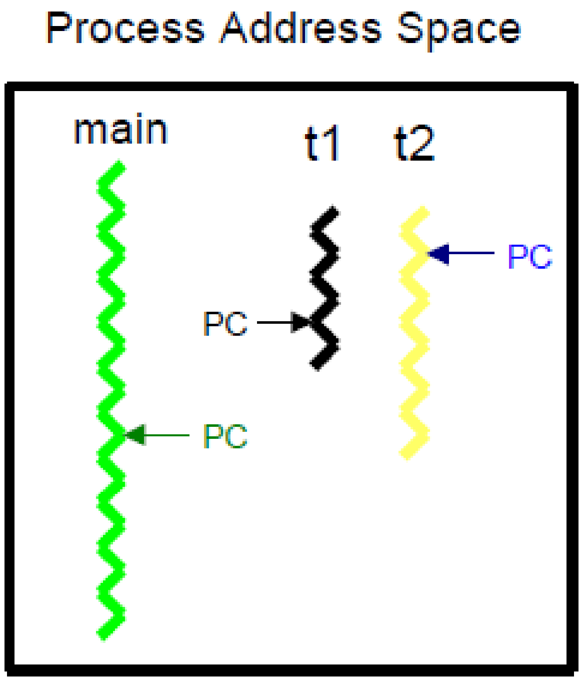
```
main()
{


        f1(

        f2(...);
        ...
}
f1(...)
{  ...  }
f2(...)
{  ...  }
```



Code structure of a multi-threaded process

- Hypothetical function thread() to create a thread - takes two arguments: the name of a function for which a thread has to be created and a variable in which the ID of the newly created thread is to be stored.
- The important point to note here is that multiple threads of control are simultaneously active within the same process; each thread steered by its own PC.

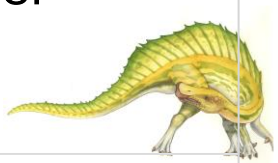
```
main()
{
        ...
        thread(t1,f1);
        ...
        thread(t2,f2);
        ...
}
f1(...)
{  ...  }
f2(...)
{  ...  }
```



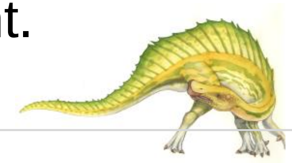
Why use Threads?

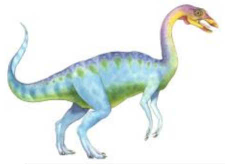
- Large multiprocessors need many computing entities (one per CPU)
- Switching between processes incurs high overhead
- With threads, an application can avoid per-process overheads
  - Thread creation, deletion, switching cheaper than processes
- Threads have full access to address space (easy sharing)
- Threads can execute in parallel on multiprocessors

Basis	Process	Thread
Definition	A process is a program under execution i.e an active program. Process means any program is in execution	A thread is a lightweight process that can be managed independently by a scheduler. Thread means a segment of a process.
Context switching time	Processes require more time for context switching as they are more heavy.	Threads require less time for context switching as they are lighter than processes.
Memory Sharing	Processes are totally independent and don't share memory.	A thread may share some memory with its peer threads.

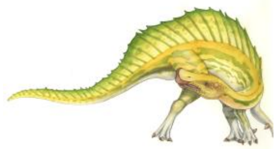


Basis	Process	Thread
Communication	Communication between processes requires more time than between threads. The process is less efficient in terms of communication	Communication between threads requires less time than between processes . The thread is more efficient in terms of communication
Blocked	If a process gets blocked, remaining processes can continue execution.	If a user level thread gets blocked, all of its peer threads also get blocked.
Resource Consumption	Processes require more resources than threads.	Threads generally need less resources than processes.
Dependency	Individual processes are independent of each other.	Threads are parts of a process and so are dependent.

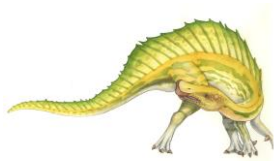




Basis	Process	Thread
Data and Code sharing	Processes have independent data and code segments.	A thread shares the data segment, code segment, files etc. with its peer threads.
Treatment by OS	All the different processes are treated separately by the operating system.	All user level peer threads are treated as a single task by the operating system.
Time for creation	Processes require more time for creation.	Threads require less time for creation.
Time for termination	Processes require more time for termination.	Threads require less time for termination.



Basis	Process	Thread
Parent-child	Changes to the parent process do not affect child processes.	Since all threads of the same process share address space and other resources so any changes to the main thread may affect the behavior of the other threads of the process.
Switching	Process switching uses an interface in an operating system.	Thread switching does not require calling an operating system and causes an interrupt to the kernel.



Advantages of Thread

- ***Responsiveness:*** If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
  - Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- ***Faster context switch:*** Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.
- ***Effective utilization of multiprocessor system:*** If we have multiple threads in a single process, then we can schedule multiple threads on multiple processor. This will make process execution faster.

Advantages of Thread

- ***Resource sharing:*** Resources like code, data, and files can be shared among all threads within a process.
  - By default, threads share the memory and the resources of the process to which they belong.
  - Code sharing allows an application to have several different threads of activity all within the same address space.
- ***Economy***: Allocating memory and resources for process creation is costly. Alternatively, because threads share resources of the process to which they belong, it is more economical to create and context switch threads

Main disadvantages of threads

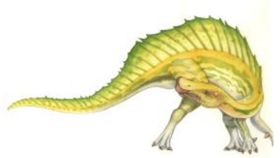
- ***Resource sharing:*** Whereas resource sharing is one of the major advantages of threads, it is also a disadvantage because proper synchronization is needed between threads for accessing the shared resources (e.g., data and files).
- ***Difficult programming model***: It is difficult to write, debug, and maintain multithreaded programs for an average user.
  - This is particularly true when it comes to writing code for synchronized access to shared resources.

