Process (contd.)

Indranil Sen Gupta (odd section)
and Mainack Mondal (even section)
CS39002

Spring 2019-20

So far on processes

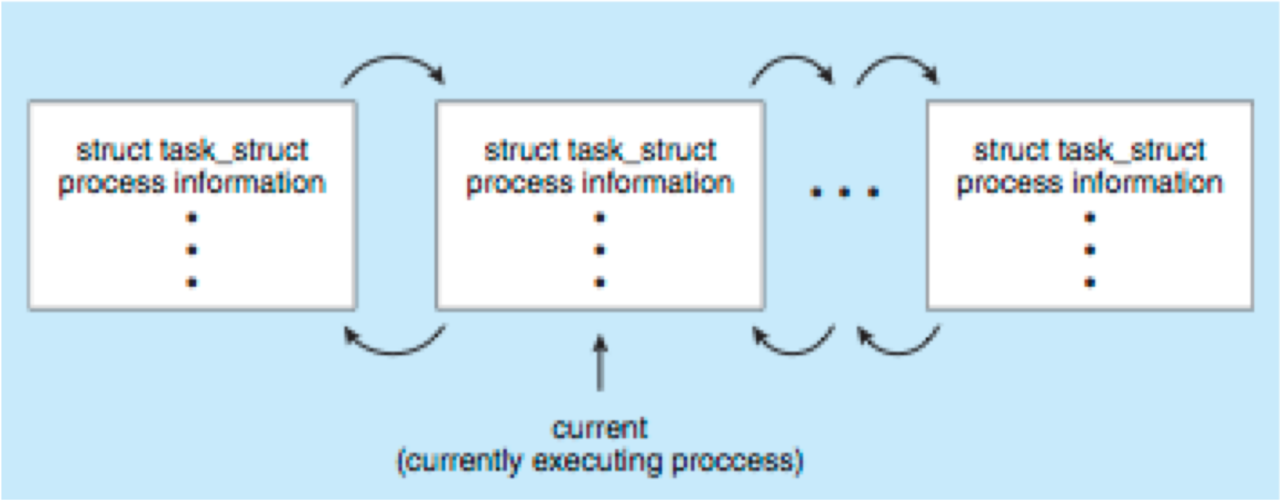
- What is a process?
- Structure of a process
- Process states
- Process control block
- Context switch

