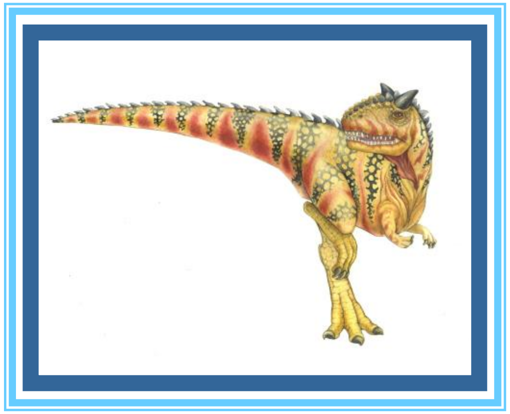
Chapter 3: Processes



Chapter 3: Processes

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

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

PROCESS CONCEPT

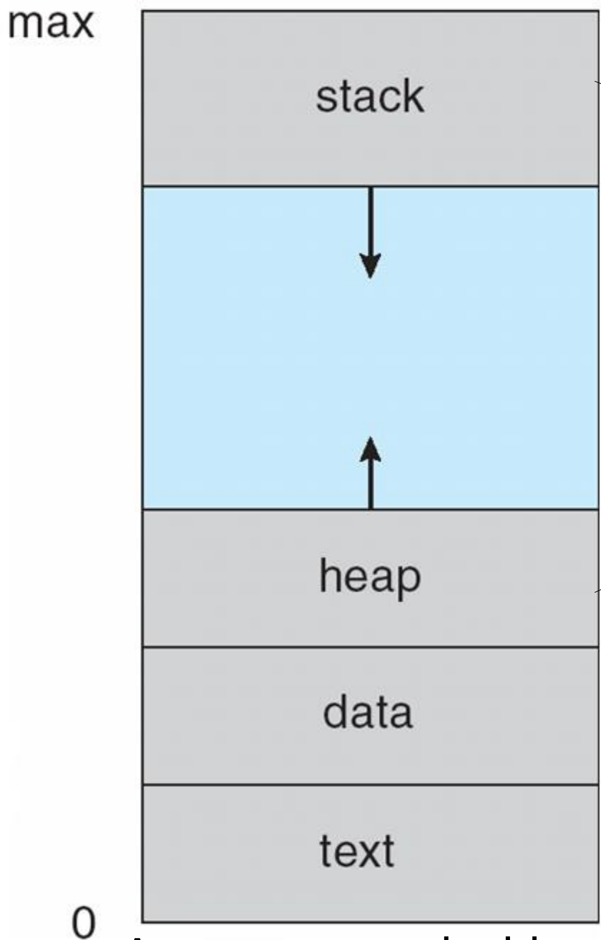
Process Concept

- 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

Process Concept (Cont.)

- 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



Stack segment
(holds the called function parameters, local variables)

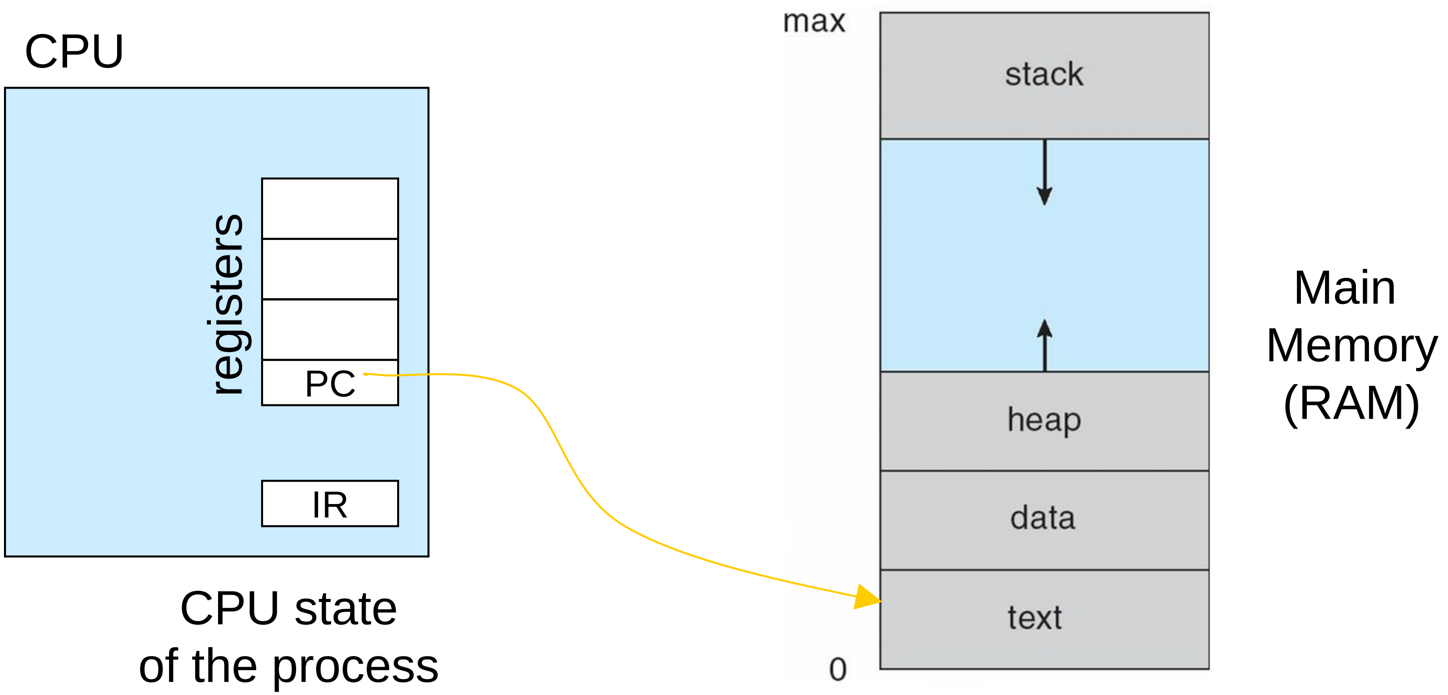
Storage for dynamically allocated variables

Data segment
(includes global variables, arrays, etc., you use)

Text segment
(code segment)
(instructions are here)

A process needs this to be in memory
(address space; memory image)

Process: program in execution



CPU state
of the process
(CPU context)

process address space

(currently used portion of the address space
must be in memory)

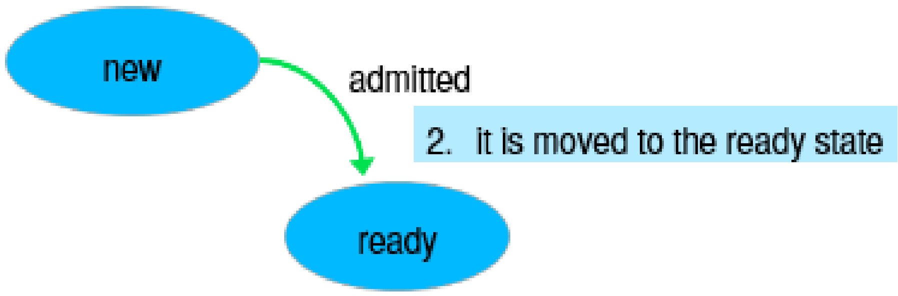
Process State

- 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
- In a single-CPU system, only one process may be in running state; many processes may be in ready and waiting states.

