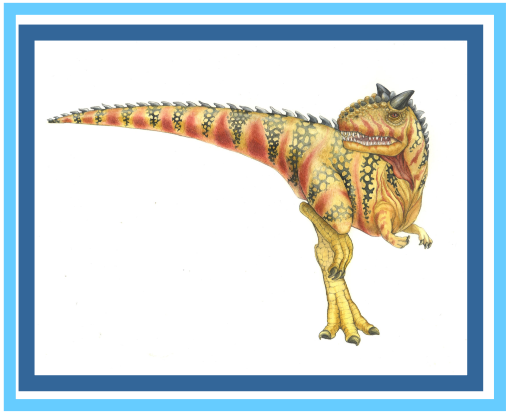
Chapter 3: Processes



Copyright

Chapter 3: Processes

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

Process Concept

INFORMAL DEFINITION:

Deterministic process – a **program** in **execution**

Process Concept

INFORMAL DEFINITION:

Deterministic process – a **program** in **execution**; the execution must progress in **sequential** fashion

Process Concept

INFORMAL DEFINITION:

Deterministic process – a **program** in **execution**; the execution must progress in **sequential** fashion

We will usually omit the adjective **deterministic** if it is clear from the context that the process in question is *deterministic*.

Process Concept

INFORMAL DEFINITION:

Deterministic process – a **program** in **execution**; the execution must progress in **sequential** fashion

We will usually omit the adjective **deterministic** if it is clear from the context that the process in question is *deterministic*.

Process has its **program** and its **state** (a.k.a. **context**)

Process Concept

INFORMAL DEFINITION:

Deterministic process – a **program** in **execution**; the execution must progress in **sequential** fashion

We will usually omit the adjective **deterministic** if it is clear from the context that the process in question is *deterministic*.

Process has its **program** and its **state** (a.k.a. **context**)

The program is fixed while the state changes as the execution proceeds.

Process Concept

A process' **state** *typically* includes:

- program counter
- status register
- stack (pointer)
- data section
- registers
- memory
- files
- etc.

Definition of Execution

Execution is a function **S: N --> States**, where

N is the set of natural numbers (**discrete time**)

States is the set of possible states of computation

Each **S(t)** is a (momentary) state of execution **S** at time **t**

(a.k.a. current **context** at time **t**)

NOTE Function **S** can be written in a sequential form:

S = <**S(0)**, **S(1)**, **S(2)**,..., **S(t)**,...>

Definition of Execution

Each **S(t)** contains:

PC(t) - current value of the program counter at time t

Mem(t) - current memory contents at time t available to

the process

Definition of Program's Execution

Given *deterministic* program **P**, the recurrence relation

between **S(t)** and **S(t+1)** is described by the transition

function $\mathbf{T_P}$ that is defined by the program **P**

$$\mathbf{S(t+1) = T_P(S(t))}, \text{ for every } t \text{ in } N$$

Definition of Program's Execution

NOTE Technically, definition of a program **P**'s execution must include a **mapping** from **P**'s variables onto addressing space of **Mem**. For HLL's, this may be easily accomplished by assuming that **P**'s variables are the actual elements of **Mem**.

Definition of Program's Execution

Deterministic program's execution is determined by a pair

<**P**, **S(0)**>

Where:

P is the program (constant)

S(0) is the state of computation at time **0** (which includes the initial value **PC(0)** of the program counter and the initial contents **Mem(0)** of **P**'s memory)

Definition of Program's Execution

Deterministic program's execution is determined by a pair

<P, S(0)>

Where:

P is the program (constant)

$\mathbf{S(1) = T_P(S(0))}$,

$\mathbf{S(2) = T_P(S(1))}$,

$\mathbf{S(3) = T_P(S(2))}$,

etc.

Formal Definition of Process

Process is defined as a pair

<P, S>

where:

P is the program (constant)

S is a function from **N** into **States** (execution, as defined before).