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 pro */
```

Doubly linked list

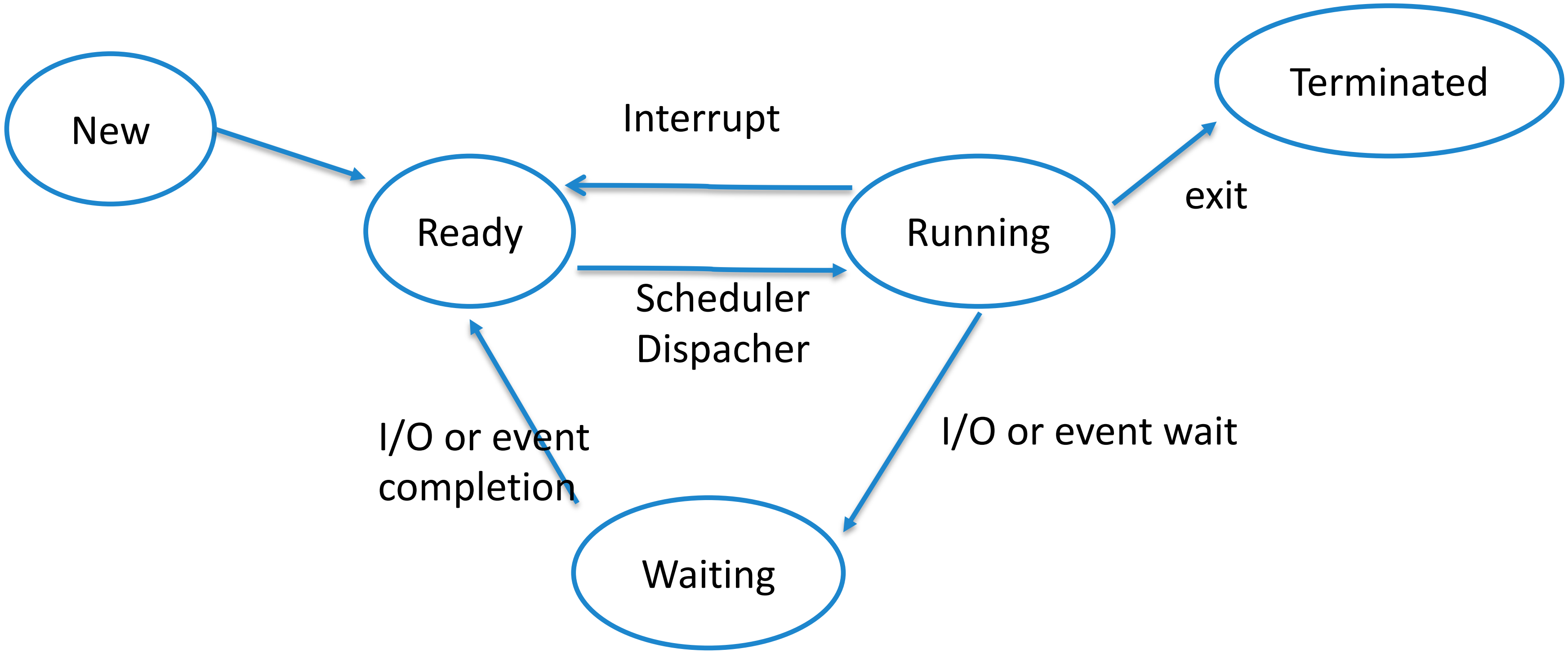


Process scheduling

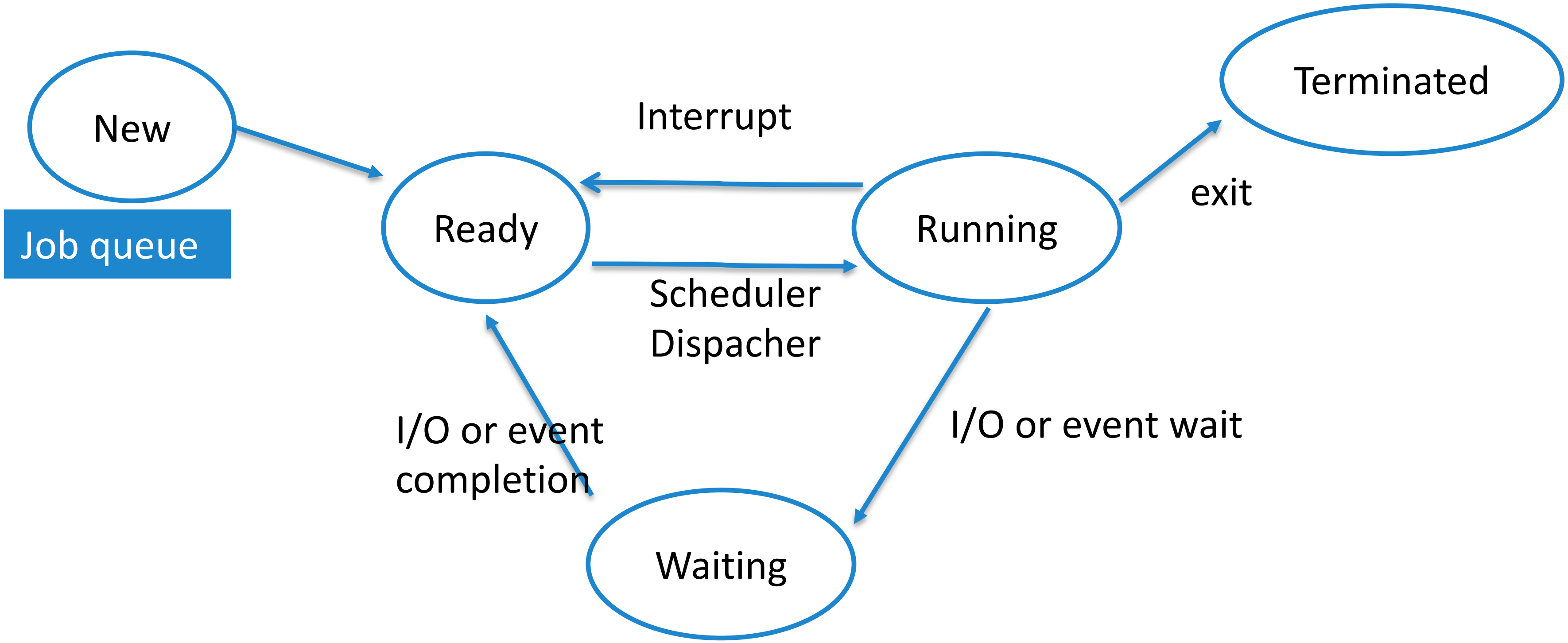
Process scheduling

- The process scheduler selects an available process for execution on the CPU
    - Dispatcher: The kernel process that assigns CPU to a process

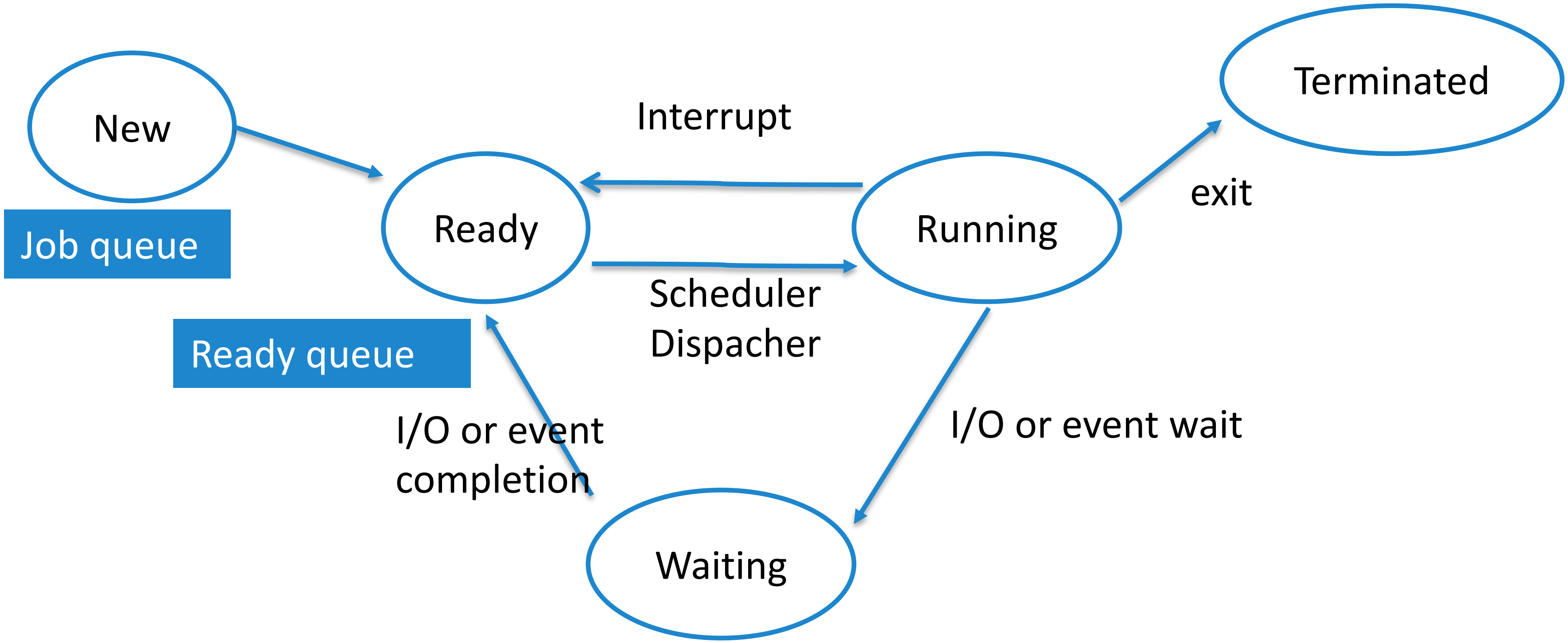
Recap: Process state diagram



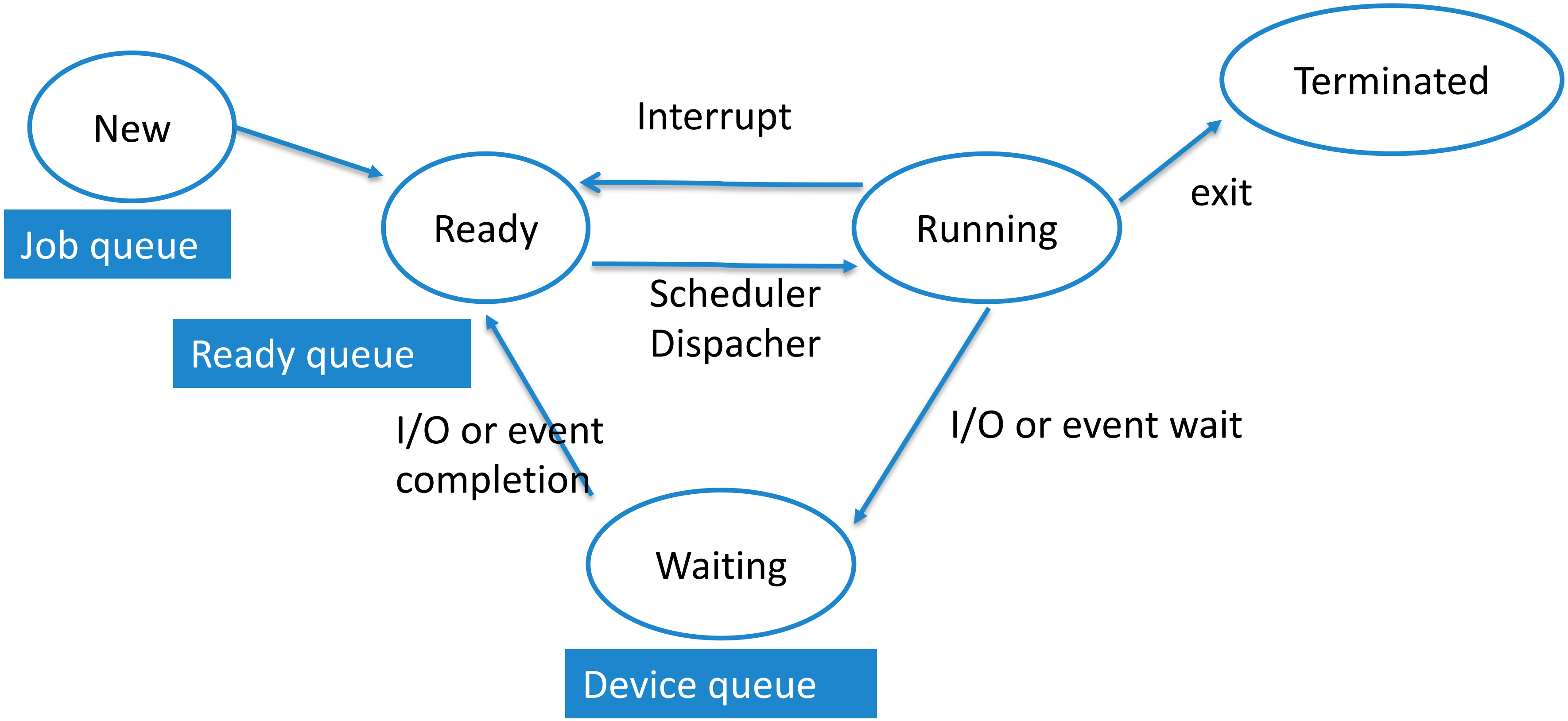