Similarity between Threads and Processes –

- Only one thread or process is active at a time
- Within process both execute sequential
- Both can create children

Differences between Threads and Processes –

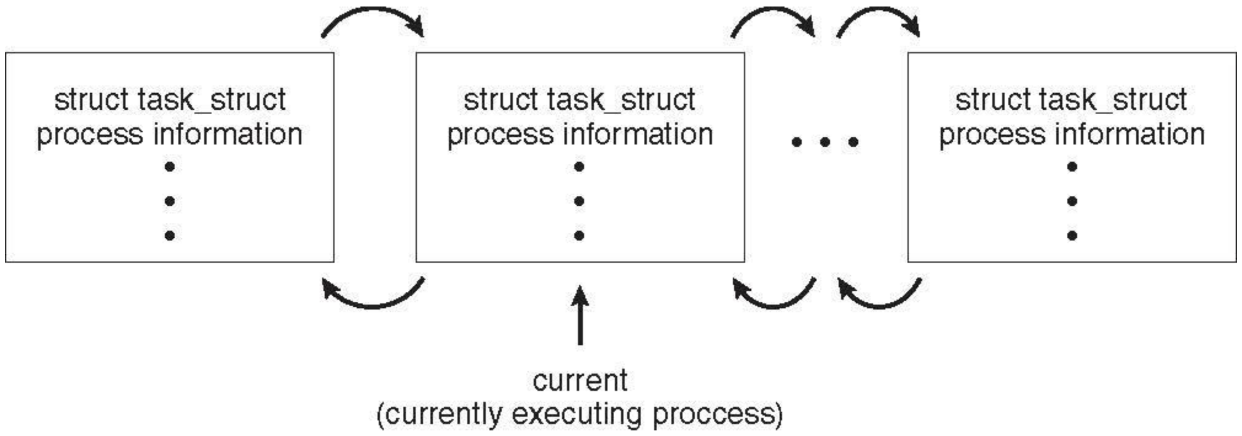
- Threads are not independent, processes are.
- Threads are designed to assist each other, processes may or may not do it



Process Representation in Linux

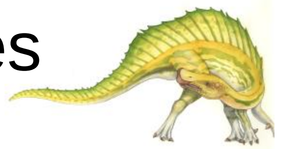
Represented by the C structure `task_struct`

```
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 */
```