Formal Definition of Process

Process is defined as a pair

<P, S>

where:

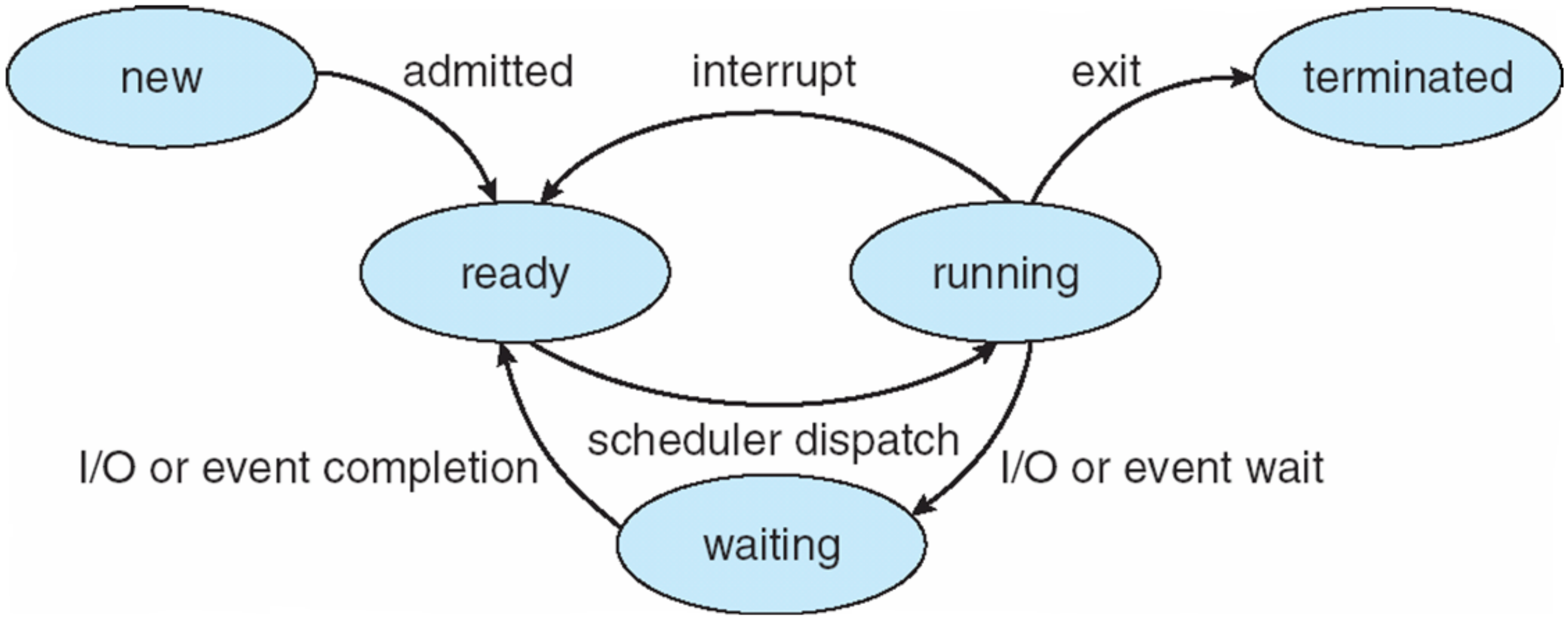
P is the program (constant) – *deterministic or not*

S is a function from **N** into **States** (execution, as defined before).

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



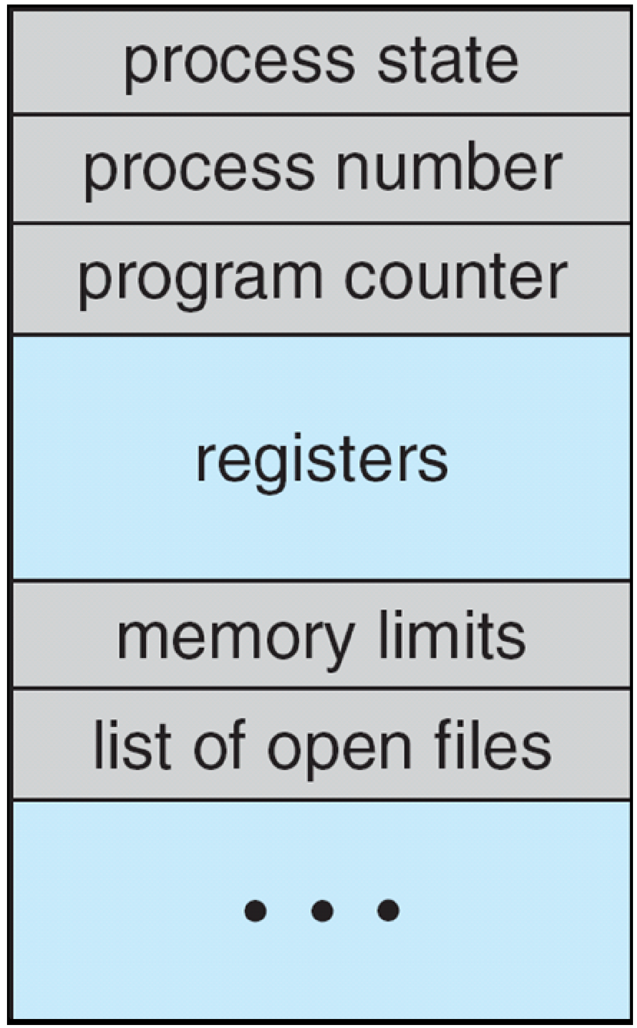
Process Control Block (PCB)