Recap: Process state diagram



Recap: Process state diagram



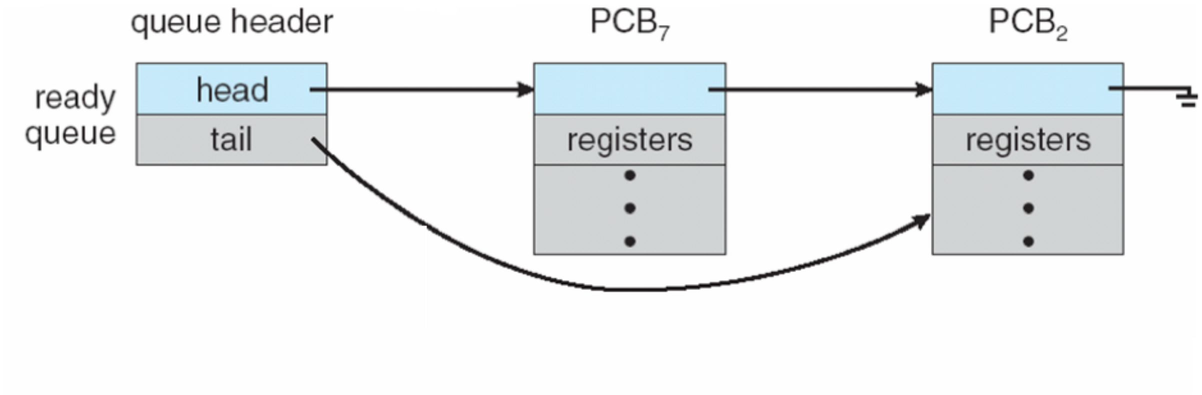
Recap: Process state diagram



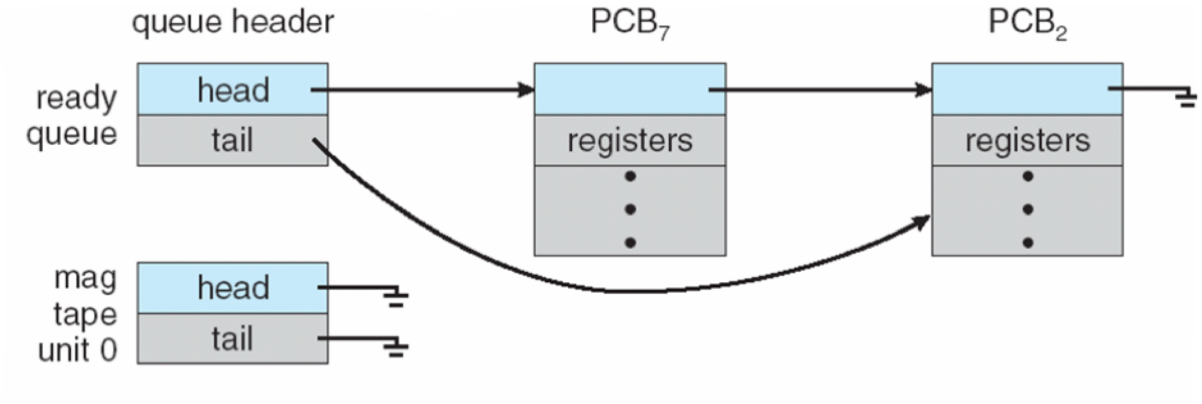
Process scheduling

- Several scheduling queues exist in OS
  - A PCB is linked to one of the queues at any given tome
  - The PCBs in a queue are connected as a linked list

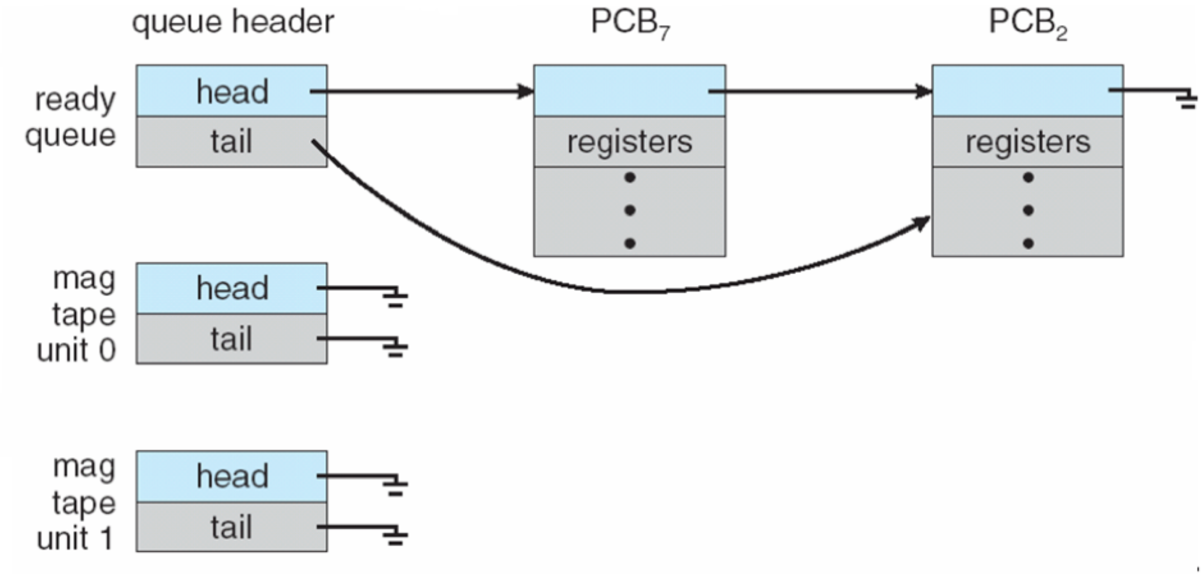
Structure of process queues



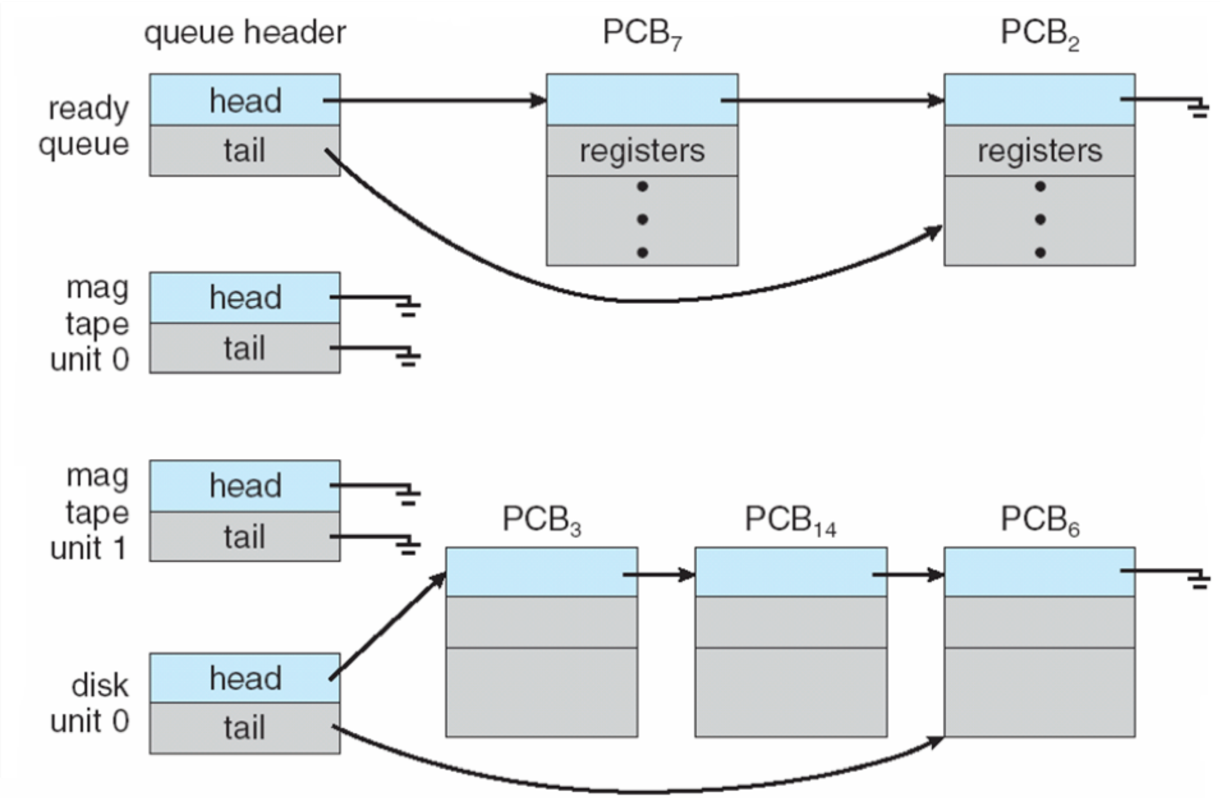