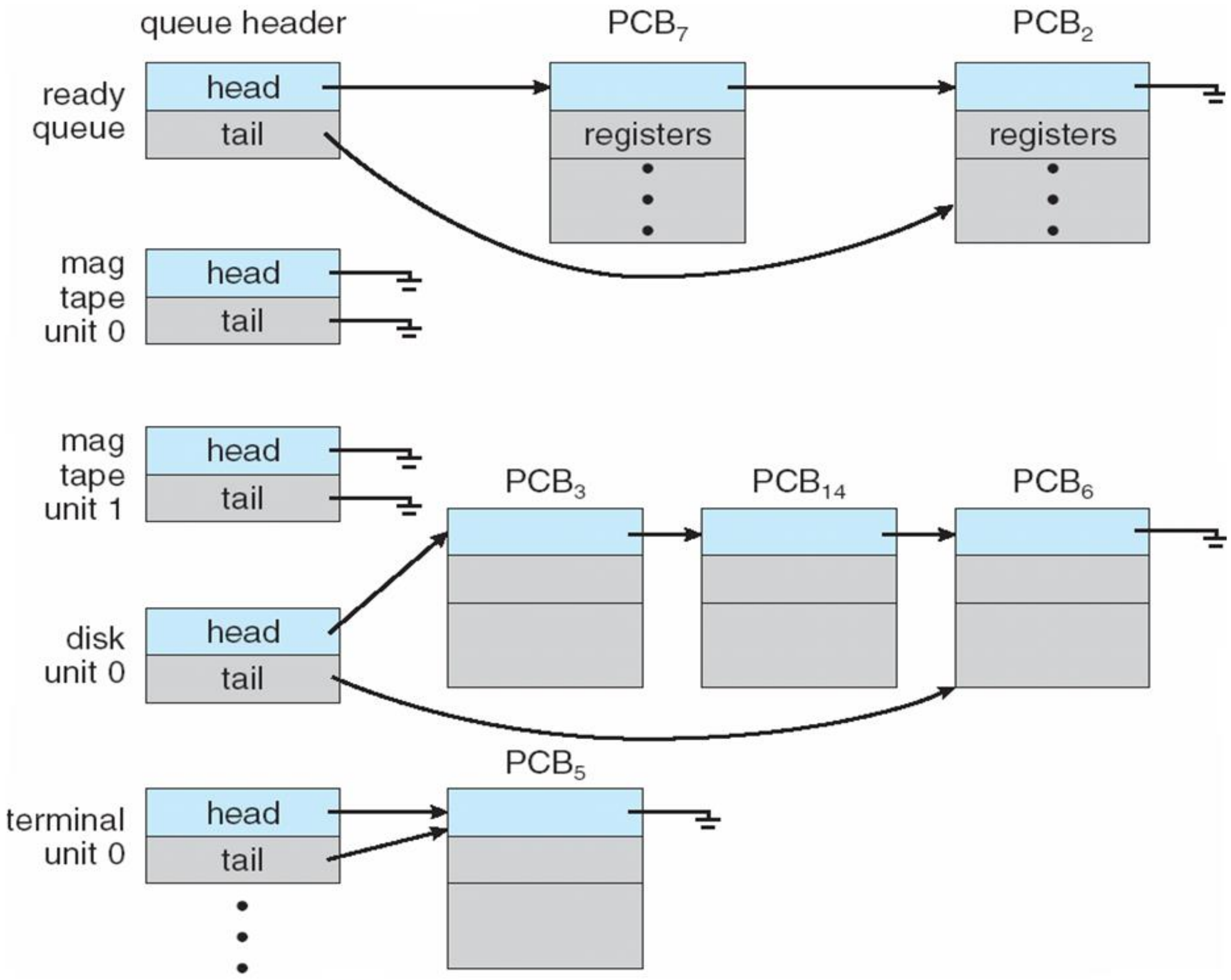


Process Scheduling

- 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

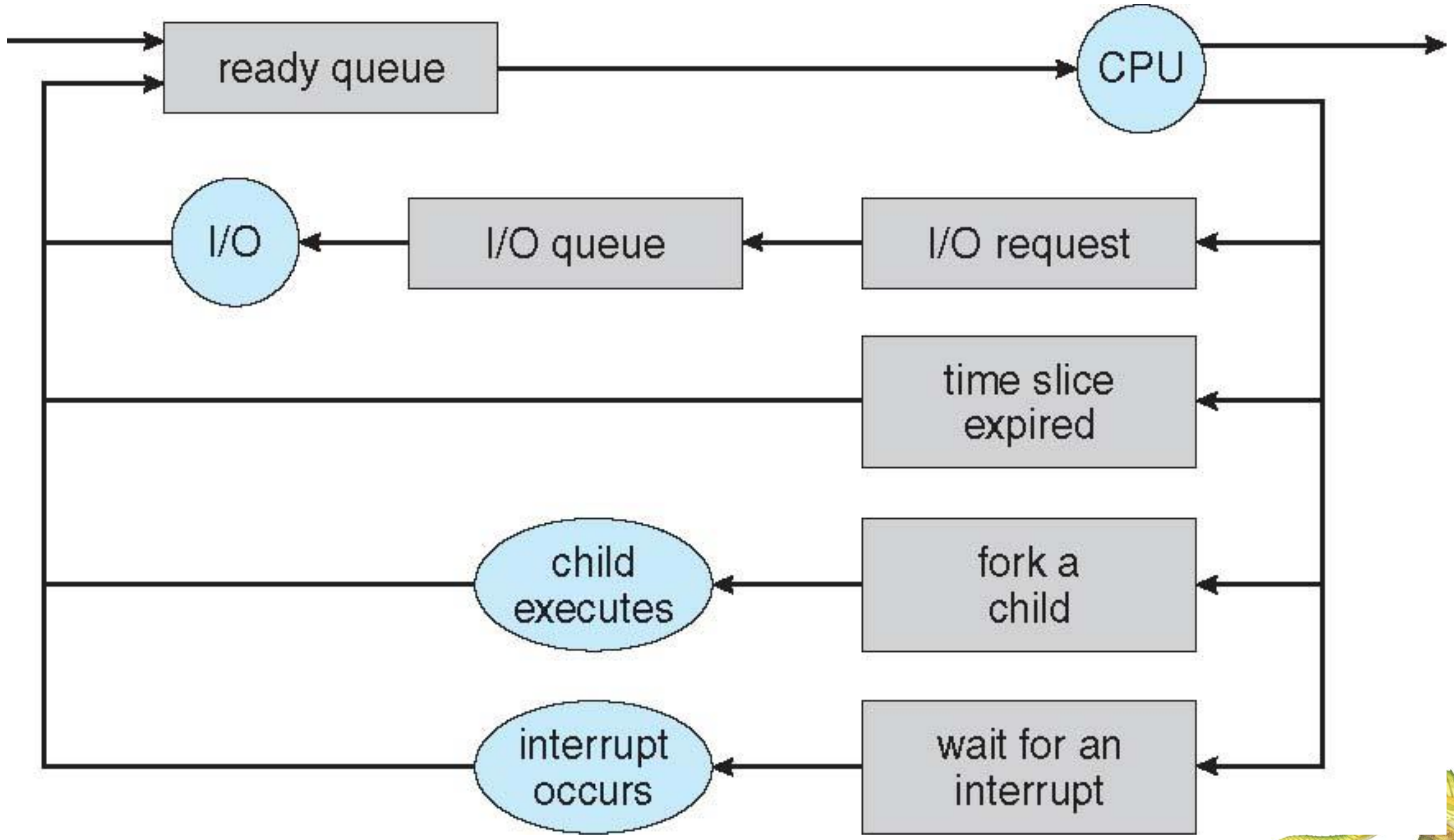


Ready Queue And Various I/O Device Queues



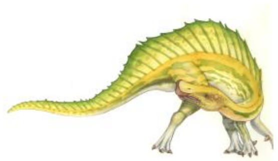
Representation of Process Scheduling

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



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)
- Goal: Efficiently allocate the CPU to one of the ready processes according to some criteria.

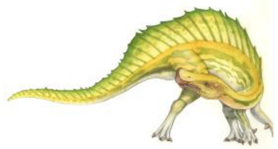


Schedulers

- **Long-term scheduler** (or **job scheduler**) – selects which processes 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**

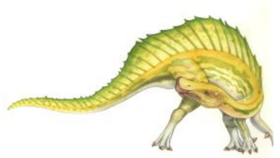
Remarks:

- Control the degree of multiprogramming
- Can take more time in selecting processes because of a longer interval between executions
- May not exist physically



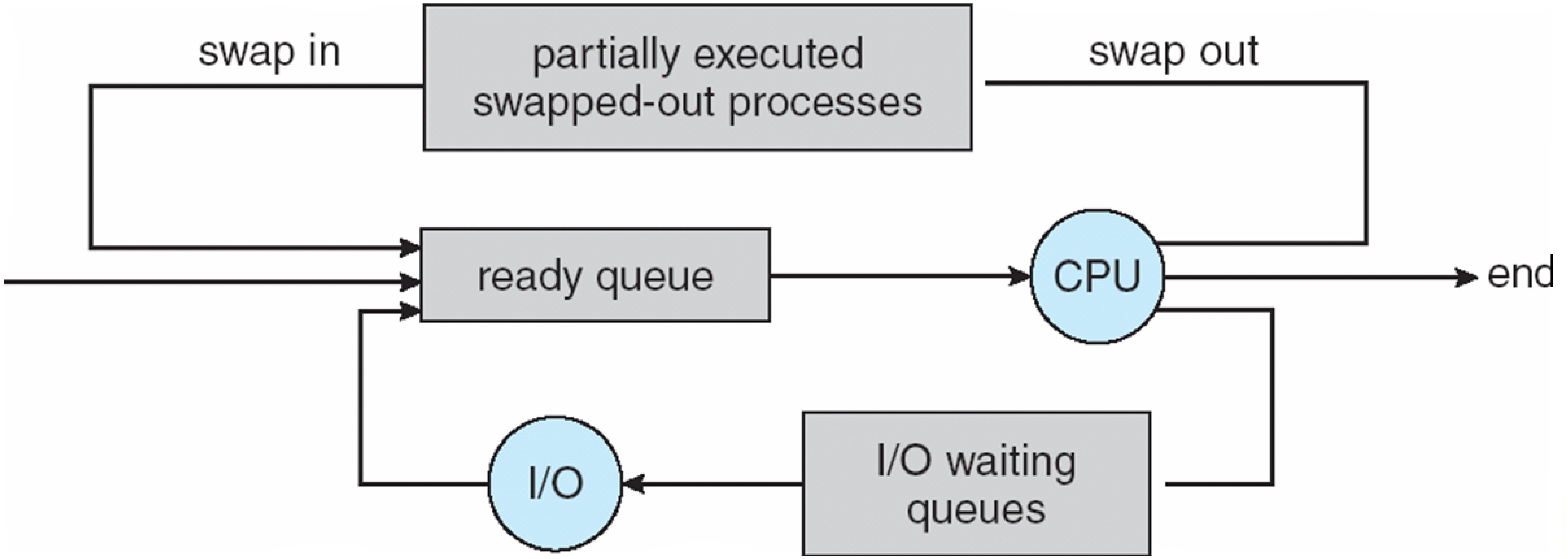
Schedulers

- 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***
  - Goal: Select a good mix of I/O-bound and CPU-bound process