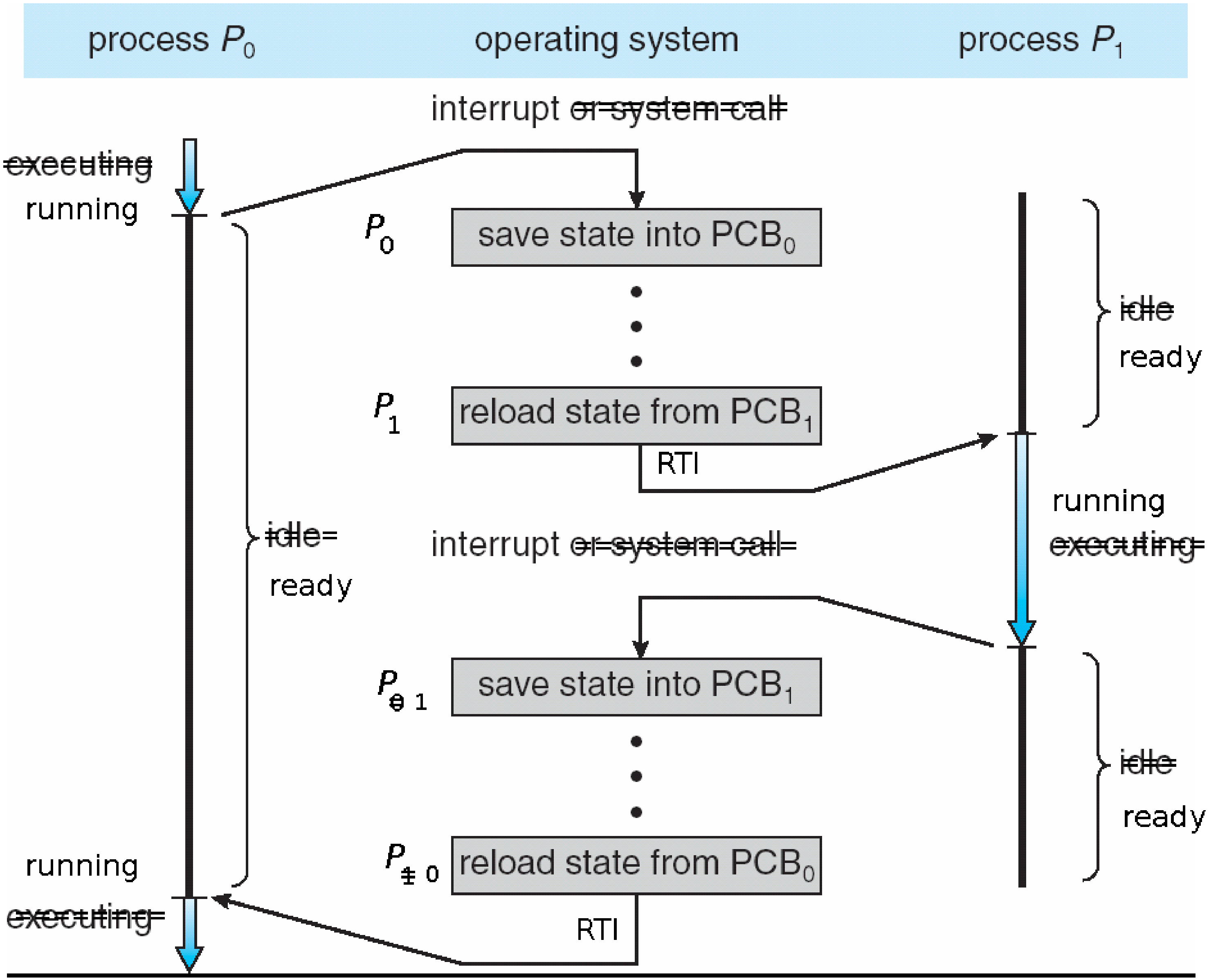
Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process Control Block (PCB)