Structure of process queues



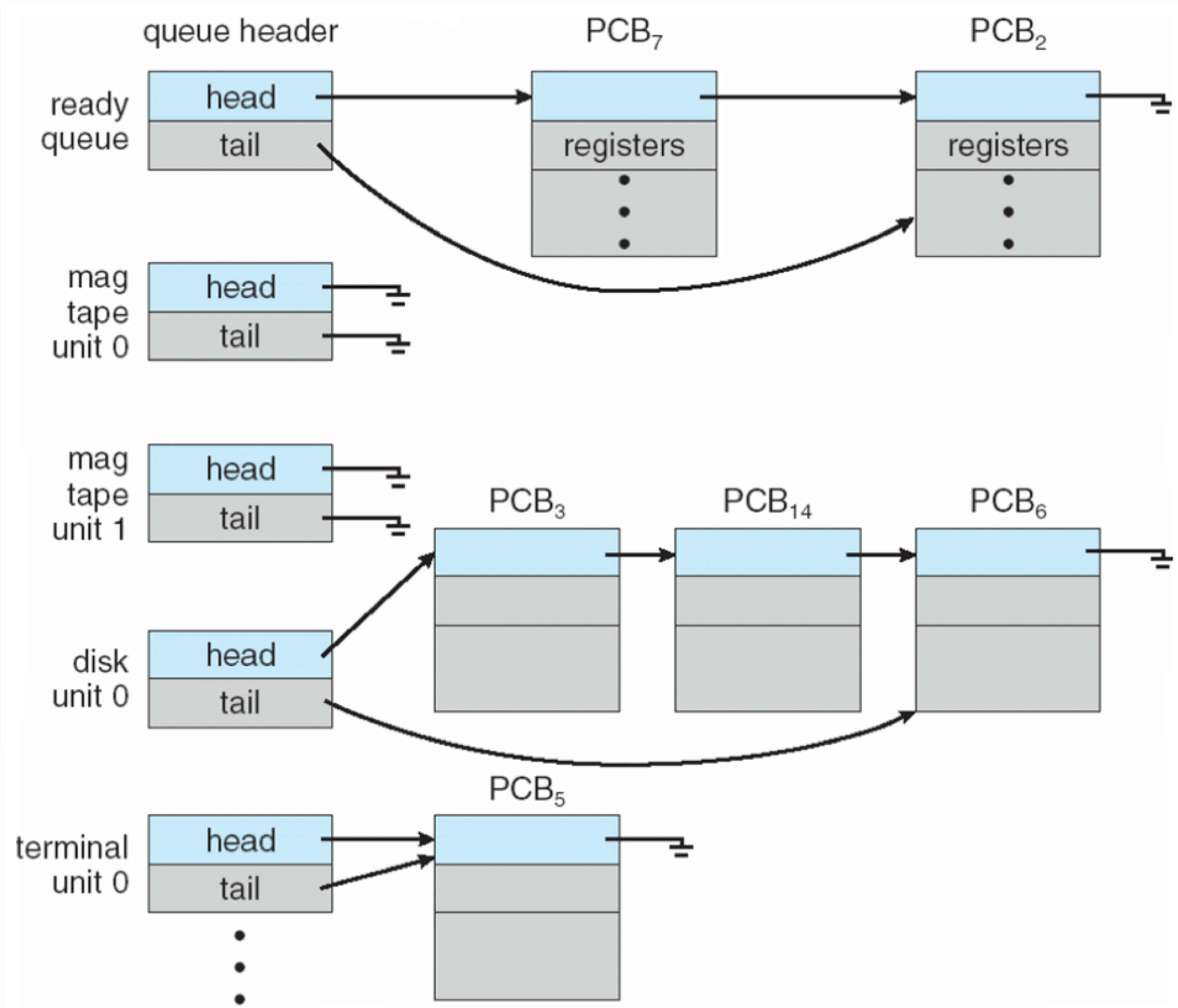
Structure of process queues



Structure of process queues



Structure of process queues



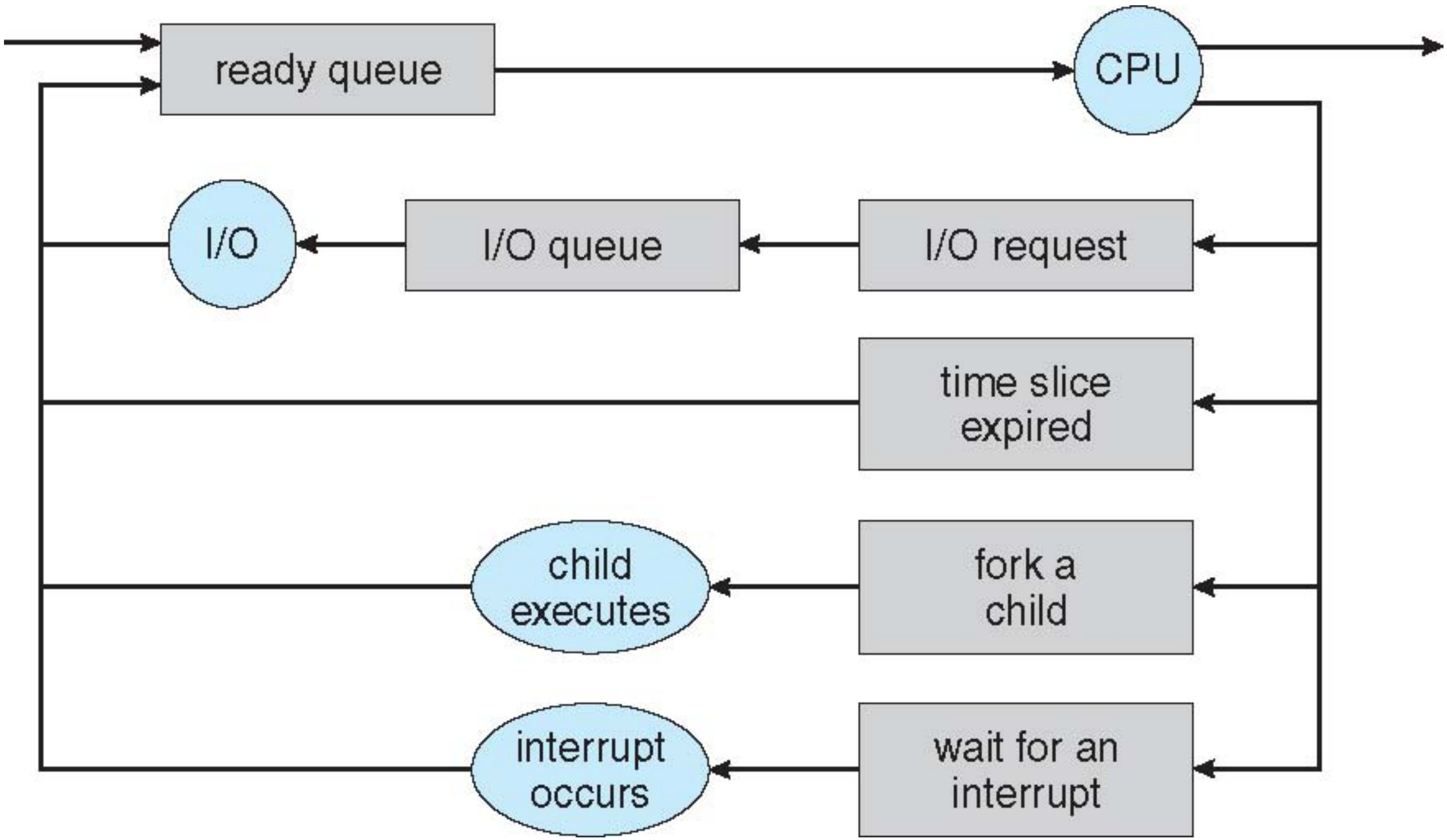
Characteristics of process queues

- Each I/O device has its own device queue
    - Each event also has its own queue

- Process scheduling can be represented as a queueing diagram
    - Queueing diagram represents queues, resources, flows