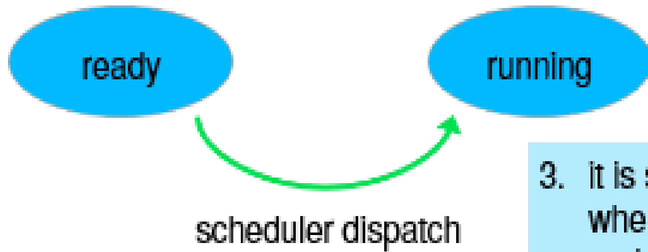
Process State Changes

1. a new process is created

Process State Changes

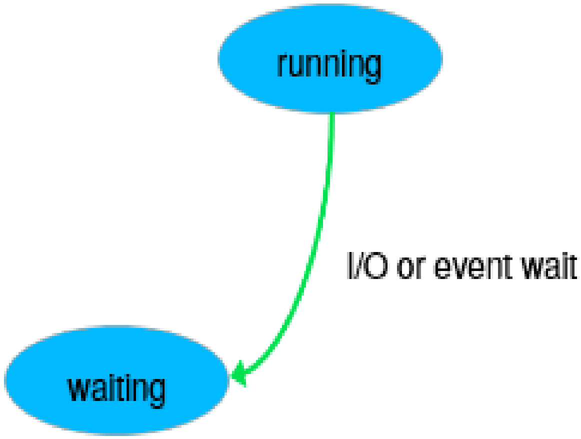


Process State Changes

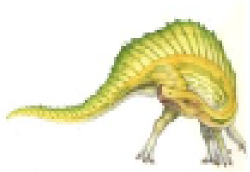


3. it is scheduled onto a core when the core is available and is now running

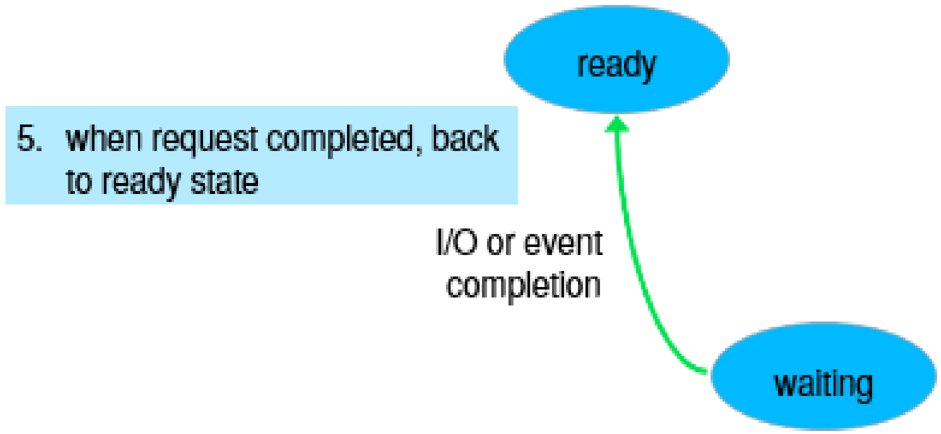
Process State Changes



4. If an I/O request or event request occurs, moves to waiting state

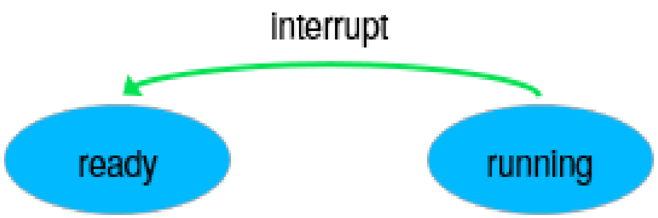


Process State Changes



Process State Changes

6. If running and the core is needed (say for an interrupt), back to the ready state



Process State Changes

7. cycle continues until the process exists, fails, or is terminated, moves to terminated state

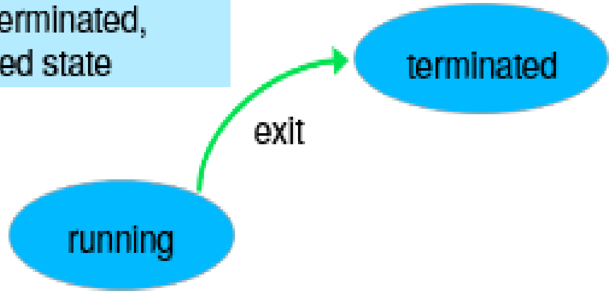
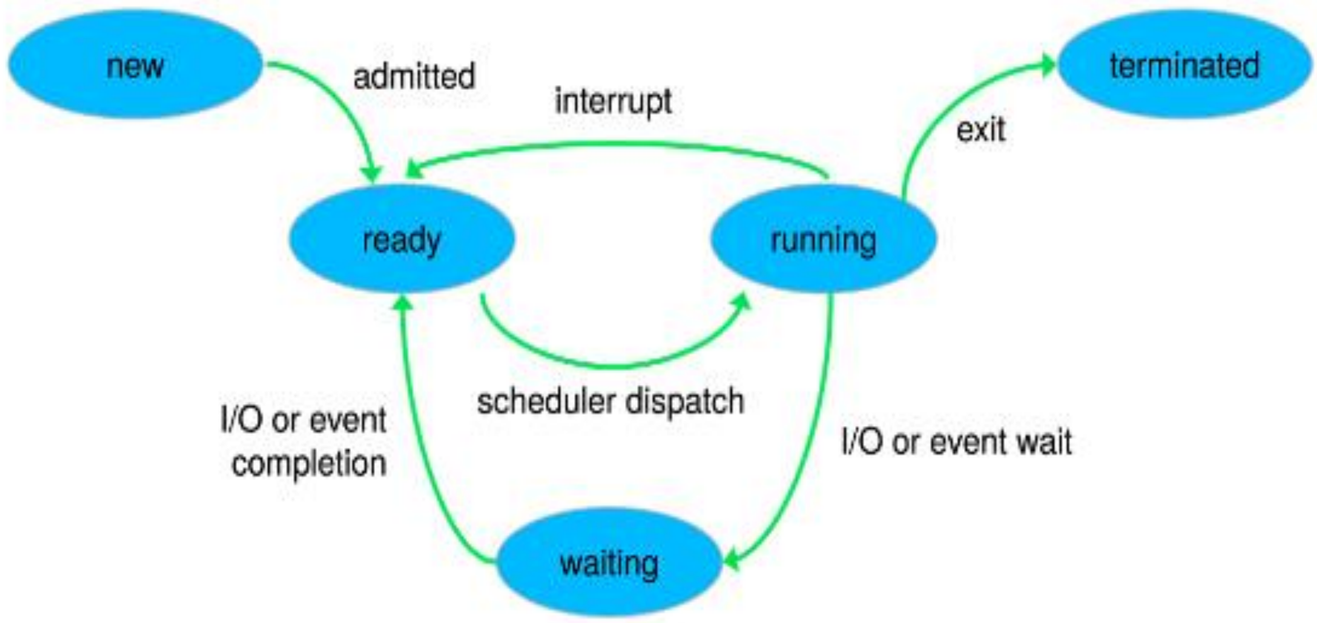
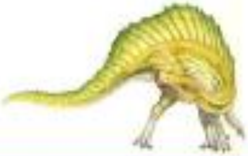


Diagram of Process State





Process: program in execution

- If we have a single program running in the system, then the task of OS is easy:
  - load the program, start it and program runs in CPU
  - (from time to time it calls OS to get some service done)

- But if we want to start several processes, then the running program in CPU (current process) has to be stopped for a while and other program (process) has to run in CPU.

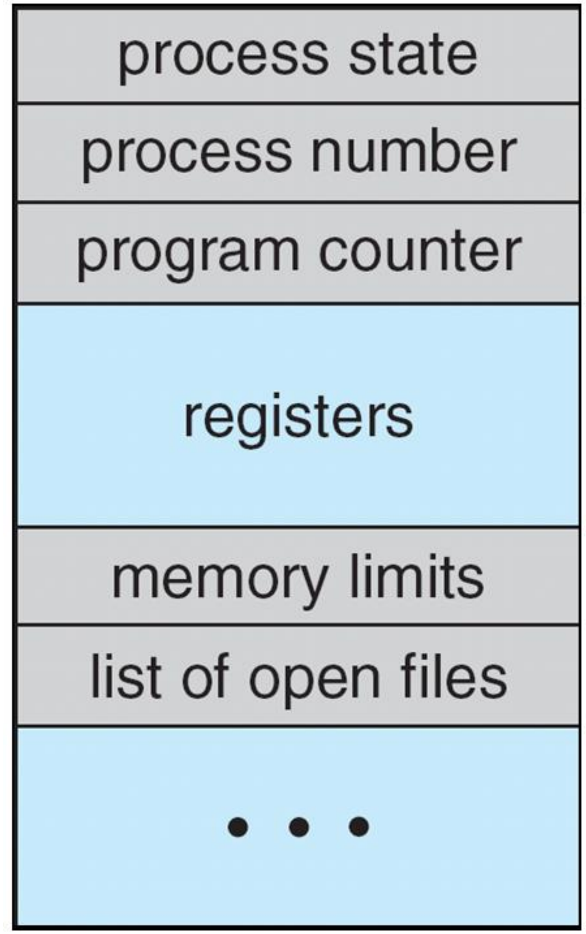
- To do this switch, we have to save the state/context (register values) of the CPU which belongs to the stopped program, so that later the stopped program can be re-started again as if nothing has happened.

Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

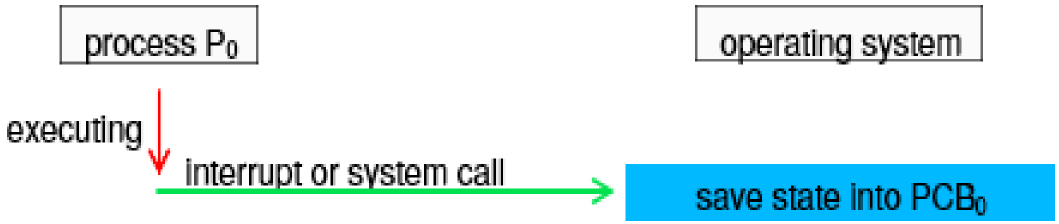
- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