Context:





process P_0
operating system
process P_1
interrupt or system call
executing
running
P_0
save state into PCB_0
idle
ready
P_1
reload state from PCB_1
RTI
running
executing
interrupt or system call
idle
ready
P_1
save state into PCB_1
idle
ready
P_0
reload state from PCB_0
running
executing
RTI

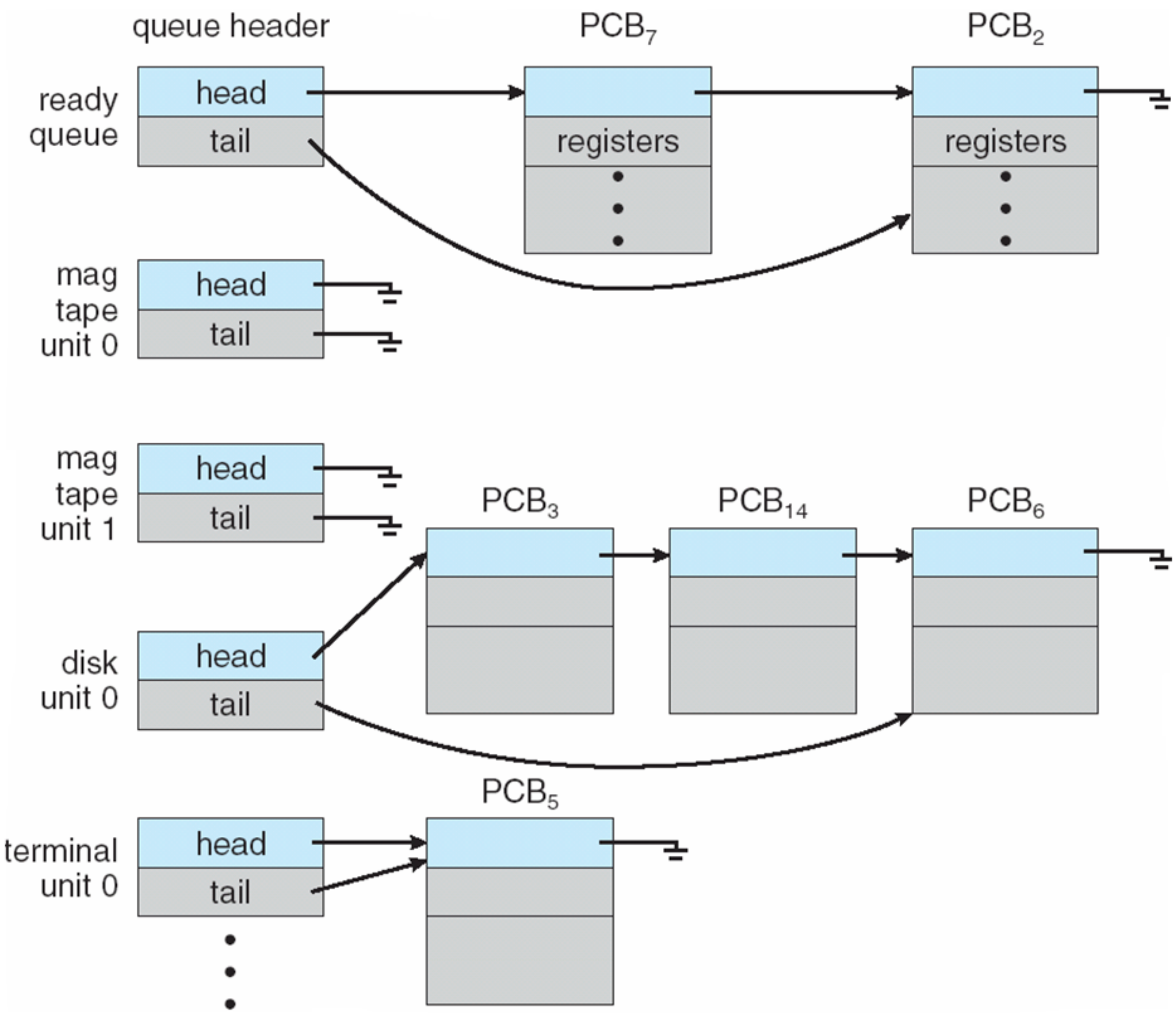
Context Switch

- 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 is represented (stored) in the PCB
- It begins with an interrupt and ends with RTI (ReTurn from Interrupt)
- Context-switch time is overhead (“waste”, that is), as the system does no useful work while switching
- Time dependent on hardware support and the size of the working set (things to save and restore)
  - Typically, from a single to a few hundred microseconds (10^{-6} sec)