    - We will discuss actual scheduling algorithm later

Representation of process scheduling

Representation of process scheduling



Operations on processes

Process creation

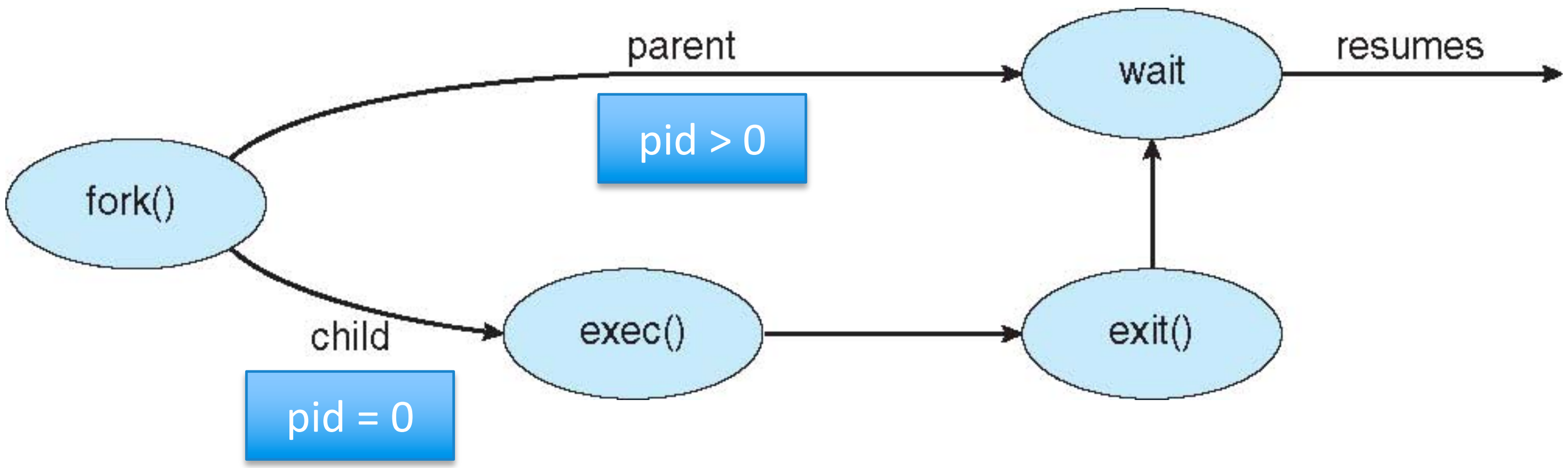
- During execution a process may create several new processes
  - Each process has a unique process identifier (pid)
  - Other than the first process (init), all other processes are created by fork system call
  - Parent process create children processes, which, in turn create other processes, forming a tree of processes

Process creation (contd.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX example
  - fork() : creates a new process
  - exec(): replace new process’s memory with new code

Process creation (contd.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX example
  - fork() : creates a new process
  - exec(): replace new process's memory with new code



Process creation example