Context Switch

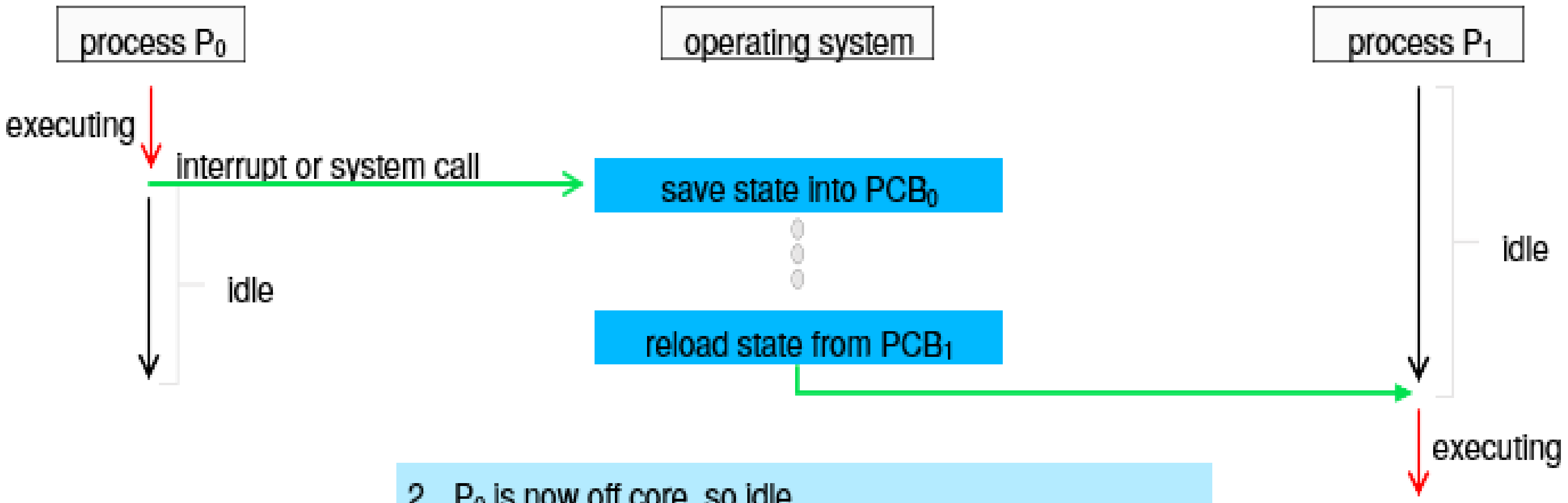
- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - 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

Context Switch from Process to Process

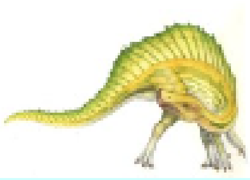


1. the system has two processes, P_0 and P_1
   - P_1 is idle, P_0 is executing and executes a system call, or the system receives an interrupt
   - the operating system saves the state of P_0 in its PCB

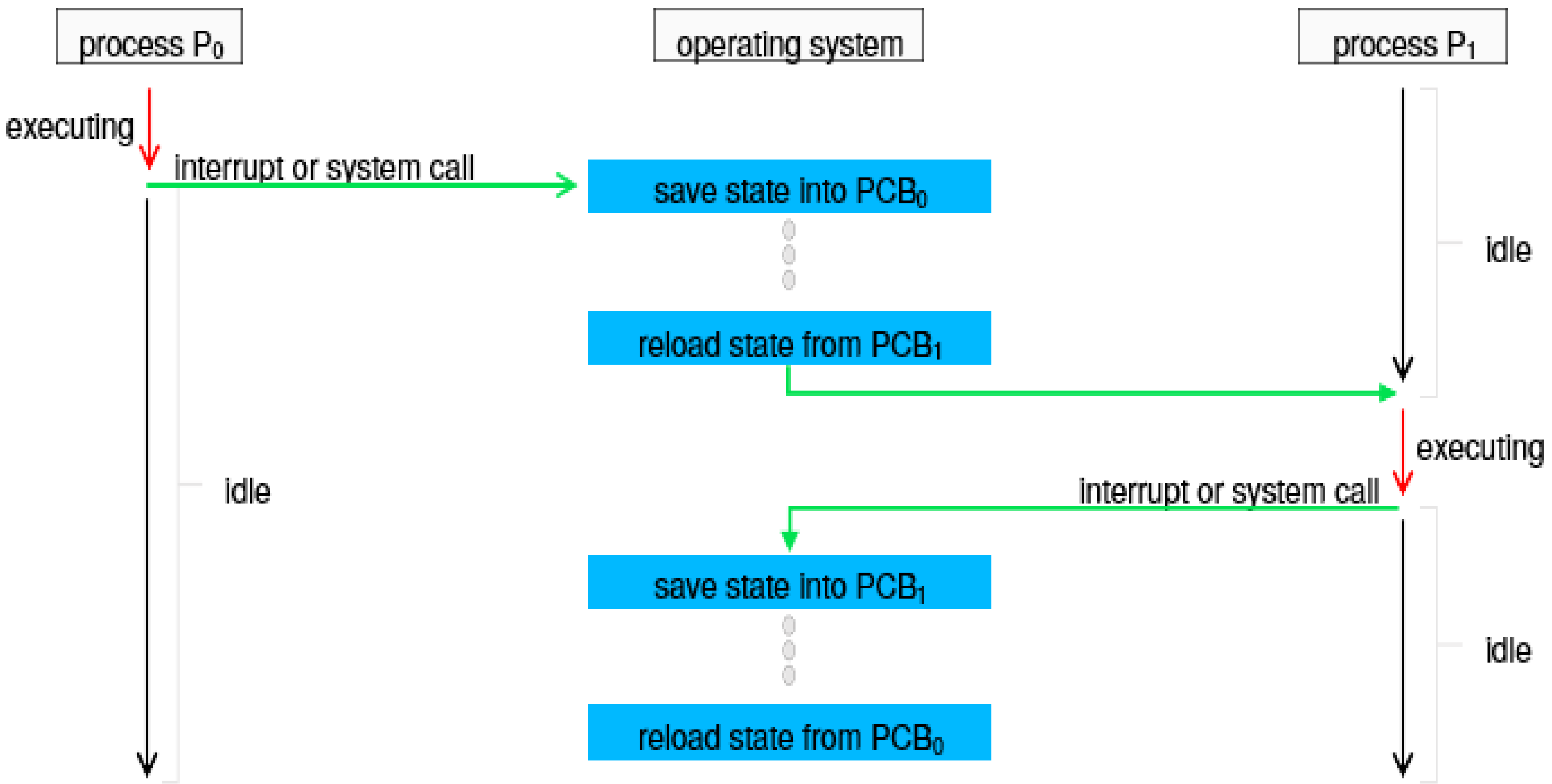
Context Switch from Process to Process



2. P_0 is now off core, so idle
   - the operating system restores the state of P_1 from its PCB and puts P_1 on the core
   - it continues execution

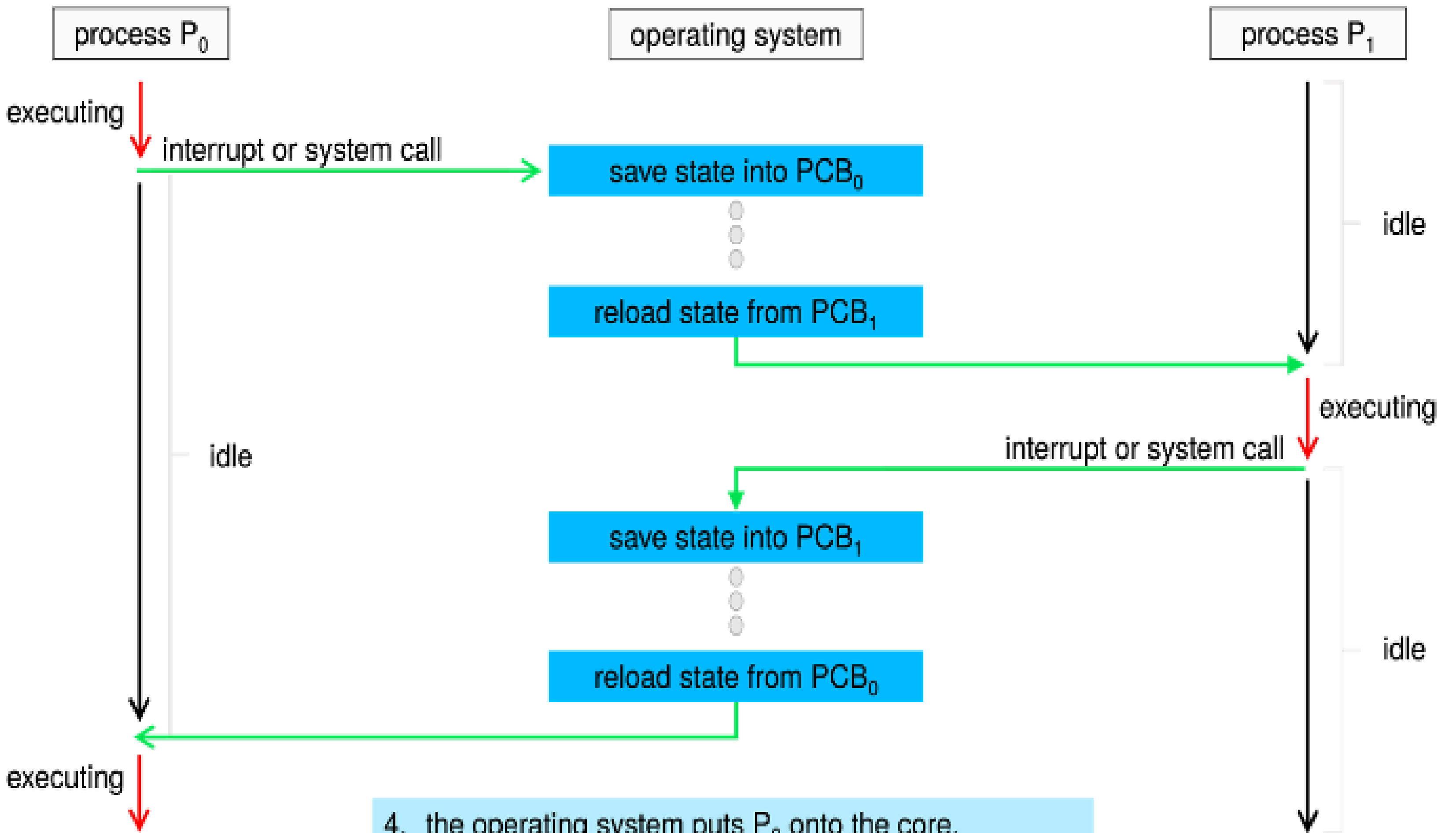


Context Switch from Process to Process



3. P1 execution is interrupted, the operating system saves its state to its PCB and restores the next process's state (P0 in this case) to prepare it to continue execution