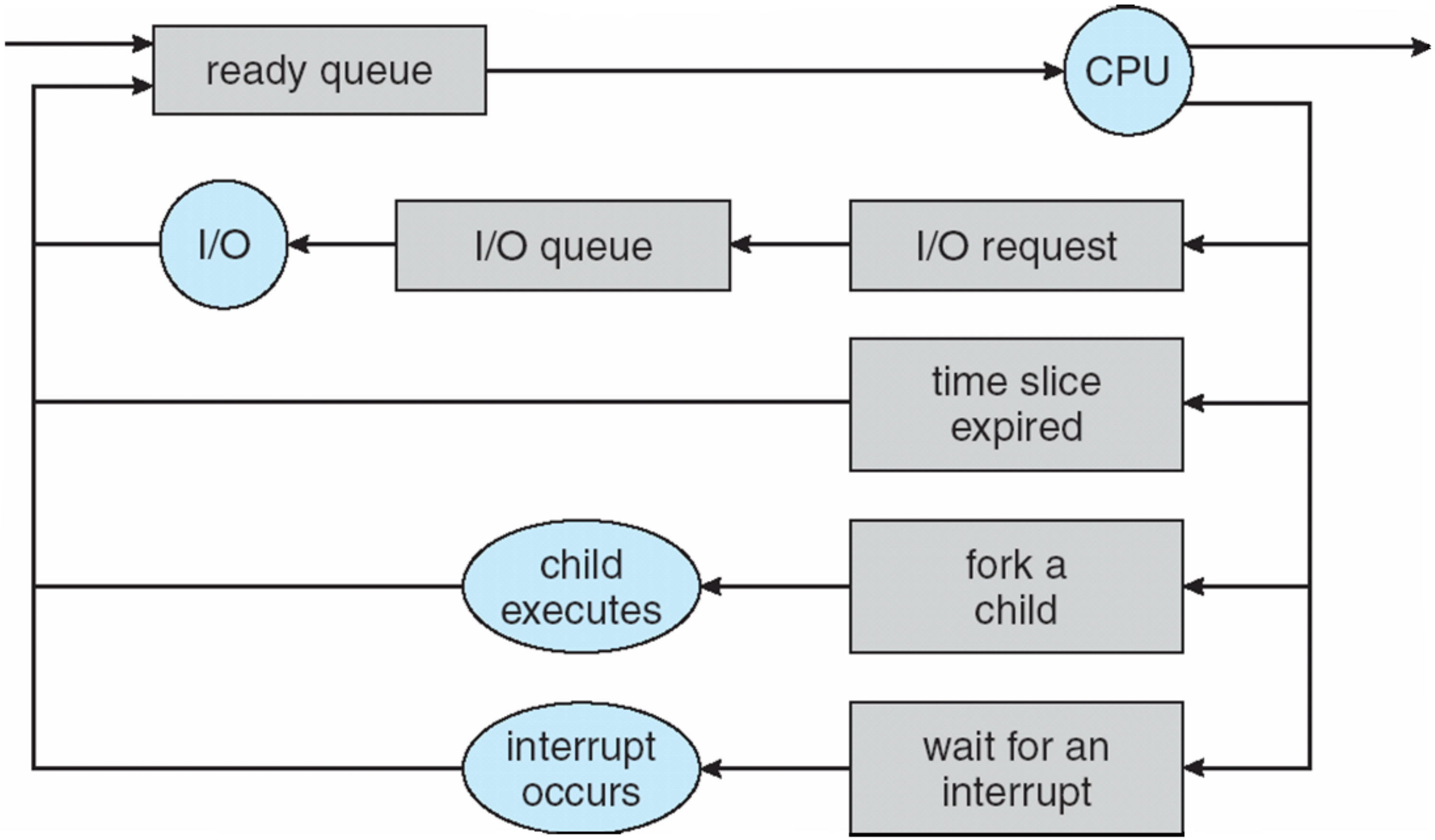
Process Scheduling Queues

- **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



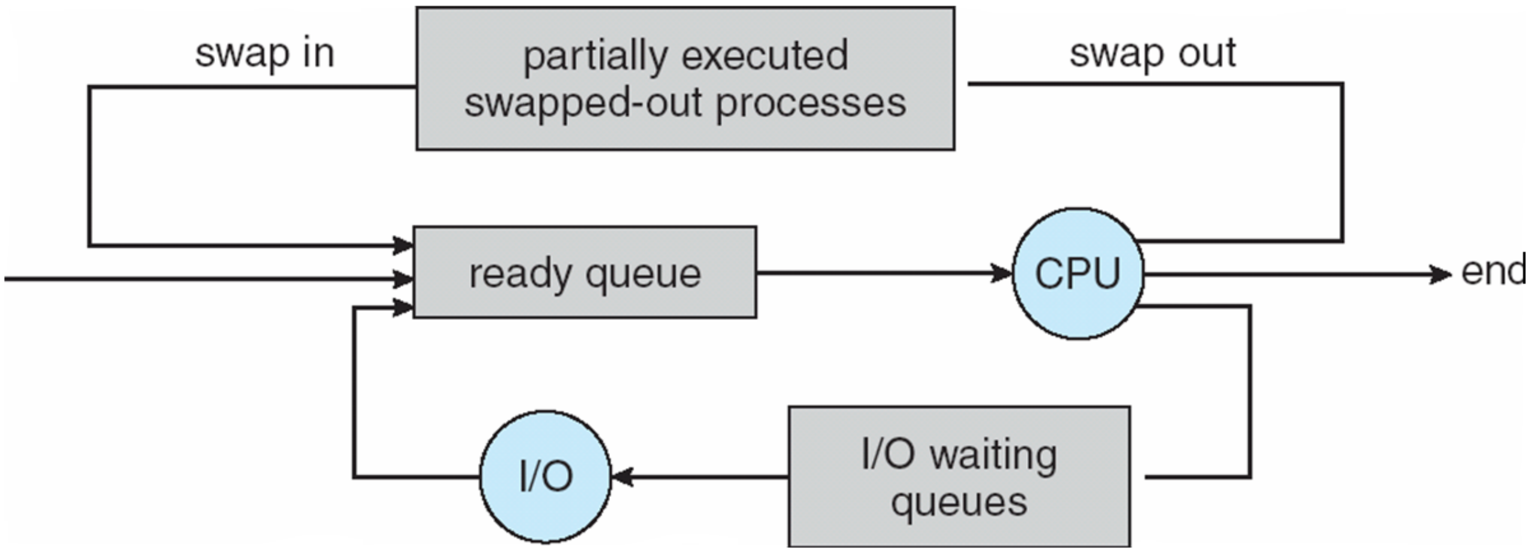
Representation of Process Scheduling



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

Addition of Medium Term Scheduling