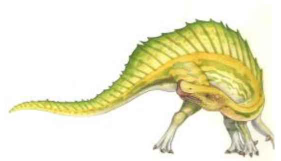
Addition of Medium Term Scheduling

- **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 ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once



- The kernel is activated when an *event*, which is a situation that requires the kernel’s attention, leads to either a hardware interrupt or a system call
- The kernel performs four fundamental functions to control operation of processes
  - ***Context save:*** Saving CPU state and information concerning resources of the process whose operation is interrupted.
  - ***Event handling:*** Analyzing the condition that led to an interrupt, or the request by a process that led to a system call, and taking appropriate actions.
  - ***Scheduling:*** Selecting the process to be executed next on the CPU.
  - ***Dispatching:*** Setting up access to resources of the scheduled process and loading its saved CPU state in the CPU to begin or resume its operation.

Context Save, Scheduling, and Dispatching

- An OS contains two processes *P*1 and *P*2, with *P*2 having a higher priority than *P*1. Let *P*2 be *blocked* on an I/O operation and let *P*1 be *running*. The following actions take place when the I/O completion event occurs for the I/O operation of *P*2:
  - The **context save** function is performed for *P*1 and its state is changed to *ready*.
  - Using the ***event information*** field of PCBs, the event handler finds that the I/O operation was initiated by *P*2, so it changes the state of *P*2 from *blocked* to *ready*.
  - **Scheduling** is performed. *P*2 is selected because it is the highest-priority *ready* process.
  - *P*2's state is changed to *running* and it is **dispatched**.

Event Handling

The following events occur during the operation of an OS:

1. *Process creation event:* A new process is created.

2. *Process termination event:* A process completes its operation.

3. *Timer event:* The timer interrupt occurs.

4. *Resource request event:* Process makes a resource request.

5. *Resource release event:* A process releases a resource.

6. *I/O initiation request event:* Process wishes to initiate an I/O operation.

7. *I/O completion event:* An I/O operation completes.

8. *Message send event:* A message is sent by one process to another.

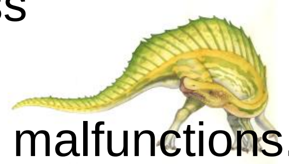
9. *Message receive event:* A message is received by a process.

10. *Signal send event:* A signal is sent by one process to another.

11. *Signal receive event:* A signal is received by a process.

12. *A program interrupt:* The current instruction in the *running* process malfunctions.

13. *A hardware malfunction event:* A unit in the computer's hardware malfunctions.



- The timer, I/O completion, and hardware malfunction events are caused by situations that are external to the running process.
- All other events are caused by actions in the *running* process.
- The kernel performs a standard action like aborting the *running* process when events 12 or 13 (*program interrupt, hardware malfunction event)* occur.

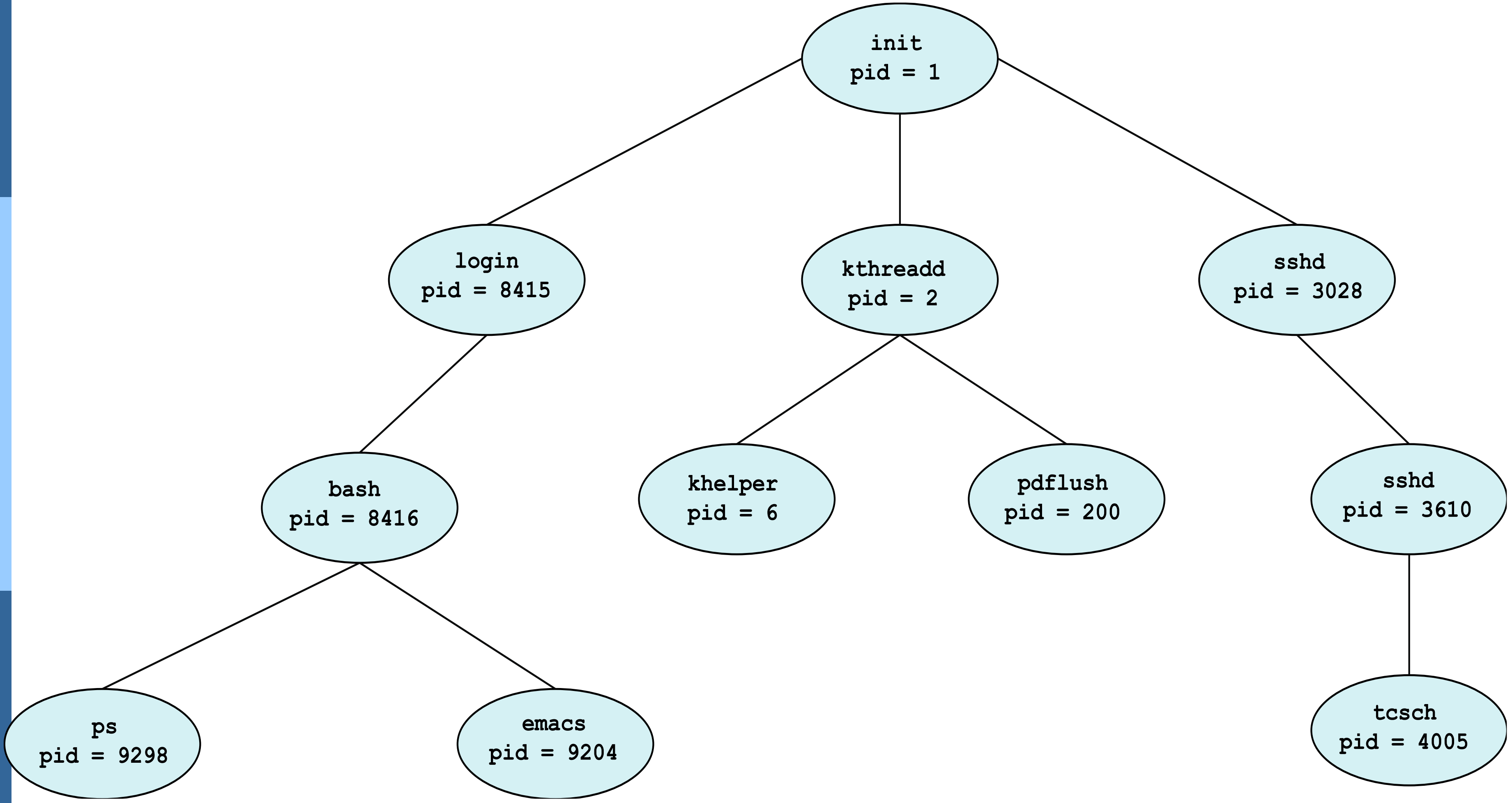
Operations on Processes

- System must provide mechanisms for:
  - Process creation,
  - Process termination