Context Switch from Process to Process



4. the operating system puts P_0 onto the core, continuing its execution while P_1 is now idle, in ready state, waiting its turn

Threads

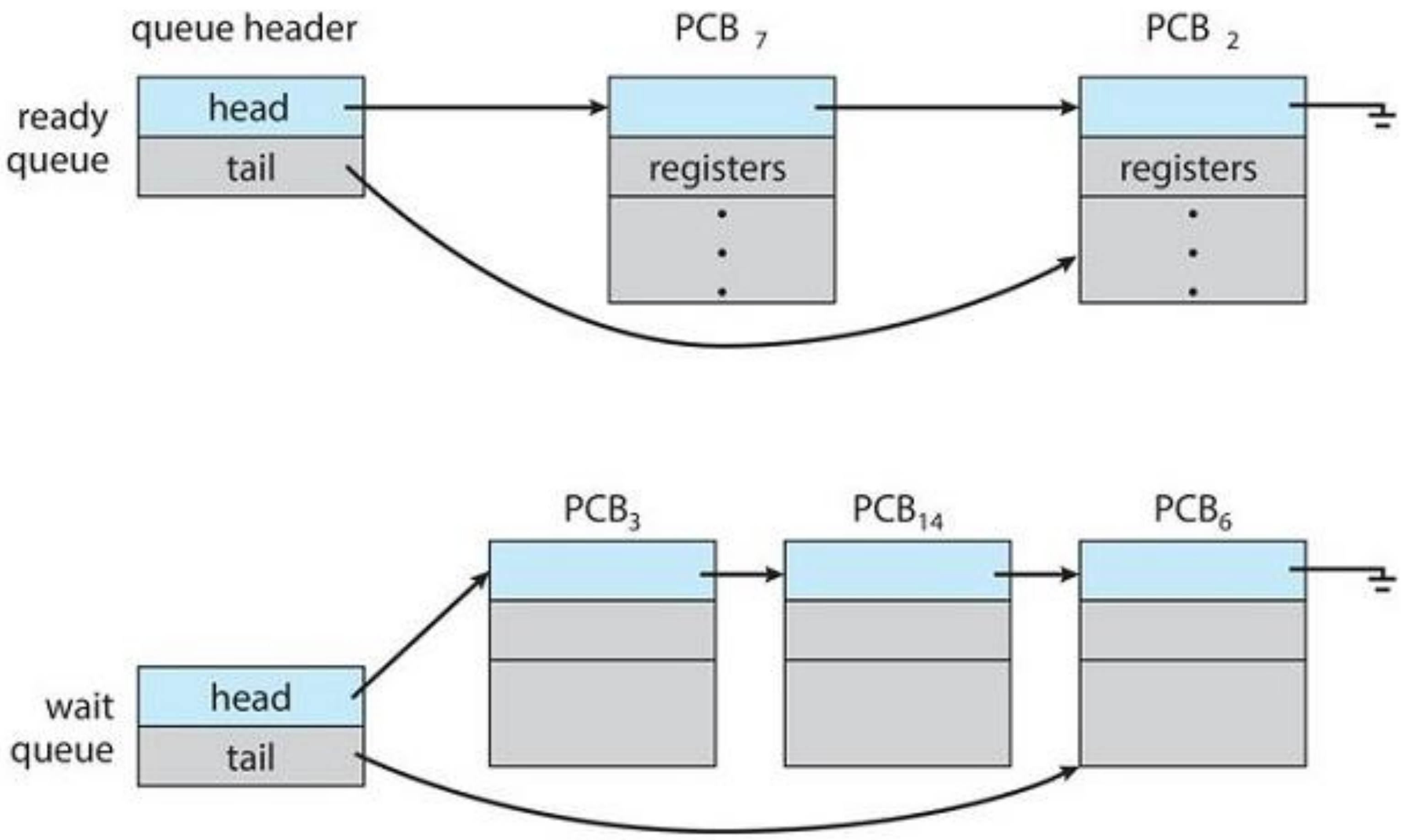
- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- See next chapter

PROCESS SCHEDULING

Process Scheduling

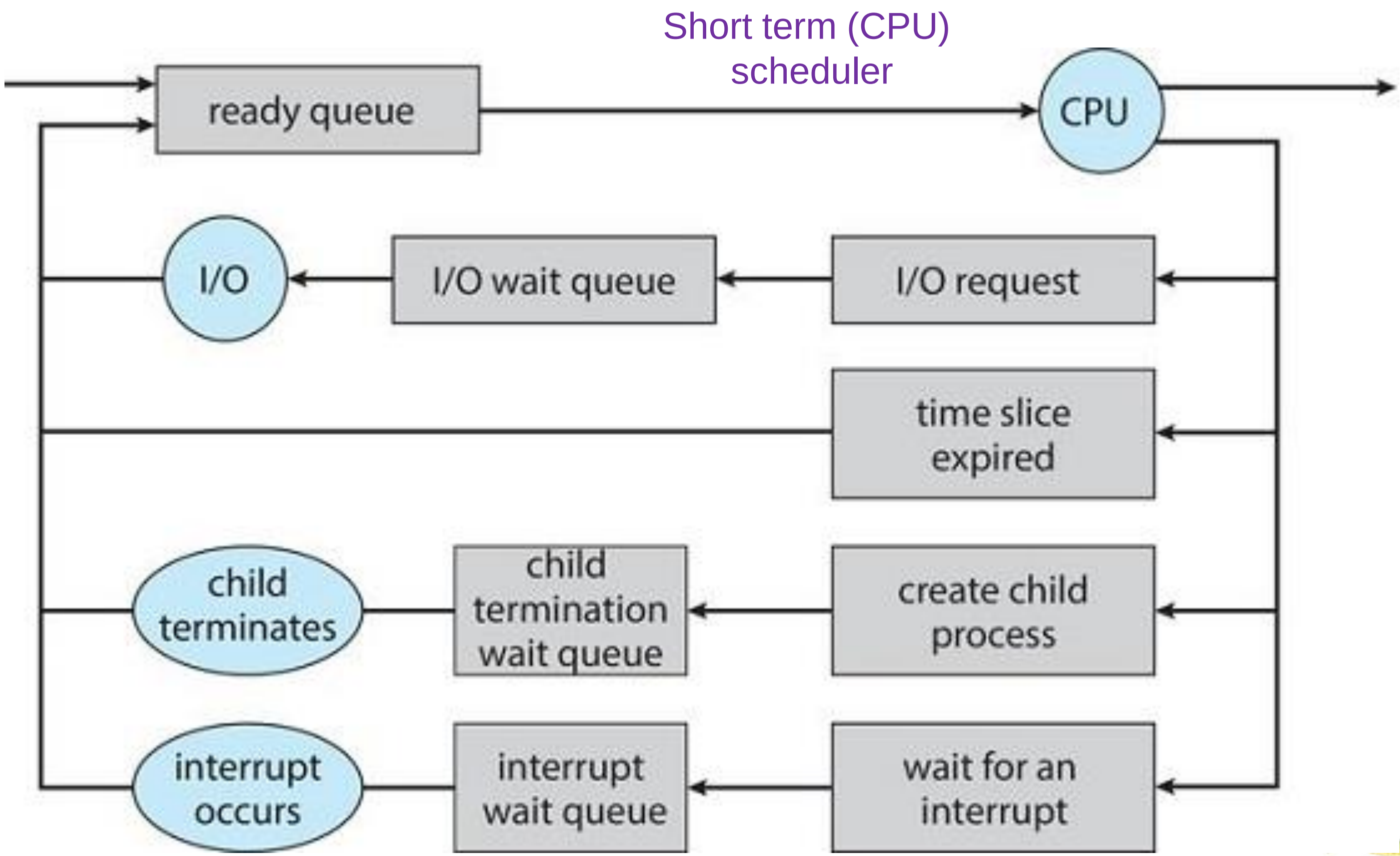
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing 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 A Wait Queue