Schedulers (Cont)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (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

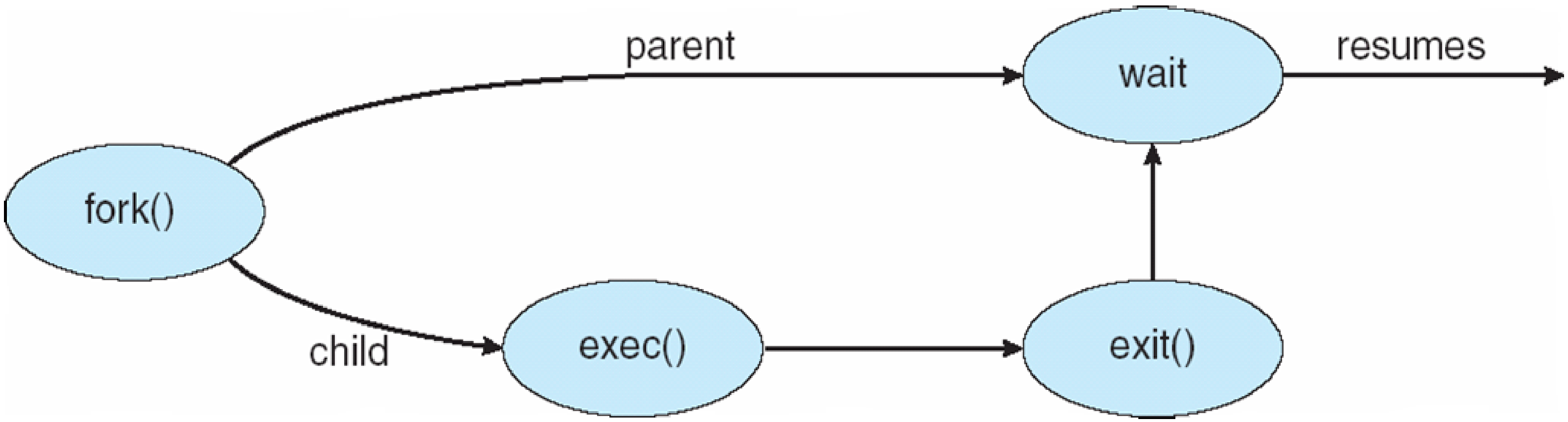
Process Creation

- **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
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process (a duplicate of the creating one)
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