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

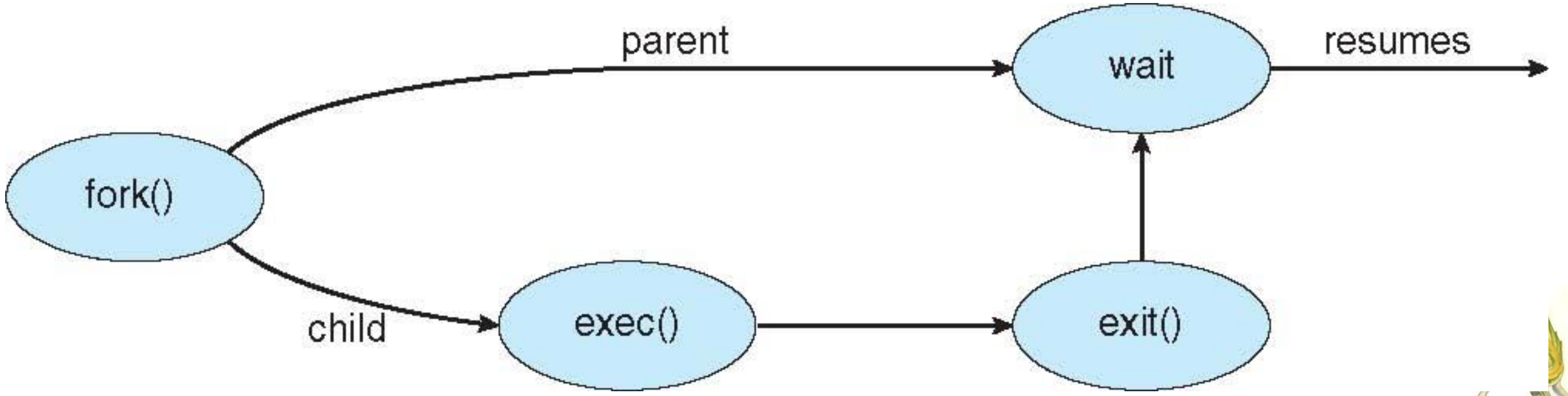
A Tree of Processes in Linux

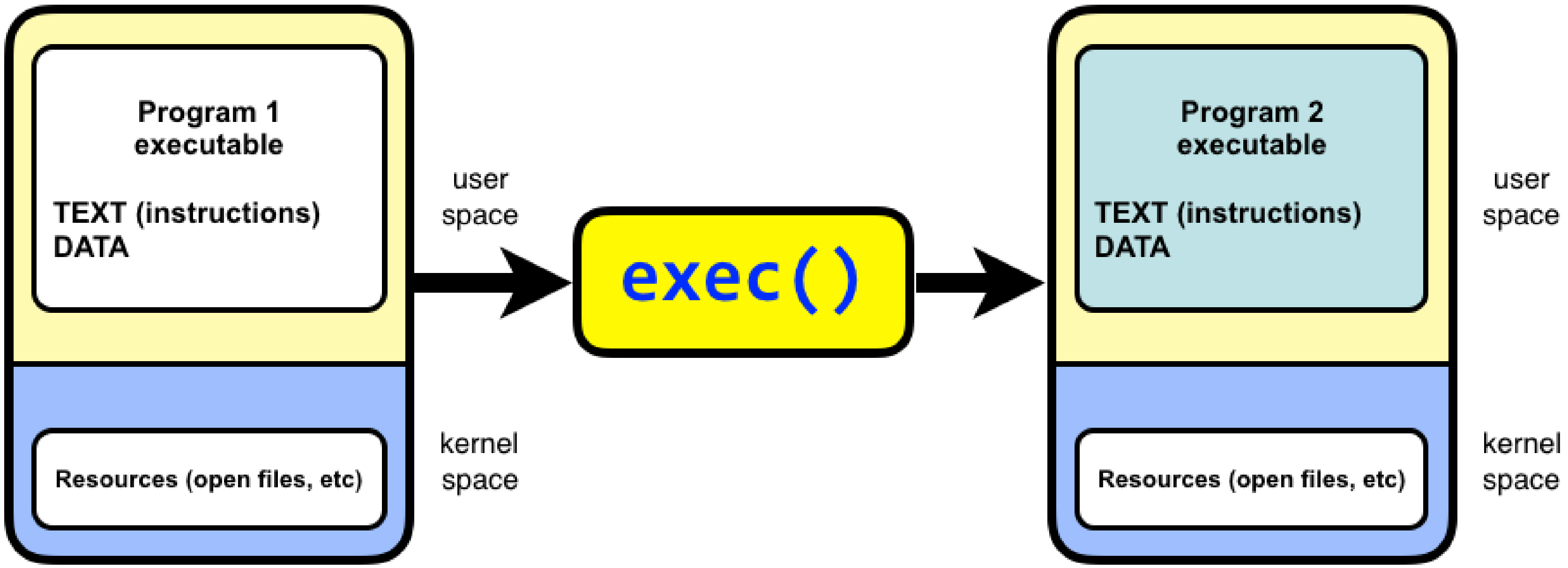


- Fork system call is used for creating a new process, which is called ***child process***, which runs concurrently with the process that makes the fork() call (parent process).
- After a new child process is created, both processes will execute the next instruction following the fork() system call.
- A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

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 exec family of system calls replaces the program executed by a process.
- When a process calls exec, all code (text) and data in the process is lost and replaced with the executable of the new program.
- Although all data is replaced, all open file descriptors remains open after calling exec unless explicitly set to close-on-exec.

Fork system call

- It takes no parameters and returns an integer value. Below are different values returned by fork().
  - ***Negative Value***: creation of a child process was unsuccessful.
  - ***Zero***: Returned to the newly created child process.
  - ***Positive value***: Returned to parent or caller. The value contains process ID of newly created child process.