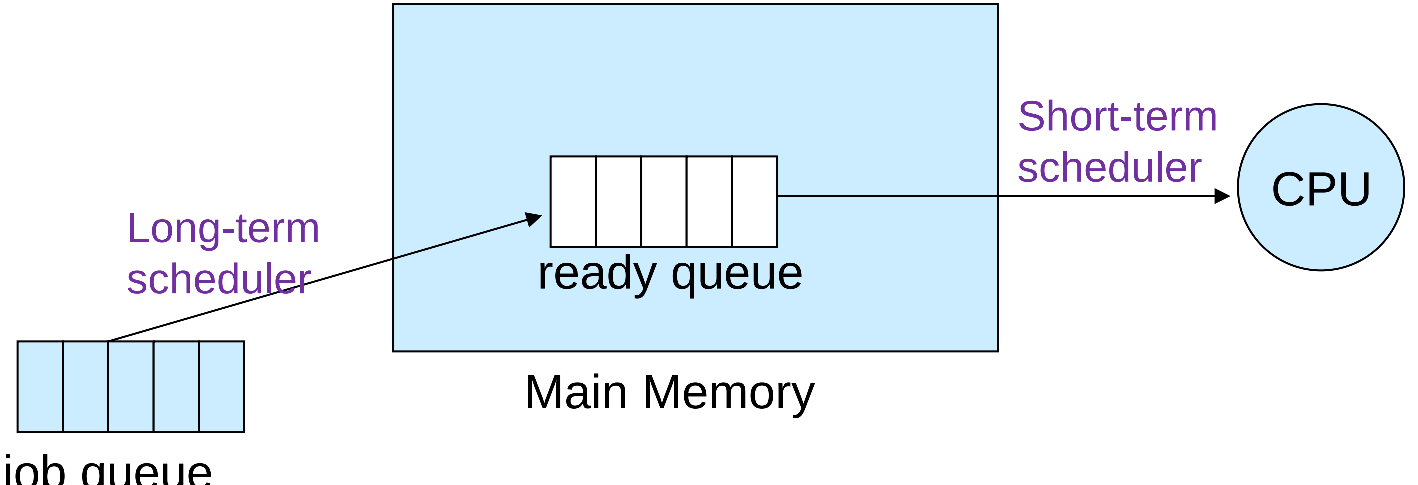
Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows.
- Two types of queues are present: the ready queue and a set of wait queues.
- The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.



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) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory)

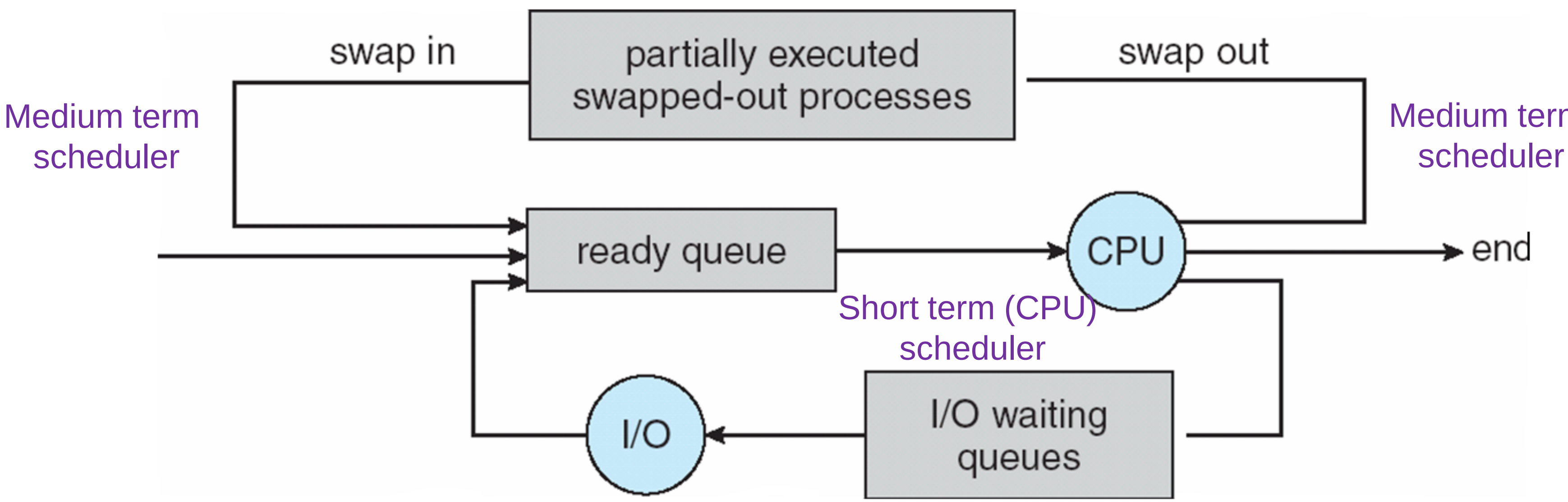


Schedulers- Cont.

- **CPU burst**: the execution of the program in CPU between two I/O requests (i.e. time period during which the process wants to continuously run in the CPU without making I/O)
  - We may have a short or long CPU burst.
- 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**



OPERATIONS ON PROCESSES

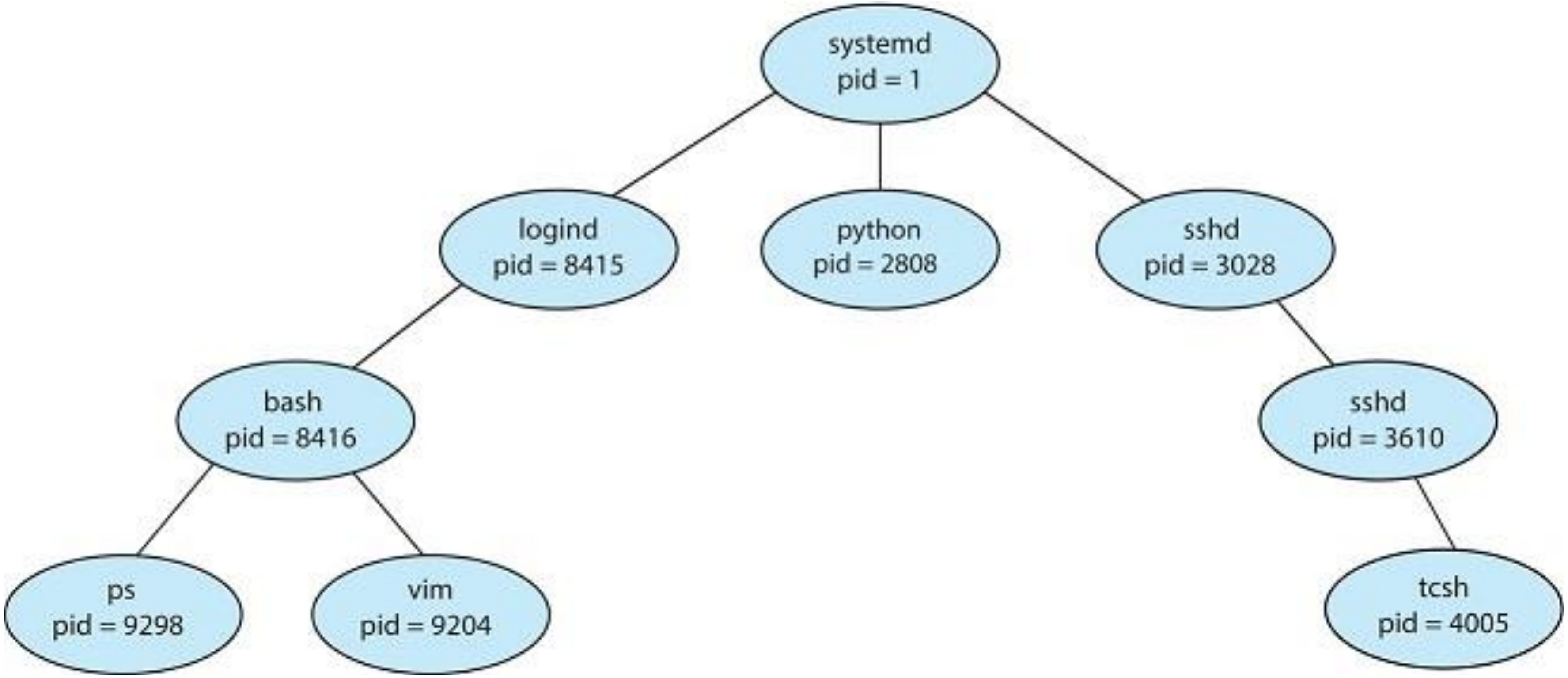
Operations on Processes

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

Process Creation

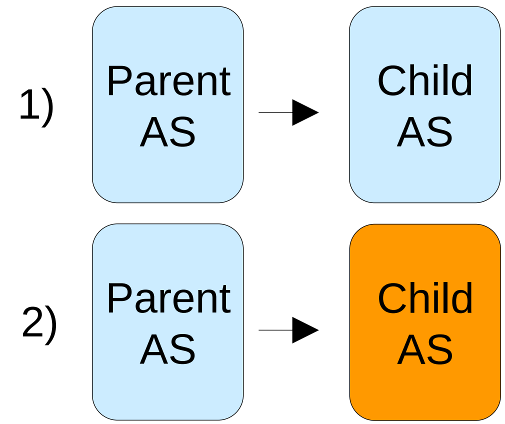
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset 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