```
#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 condition

```
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) {
```

child process

```
       execlp("/bin/ls","ls",NULL);
    }
    else {
```

parent process

```
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Process termination

- A child process executes last statement
  - exit() call for deleting the process
  - return status data from child to parent via wait()
  - Deallocate the resources

Process termination

- A child process executes last statement
  - exit() call for deleting the process
  - return status data from child to parent via wait()
  - Deallocate the resources

<u>Child process</u>

.

.

exit(2) // Exit with status code

<u>Parent process</u>

pid_t pid;

Int status;

.

pid = wait (&status) // pid of terminated child

Process termination: Corner cases

- In some OS
  - All child must terminate when a process terminates
  - Cascading termination: All children, grandchildren etc. must be terminated
  - OS takes care of this cascade

- Combinations of exit() and wait()
  - If no parent is waiting then zombie process
  - If parent terminated without invoking wait then orphan process

Zombie and orphan process

- Zombie process
  - A process that has terminated, but who parent had not not yet called wait()
  - All processes move to this state when they terminate and remain there until parent calls wait()
  - Entry in process table removed only after calling wait()

Zombie and orphan process

- Zombie process
  - A process that has terminated, but who parent had not not yet called wait()
  - All processes move to this state when they terminate and remain there until parent calls wait()
  - Entry in process table removed only after calling wait()

- Orphan process
  - parent terminated without invoking wait
  - Immediately “init” process assigned as parent
  - “init” periodically invokes wait()

Inter-process communication (IPC)

- Processes executing concurrently in OS may be independent or cooperating
- Cooperating process
  - Affect or be affected by other processes, e.g., sharing data

Inter-process communication (IPC)

- Processes executing concurrently in OS may be independent or cooperating
- Cooperating process
  - Affect or be affected by other processes, e.g., sharing data
  - Can share information
  - Speed-up in computation
  - Design can be modular

Inter-process communication (IPC)

- Processes executing concurrently in OS may be independent or cooperating
- Cooperating process
  - Affect or be affected by other processes, e.g., sharing data
  - Can share information
  - Speed-up in computation
  - Design can be modular

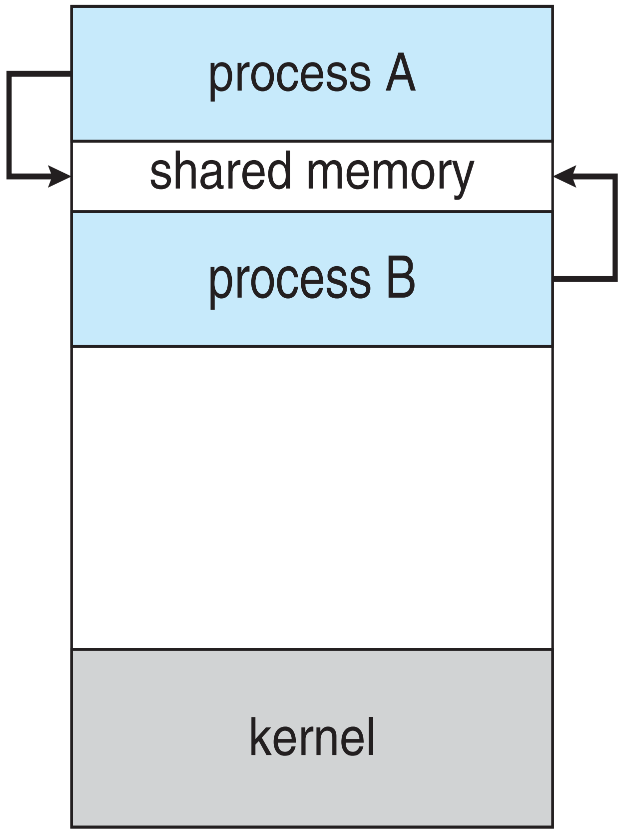
- Cooperating processes need IPC
  - shared memory
  - Message passing

Inter-process communication (IPC)

- Ways to do IPC
  - way 1: shared memory - shmget(), shmcat(), shmaddr(), shmat(), shmdt(), shmctl()
  - way 2: message passing (pipe) - pipe(), read(), write(), close()
  - way 3: message passing (named pipe) - mkfifo(), read(), write(), close()
  - way 4: Over network - RPC or Remote Procedure Call, sockets

Shared memory system

Schematic for shared memory



Let’s check the function calls

```
char *myseg;

key_t key; int shmid;

key = 235; // some unique id

shmid = shmget(key, 250, IPC_CREAT | 0666);

myseg = shmat(shmid, NULL, 0);
.
.
shmdt(myseg);
.
.
shmctl(shmid, IPC_RMID, NULL);
```

Let’s check the function calls

```
char *myseg;

key_t key; int shmid;

key = 235; // some unique id

shmid = shmget(key, 250, IPC_CREAT | 0666); // create shared memory segment

myseg = shmat(shmid, NULL, 0);
.
.
shmdt(myseg);
.
.
shmctl(shmid, IPC_RMID, NULL);
```

Let’s check the function calls

```
char *myseg;

key_t key; int shmid;

key = 235; // some unique id

shmid = shmget(key, 250, IPC_CREAT | 0666); // create shared memory segment

myseg = shmat(shmid, NULL, 0); // attach the segment to the
                               // address space of this process
.
.
shmdt(myseg);
.
.
shmctl(shmid, IPC_RMID, NULL);
```

Let's check the function calls

```
char *myseg;

key_t key; int shmid;

key = 235; // some unique id

shmid = shmget(key, 250, IPC_CREAT | 0666); // create shared memory segment

myseg = shmat(shmid, NULL, 0); // attach the segment to the
                                // address space of this process
.
.
shmdt(myseg);                   // detach the segment from the address  space
.
.
shmctl(shmid, IPC_RMID, NULL);
```

Let’s check the function calls

```
char *myseg;

key_t key; int shmid;

key = 235; // some unique id

shmid = shmget(key, 250, IPC_CREAT | 0666); // create shared memory segment

myseg = shmat(shmid, NULL, 0); // attach the segment to the
                               // address space of this process
.
.
shmdt(myseg);                  // detach the segment from the address  space
.
.
shmctl(shmid, IPC_RMID, NULL); // mark the segment to be destroyed
```

Producer consumer problem

- A producer process produces information that is consumed by the consumer process
  - Compiler produces assembly code consumed by assembler
  - Program produces lines to print, print spool consumes
  - The information is read/write from a buffer

Producer consumer problem

- A producer process produces information that is consumed by the consumer process
  - Compiler produces assembly code consumed by assembler
  - Program produces lines to print, print spool consumes
  - The information is read/write from a buffer
- Two variants
  - Bounded buffer
  - Unbounded buffer

Producer consumer problem

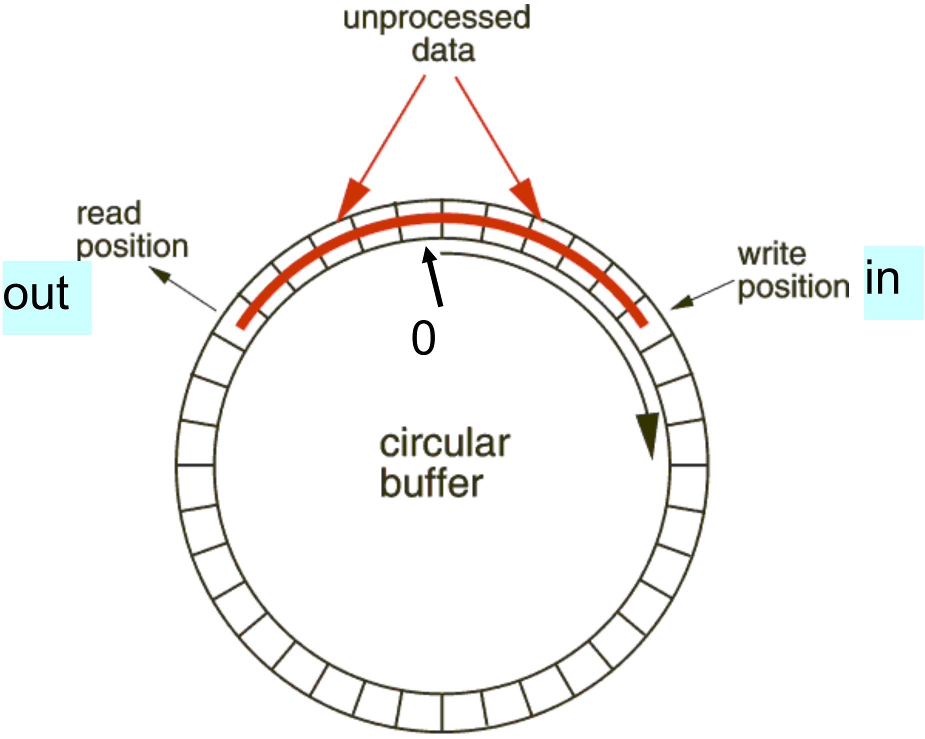
- A producer process produces information that is consumed by the consumer process
  - Compiler produces assembly code consumed by assembler
  - Program produces lines to print, print spool consumes
  - The information is read/write from a buffer

- Two variants
  - Bounded buffer
  - Unbounded buffer
  - Bounded buffer : producer waits when buffer is full, consumer waits when buffer is empty

Producer consumer solution with bounded buffer

- Shared data: implemented as a circular array