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;
}
```

Creating a Separate Process via Windows API

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

   /* allocate memory */
   ZeroMemory(&si, sizeof(si));
   si.cb = sizeof(si);
   ZeroMemory(&pi, sizeof(pi));

   /* create child process */
   if (!CreateProcess(NULL, /* use command line */
    "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
    NULL, /* don't inherit process handle */
    NULL, /* don't inherit thread handle */
    FALSE, /* disable handle inheritance */
    0, /* no creation flags */
    NULL, /* use parent's environment block */
    NULL, /* use parent's existing directory */
    &si,
    &pi))
   {
     fprintf(stderr, "Create Process Failed");
     return -1;
   }
   /* parent will wait for the child to complete */
   WaitForSingleObject(pi.hProcess, INFINITE);
   printf("Child Complete");

   /* close handles */
   CloseHandle(pi.hProcess);
   CloseHandle(pi.hThread);
}
```

Calculate number of times hello is printed:

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

```
fork ();   // Line 1
fork ();   // Line 2
fork ();   // Line 3
```

```
      L1                    There will be 1 child process created by line 1.
     /   \
   L2     L2                There will be 2 child processes  created by line 2
   / \      / \
L3  L3  L3  L3               There will be 4 child processes  created by line 3
```

- The main process: P0
  Processes created by the 1st fork: P1
  Processes created by the 2nd fork: P2, P3
  Processes created by the 3rd fork: P4, P5, P6, P7

```
            P0
        /    |    \
     P1    P4   P2
   /   \           \
 P3    P6         P5
 /
P7
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
    if (fork() == 0)
        printf("Hello from Child!\n");
    else
        printf("Hello from Parent!\n");
}
int main()
{
    forkexample();
    return 0;
}
```

```
void forkexample()
{
    int x = 1;


    if (fork() == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
}
int main()
{
    forkexample();
    return 0;
}
```

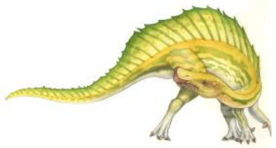
Output:

```
Parent has x = 0
Child has x = 2
     (or)
Child has x = 2
Parent has x = 0
```

Here, global variable change in one process does not affected two other processes because data/state of two processes are different.

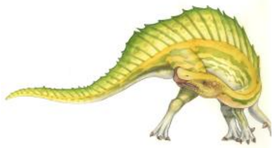
And also parent and child run simultaneously so two outputs are possible.

- The child and parent processes reside on different memory spaces.
- These memory spaces have same content and whatever operation is performed by one process will not affect the other process.
- When the child processes is created; now both the processes will have the same Program Counter (PC), so both of these processes will point to the same next instruction.
- The files opened by the parent process will be the same for child process.



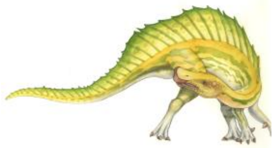
Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system



Process Termination

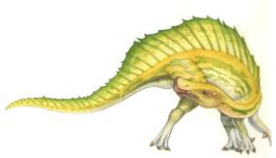
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates



Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

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

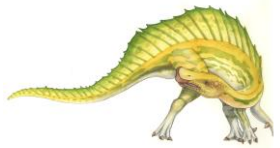


Zombie state

- When a process is created in UNIX using fork() system call, the address space of the Parent process is replicated.
- If the parent process calls wait() system call, then the execution of the parent is suspended until the child is terminated.
- At the termination of the child, a ‘SIGCHLD’ signal is generated which is delivered to the parent by the kernel.
- Parent, on receipt of ‘SIGCHLD’ reads the status of the child from the process table.

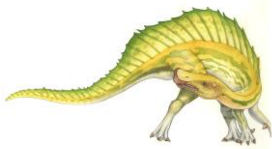
Zombie state

- Even though the child is terminated, there is an entry in the process table corresponding to the child where the status is stored.
- When the parent collects the status, this entry is deleted. Thus, all the traces of the child process are removed from the system.
- If the parent decides not to wait for the child’s termination and executes its subsequent task, then at the termination of the child, the exit status is not read.
- Hence, there remains an entry in the process table even after the termination of the child. This state of the child process is known as the Zombie state.



Process Termination

- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
- A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process.
- A child process always first becomes a zombie before being removed from the process table.
- The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

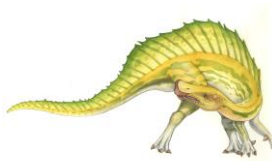


```
int main()
{
    pid_t child_pid = fork();
   // Parent process
    if (child_pid > 0)
        sleep(50);
      // Child process
    else
        exit(0);
    return 0;
}
```

The child finishes its execution using exit() system call while the parent sleeps for 50 seconds, hence doesn't call wait() and the child process's entry still exists in the process table.

Why do we need to prevent the creation of the Zombie process?

- There is one process table per system.
- The size of the process table is finite.
- If too many zombie processes are generated, then the process table will be full.
- That is, the system will not be able to generate any new process, then the system will come to a standstill.
- Hence, we need to prevent the creation of zombie processes.



- Creation of Zombie can be Prevented Using wait() system call
- When the parent process calls wait(), after the creation of a child, it indicates that, it will wait for the child to complete and it will reap the exit status of the child.
- The parent process is suspended(waits in a waiting queue) until the child is terminated.
- It must be understood that during this period, the parent process does nothing just wait.

Process Termination

- If parent terminated without invoking `wait`, process is an **orphan**
- A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.
- However, the orphan process is soon adopted by init process, once its parent process dies.

```
int main()
{
        int pid = fork();
        if (pid > 0)
                printf("in parent process");
        else if (pid == 0)
        {
                sleep(30);
                printf("in child process");
        }

        return 0;
}
```