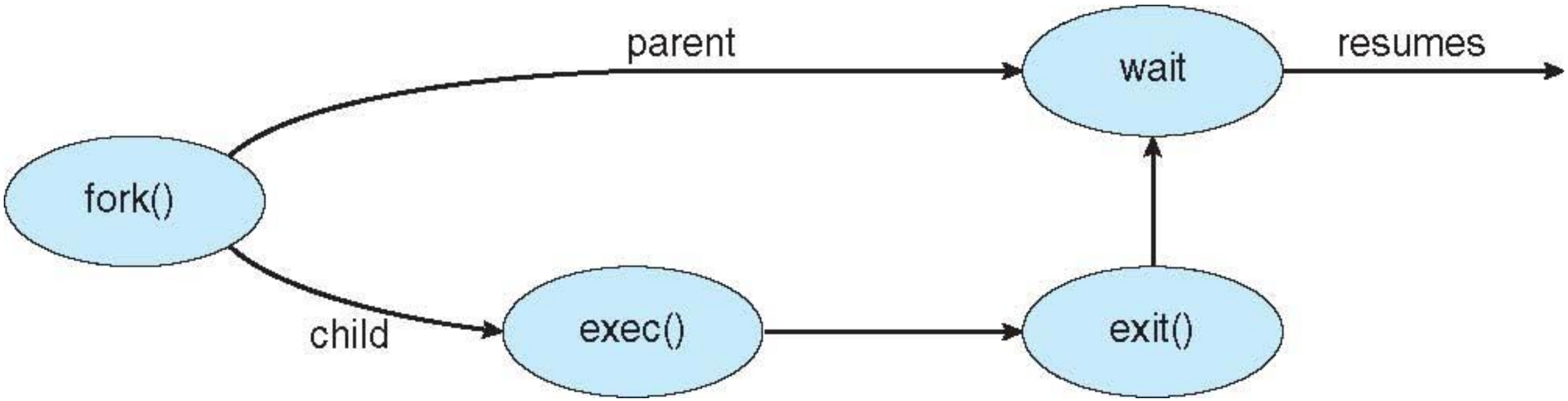
Process Creation (Cont.)

- Address space
  - 1) Child duplicate of parent
  - 2) Child has a program loaded into it

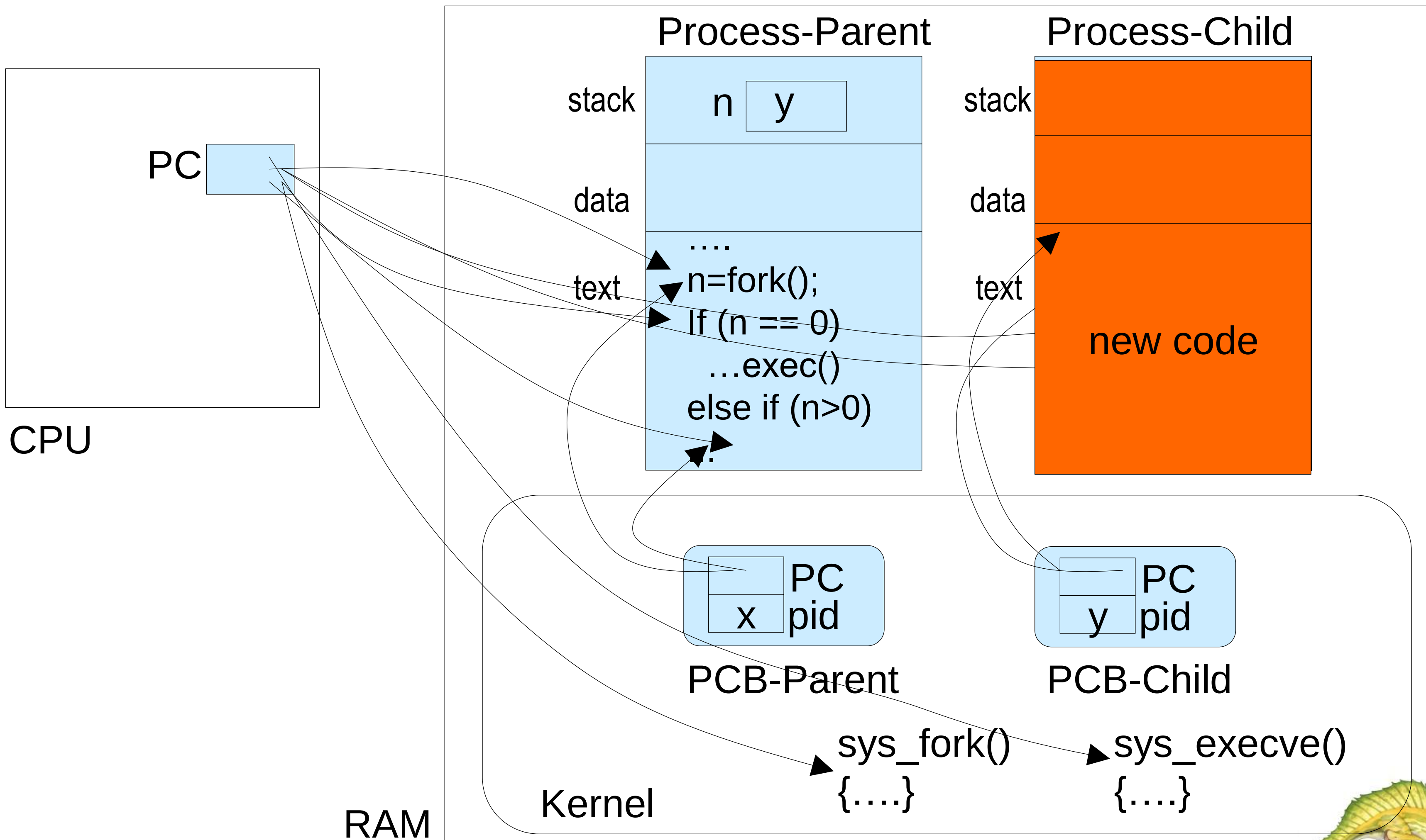


- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program

Process Creation (Cont.)



Execution Trace: fork() with execlp()



Slide borrowed from Dr. İbrahim Körpeoğlu, Bilkent University

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

```c
#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);
}
```

Process Termination

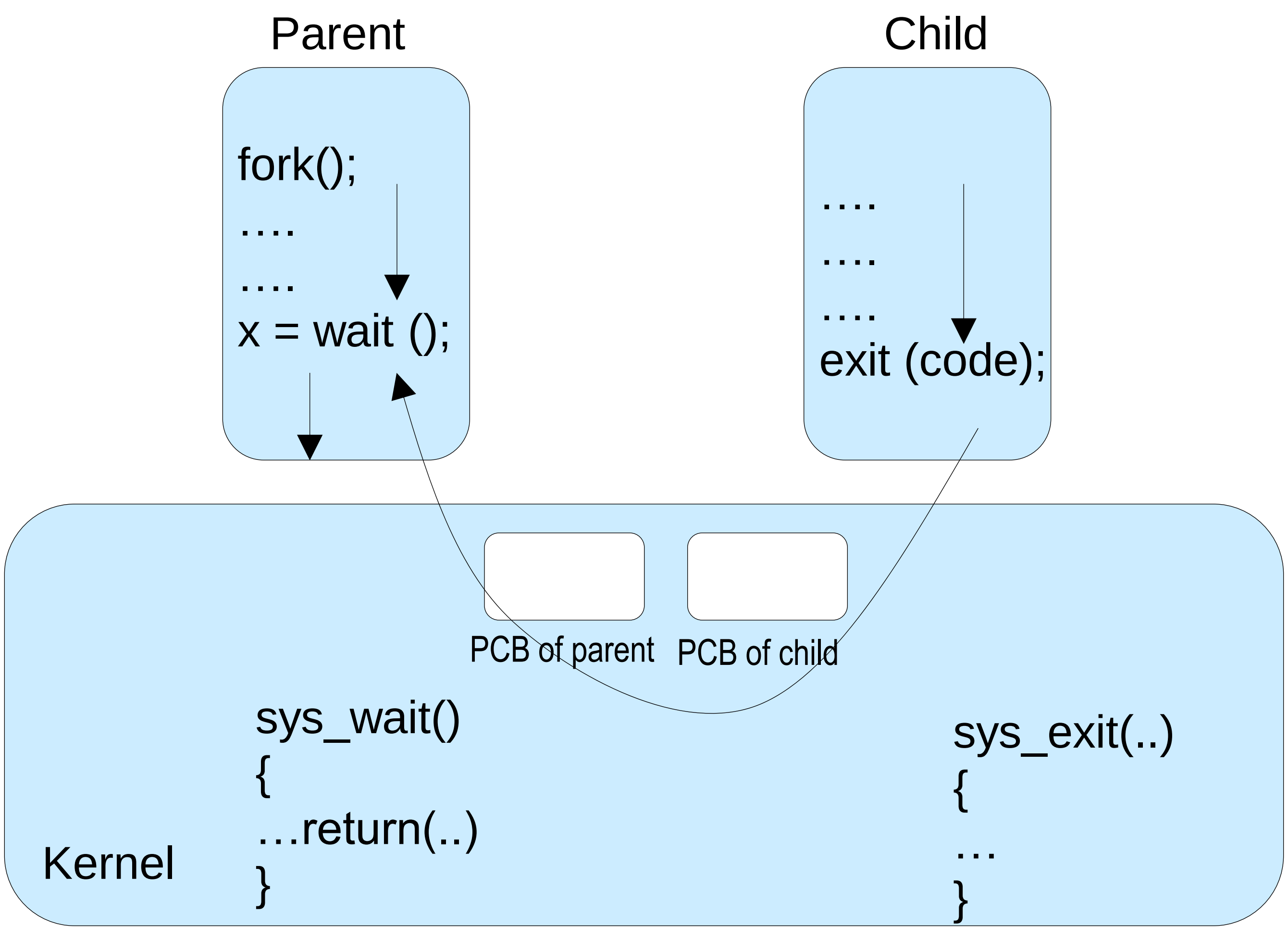
- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using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