Process Creation



C Program Forking Separate Process

```
int main()
{
pid_t  pid;
      /* fork another process */
      pid = fork();
      if (pid < 0) { /* error occurred */
              fprintf(stderr, "Fork Failed");
              exit(-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");
              exit(0);
      }
}
```

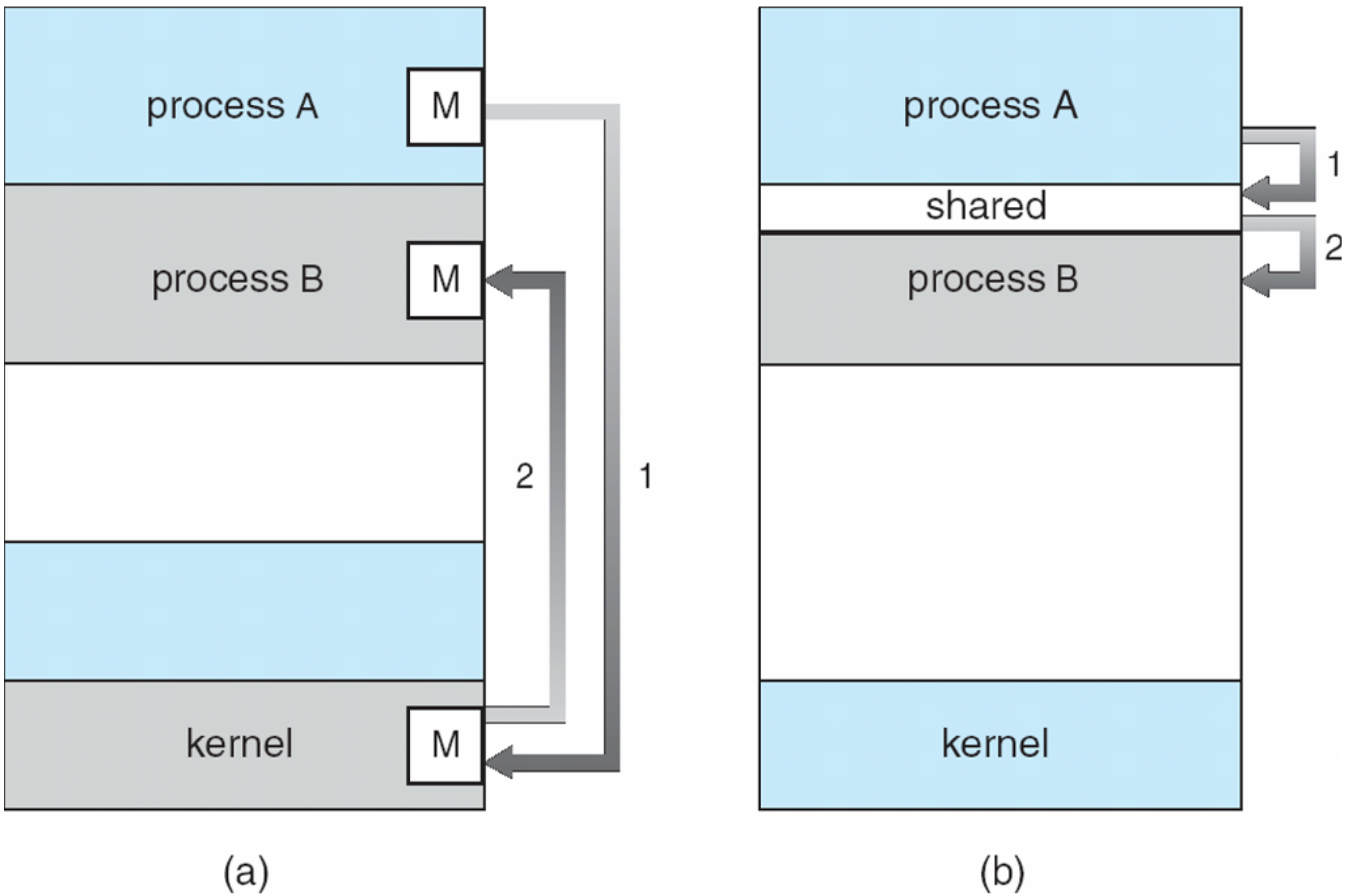
Process Termination

- 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 system do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**