Interprocess Communication

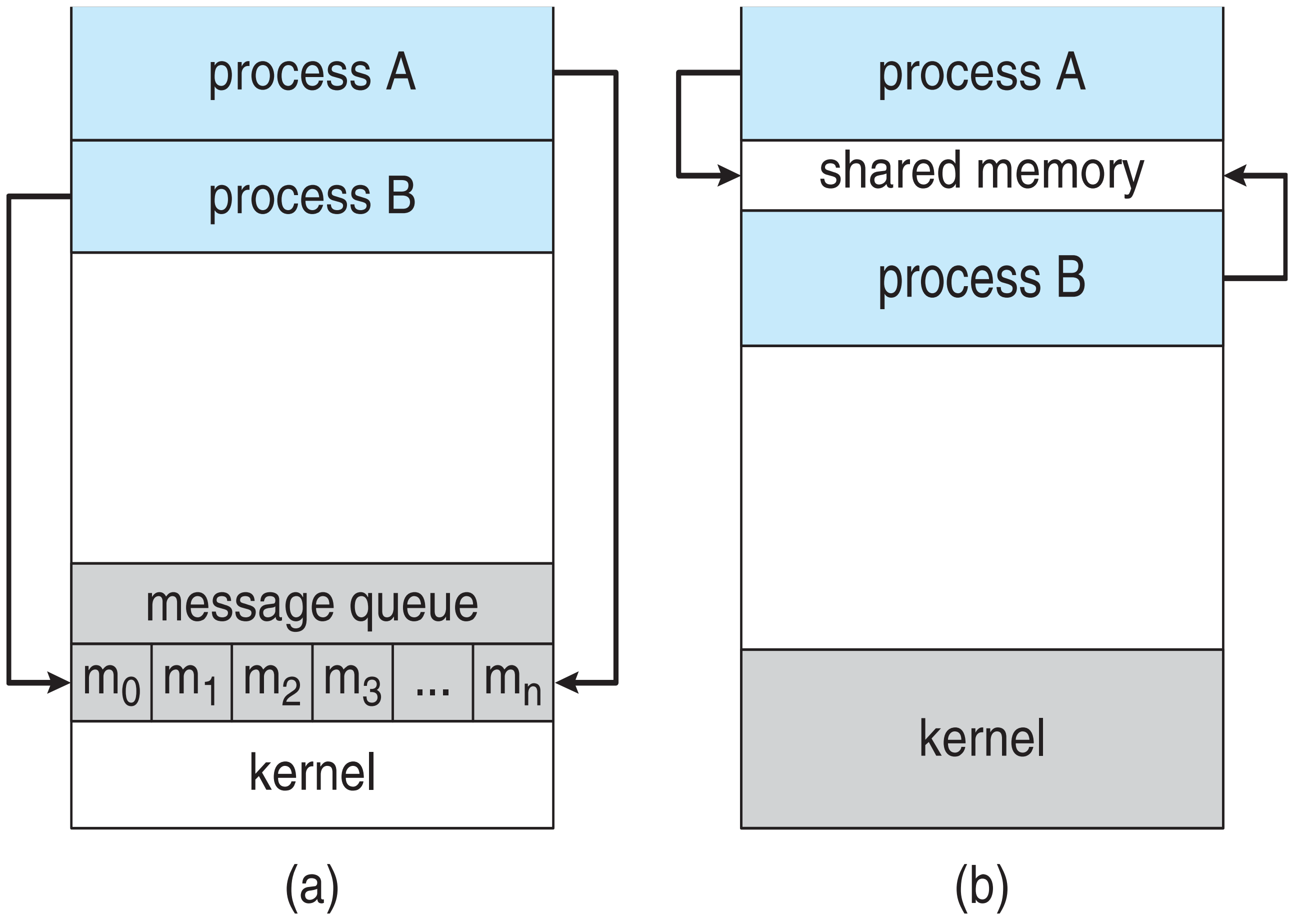
- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **IPC**

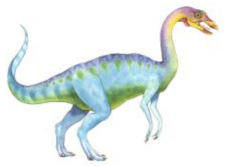
- This communication could involve
  - a process letting another process know that some event has occurred or
  - the transferring of data from one process to another.

- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

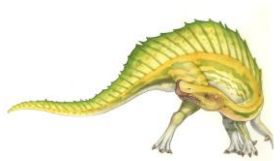
(a) Message passing.   (b) shared memory.





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



Bounded-Buffer – Shared-Memory Solution

- Shared data

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

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

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 */
}
```

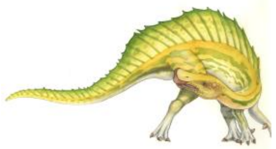
- Solution is correct, but can only use BUFFER_SIZE-1 elements

Interprocess Communication – 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.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

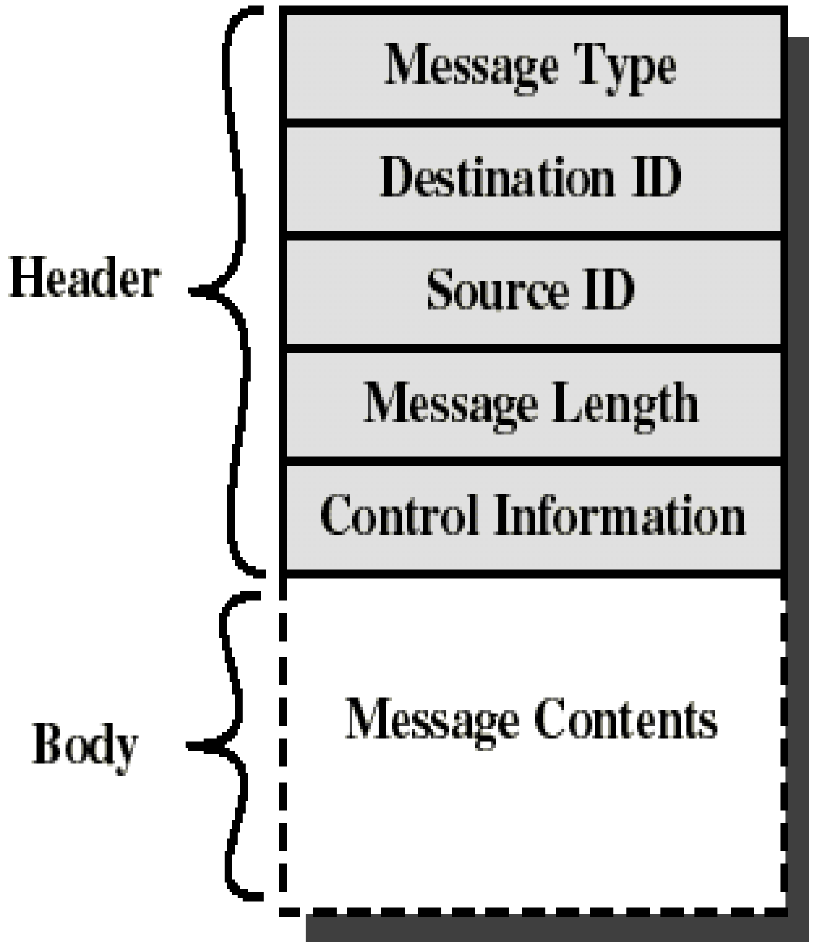
Interprocess Communication – Message Passing

- 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*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable



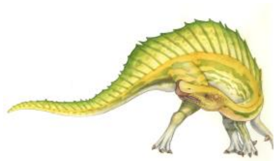
Message format

- Consists of header and body of message.
- In Unix: no ID, only message type.
- Control info:
  - what to do if run out of buffer space.
  - sequence numbers.
  - priority.
- **Queuing discipline: usually FIFO but can also include priorities.**



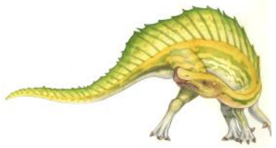
Message Passing (Cont.)

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



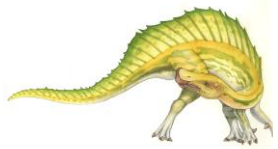
Message Passing (Cont.)

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



Naming: 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

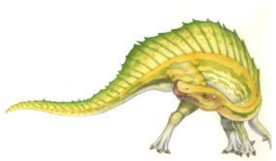


Naming : Direct Communication

- Symmetry : Both sender and receiver has to name the other to communicate
- Asymmetry:
  - **`send`** (*P*, *message*) – send a message to process P
  - **`receive`**(*id*, *message*) – receive a message from any process

Naming :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

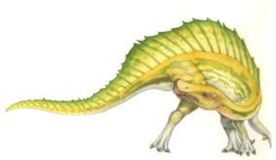


Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

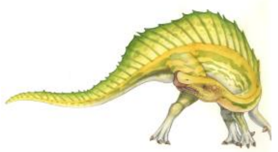
  `send`(*A, message*) – send a message to mailbox A

  `receive`(*A, message*) – receive a message from mailbox A



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.

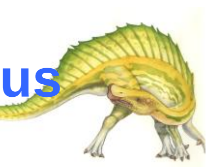


Use of a mailbox has following advantages:

- *Anonymity of receiver:* A process sending a message to request a service may have no interest in the identity of the receiver process, as long as the receiver process can perform the needed function.
  - A mailbox relieves the sender process of the need to know the identity of the receiver.
  - Additionally, if the OS permits the ownership of a mailbox to be changed dynamically, one process can readily take over the service of another.
- *Classification of messages:* A process may create several mailboxes, and use each mailbox to receive messages of a specific kind.
  - This arrangement permits easy classification of messages

Synchronization

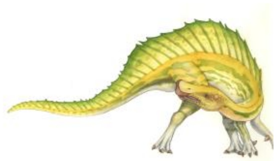
- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**



Synchronization (Cont.)

- 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 */
}
```