Process Termination



Slide borrowed from Dr. İbrahim Körpeoğlu, Bilkent University

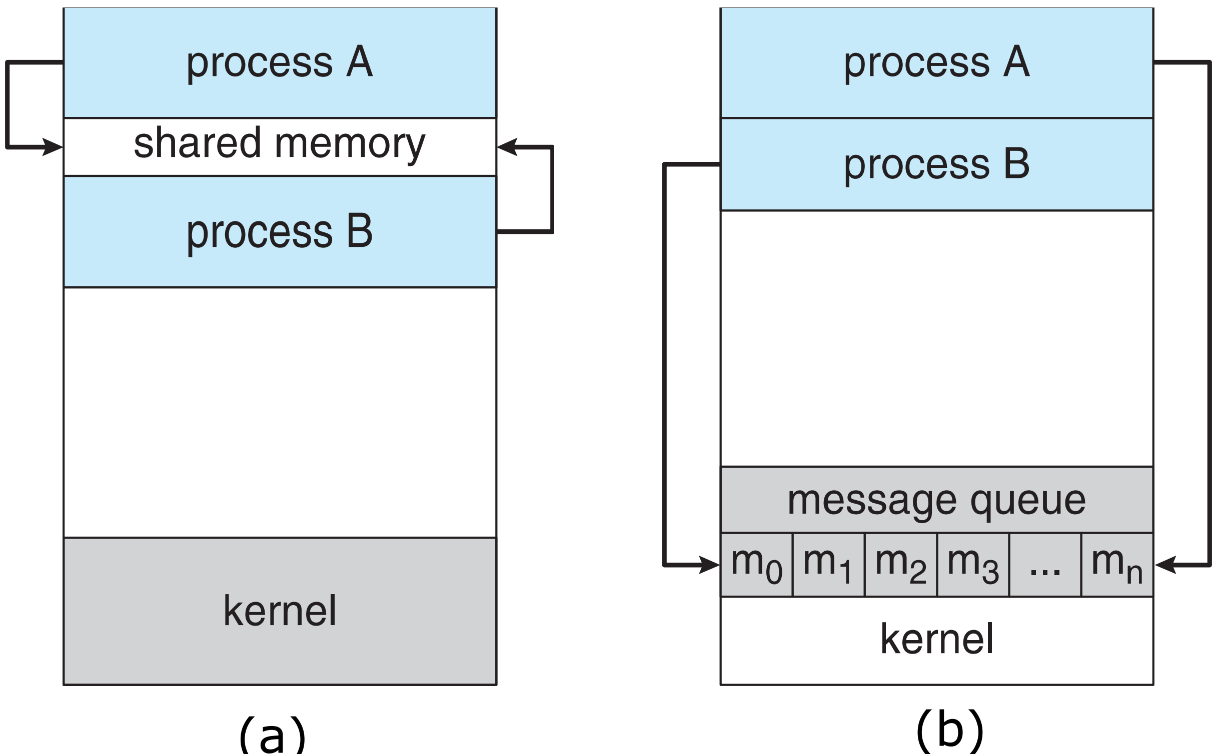
INTER-PROCESS COMMUNICATION

Cooperating Processes

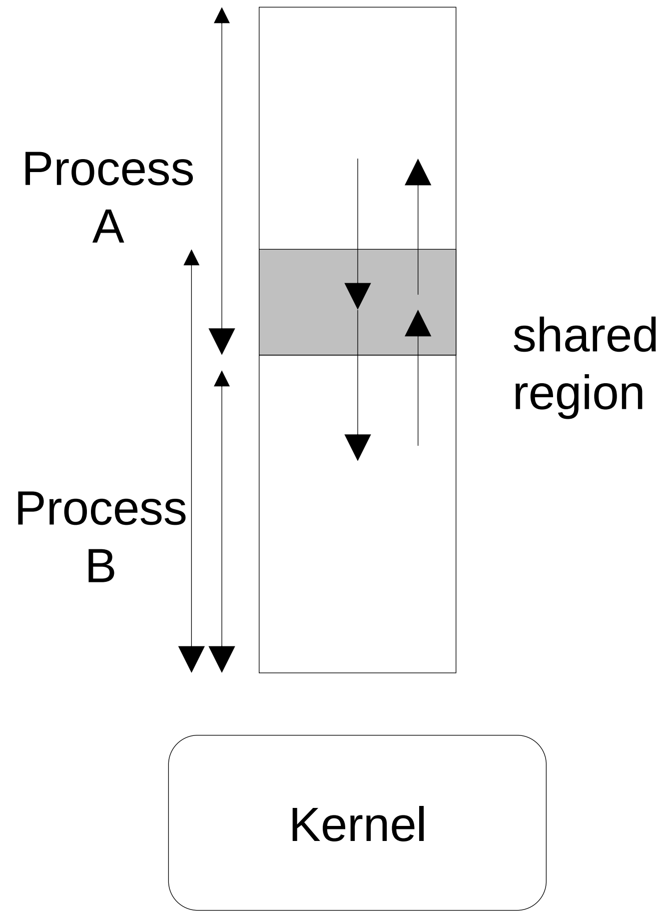
- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating processes require a facility/mechanism for **interprocess communication** (**IPC**)
- Two models of IPC
  - **a) Shared memory**
  - **b) Message passing**

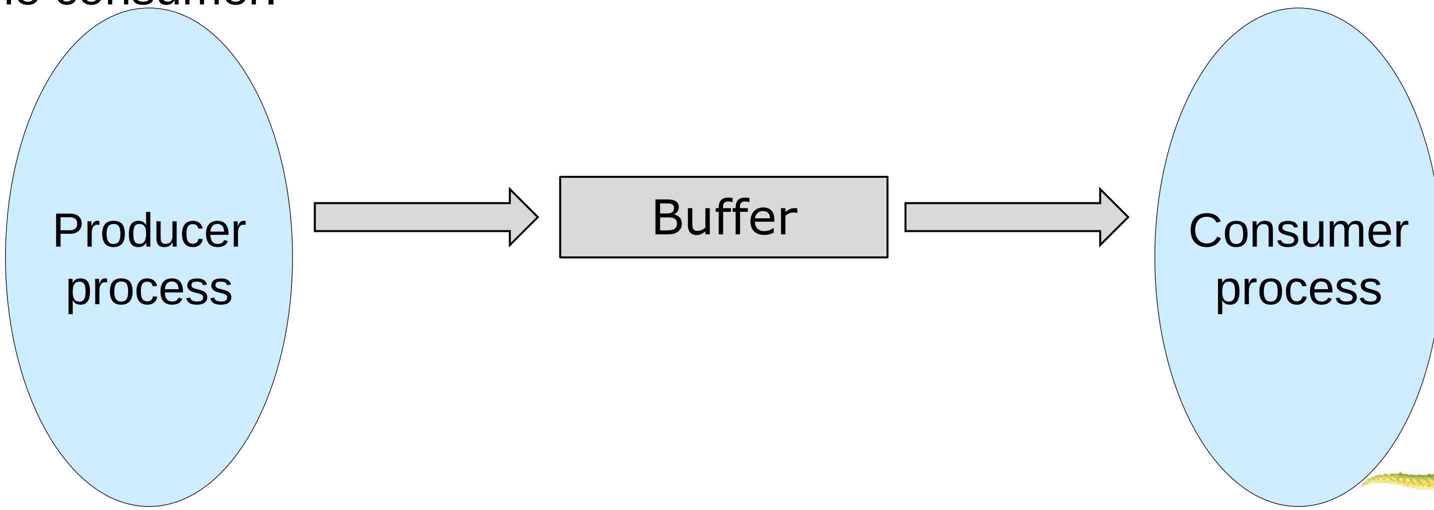


- A region of shared memory is established among two or more processes.
- Establishment of that shared region is done via the help of the kernel (via a system call).
- Then, processes can read and write shared memory region directly as ordinary memory access
- During this time, kernel is not involved: the communication is under the control of the users processes not the operating system.
- Hence it is faster than message passing.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapter 5.