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



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, P*roducer* process produces information that is consumed by a C*onsumer* process

Producer-Consumer Problem

- **Problem statement**
  - There are two processes: *Producer* and *Consumer* who have access to a shared buffer.
  - *Producer* can only write to the buffer.
  - *Consumer* can only read from the buffer.
  - **The problem** is how to synchronize them so that the following conditions are met:
    - *Producer* does not attempt to write when the buffer is full.
    - *Consumer* does not attempt to read when the buffer is empty.
    - *Consumer* does not attempt to read from the element of the buffer that is currently being written to by *Producer* (and vice versa).

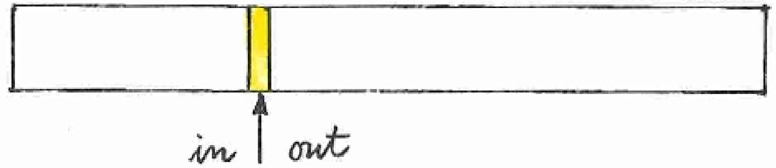
Producer-Consumer Problem

- There are two versions of the problem:
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size
    - in this case, the buffer is organized as a circular array

We are going to focus on the *bounded-buffer* version*.*

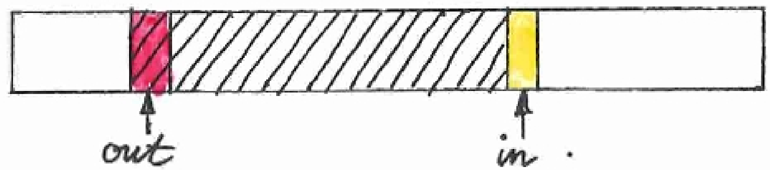
Buffer empty

(1)



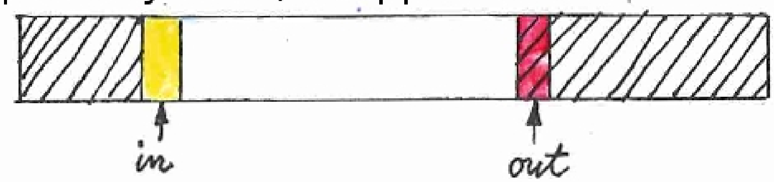
Buffer parially filled

(2)



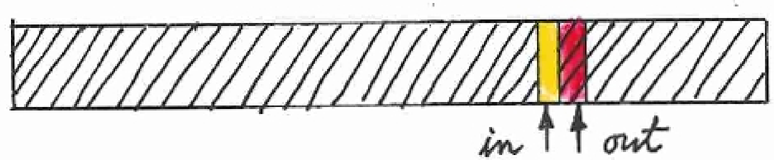
Buffer partially filled, wrapped around

(3)

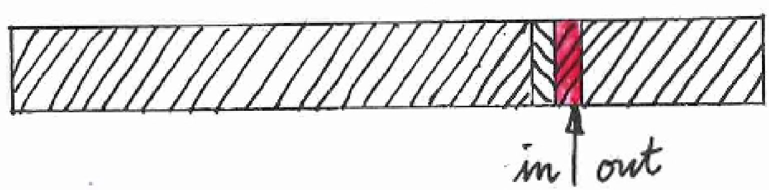


Buffer full

(4)



(5)



The above shows what would happen if the last ellement were allowed to be filled

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
public class BB_prod-cons {
int BUFFER_SIZE = 10;
Itemtype item1, item2; //details in class itemtype
itemtype [ ] buffer;
buffer = new itemtype [BUFFER_SIZE];
                int in = 0;
                int out = 0;
```

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

Bounded-Buffer – Producer

```
while (true) {
   /* Produce an item1 */

           . . .

    while (((in + 1) % BUFFER SIZE)  == out)

     ;   /* do nothing -- no more room in the buffer */

    // insert an item into the buffer

    buffer[in] = item1;

    in = (in + 1) % BUFFER SIZE;

 }
```

Bounded Buffer – Consumer

```
while (true) {
     while (in == out)
           ; // do nothing – no unconsumed items in
the buffer
     // remove an item from the buffer
     item2 = buffer[out];
     out = (out + 1) % BUFFER SIZE;
/* Consume an item2 */
       . . .
 }
}
```

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., logical properties)

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

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

End of Chapter 3