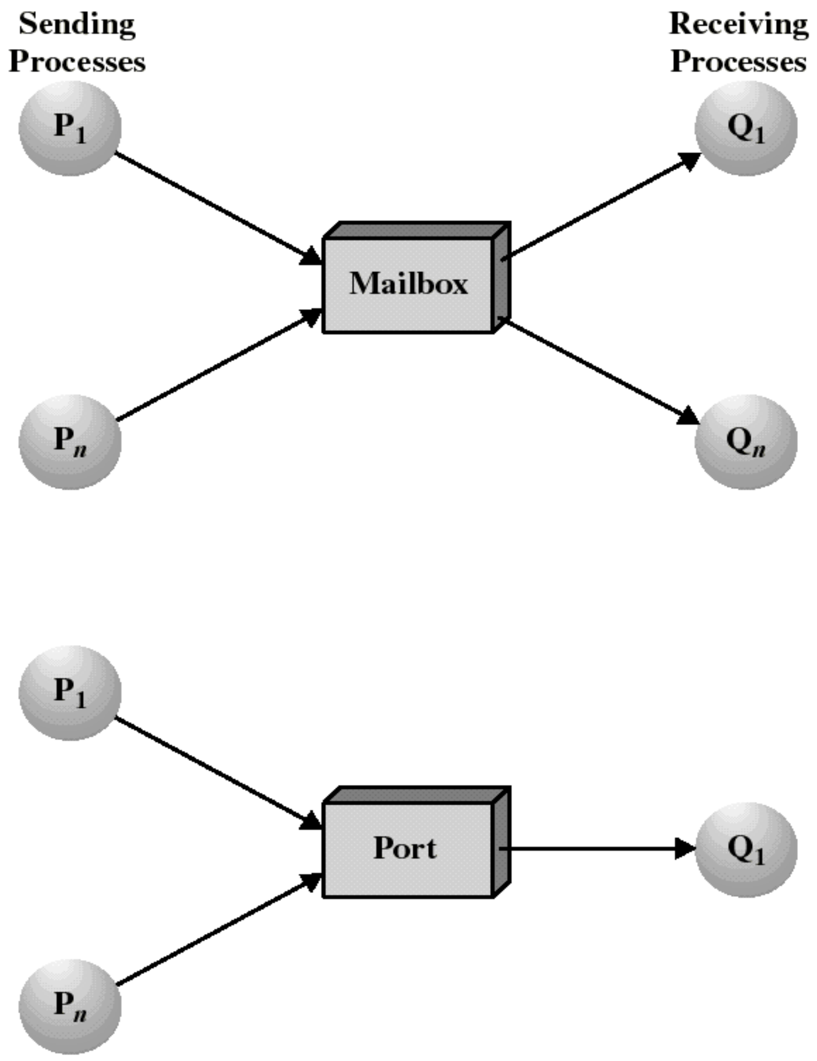


Buffering

- Queue of messages attached to the link.
  - Whether the communication is direct or indirect, messages exchanged by the processes reside in a temporary queue
- Implemented in one of three ways
- Zero capacity – no messages are queued on a link.
  Sender must wait for receiver (rendezvous)
- Bounded capacity – finite length of *n* messages
  Sender must wait if link full
- Unbounded capacity – infinite length
  Sender never waits

Mailboxes and Ports

- A mailbox can be private to one sender/receiver pair.
- The same mailbox can be shared among several senders and receivers:
  - the OS may then allow the use of message types (for selection).
- Port: is a mailbox associated with one receiver and multiple senders
  - used for client/server applications: the receiver is the server.



Ownership of ports and mailboxes

- A port is usually own and created by the receiving process.
- The port is destroyed when the receiver terminates.
- The OS creates a mailbox on behalf of a process (which becomes the owner).
- The mailbox is destroyed at the owner’s request or when the owner terminates.