Producer-Consumer Problem

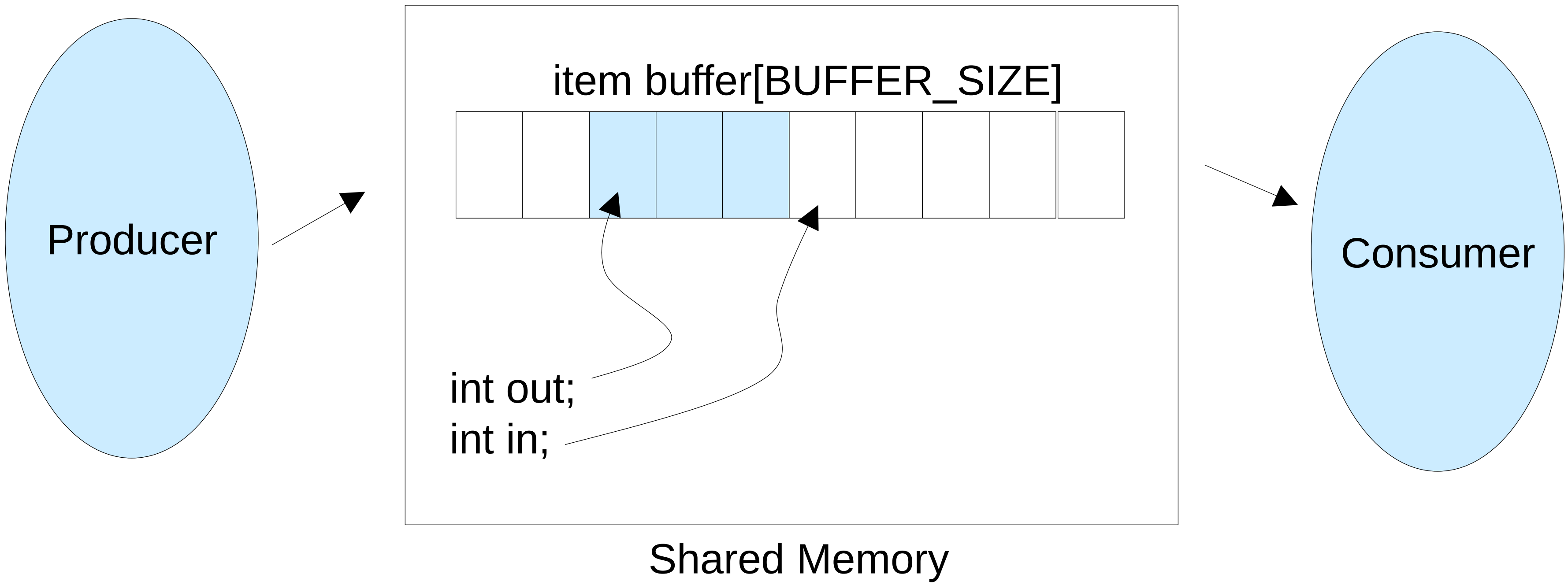
- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
  - E.g., a compiler may produce assembly code, which is consumed by an assembler.
  - E.g., client–server paradigm: a Web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client Web browser requesting the resource.
- One solution to the producer–consumer problem uses shared memory: use a buffer of items that can be filled by the producer and emptied by the consumer.



Producer-Consumer Problem

- Two types of buffers can be used:
  - **unbounded-buffer** places no practical limit on the size of the buffer
    - Consumer may have to wait for new items,
    - Producer can always produce new items.
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Consumer must wait if the buffer is empty,
    - Producer must wait if the buffer is full.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Buffer State in Shared Memory



Bounded-Buffer – Shared-Memory Solution

- Shared data

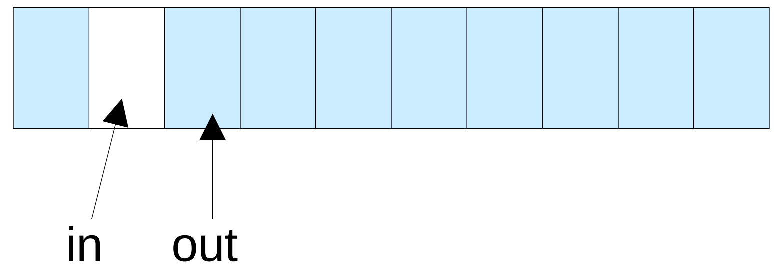
```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

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

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

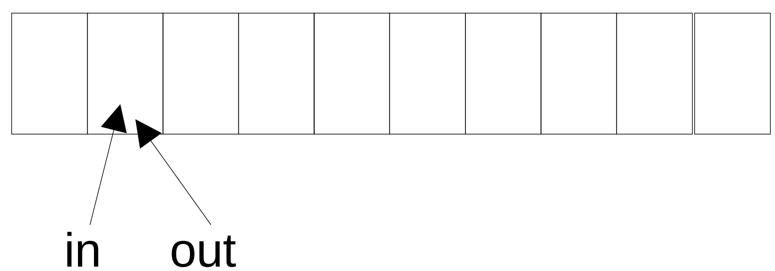
Buffer State in Shared Memory

Buffer Full



((in+1) % BUFFER_SIZE == out) : considered full buffer

Buffer Empty



In == out : empty buffer

Producer-Consumer Code 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;
}
```

Shared Memory

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

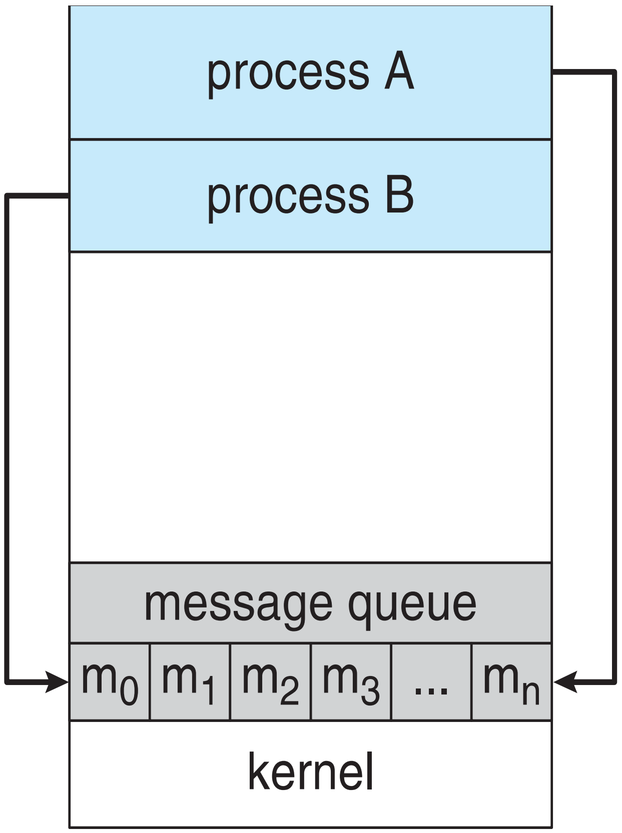
Consumer

```
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
- Particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network
- E.g. **chat** program used on the World Wide Web: chat participants communicate by exchanging messages.
- IPC facility provides two operations:
  - `send`(*message*)
  - `receive`(*message*)
- The *message* size is either fixed or variable



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

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

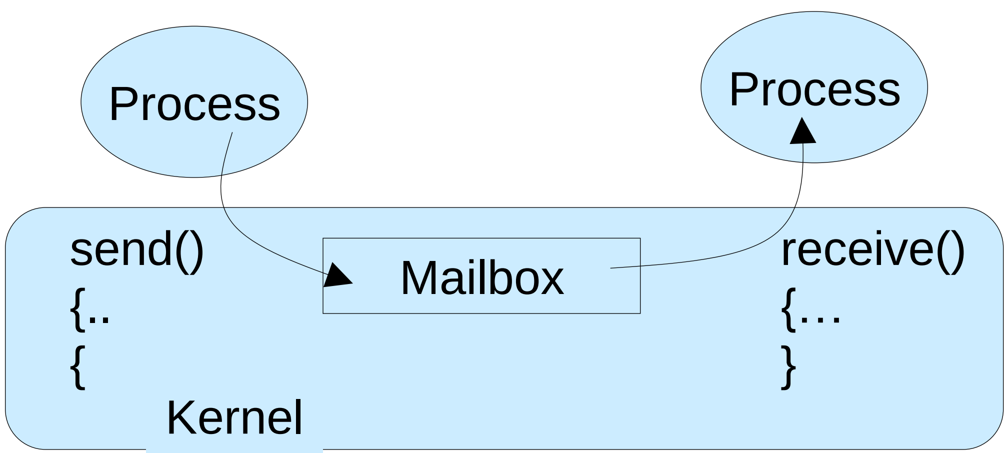
- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with 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 (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.

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.
- implemented in one of three ways
  1. **Zero capacity**
     - No messages are queued on a link.
     - 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

End of Chapter 3