```
#define BUFFER_SIZE 10
typedef struct {
   . . . // information to be shared
} item;

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



Key ideas

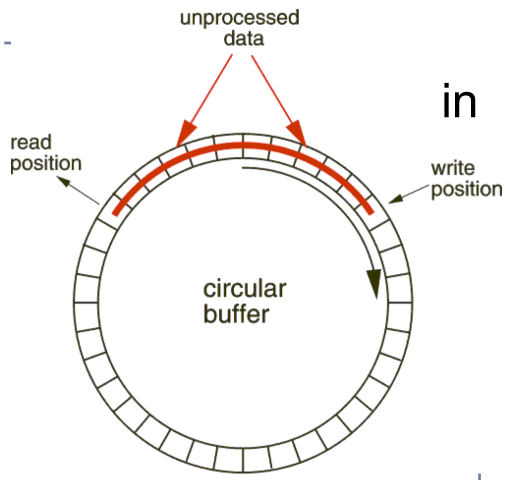
- Circular buffer
  - Index in: the next position to write to
  - Index out: the next position to read from
- To check buffer full or empty:
  - Buffer empty: in==out
  - Buffer full: in+1 % BUFFER_SIZE == out
    - Why ? There is still one slot left …

Pseudo code

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

      while (( (in + 1) % BUFFER_SIZE) == out)

     ;   /* do nothing -- no free buffers */

    buffer[in] = newProducedItem;

    in = (in + 1) % BUFFER SIZE;

   }
```

out

in



Producer

```
   while (true) {
        while (in == out)
               ; // do nothing -- nothing to consume

        // remove an item from the buffer
        itemToConsume = buffer[out];
        out = (out + 1) % BUFFER SIZE;
        return itemToComsume;
   }
```

Consumer

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

Better utilization of buffer space

- Circular buffer
- Suppose that we want to use all buffer space:
  - an integer count: the number of filled buffers
  - Initially, count is set to 0.
  - incremented by producer after it produces a new buffer
  - decremented by consumer after it consumes a buffer.

Better utilization of buffer space: Pseudo code

Producer

```
while (true) {
    /*  produce an item
        and put in nextProduced  */
    while (count == BUFFER_SIZE)
     ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

Consumer

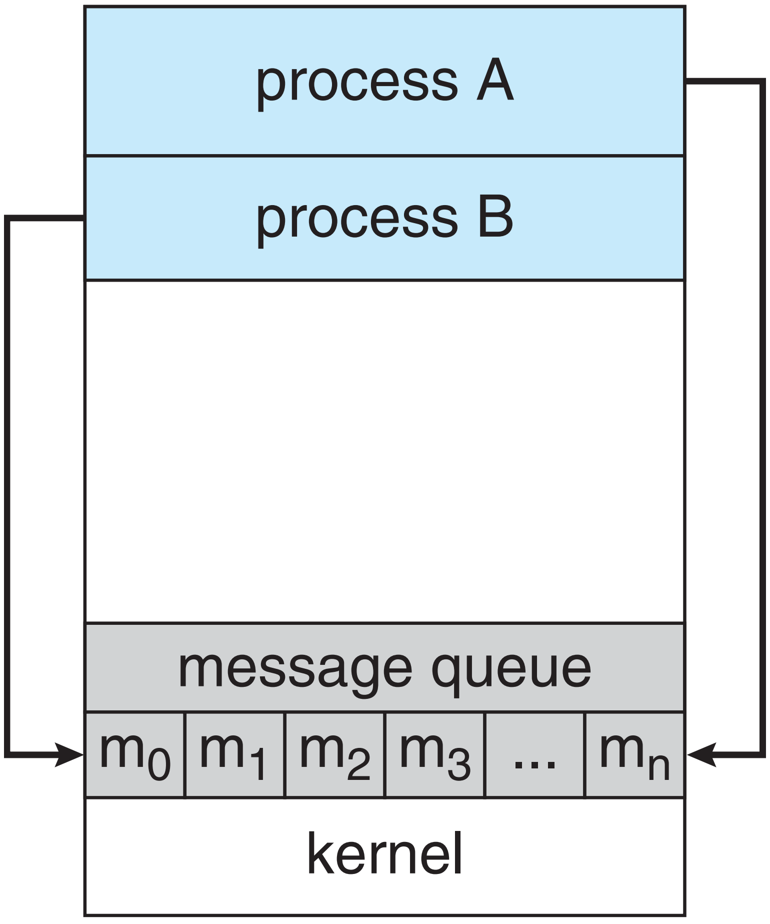
```
while (true)  {
   while (count == 0)
      ; // do nothing
   nextConsumed =  buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  count--;

  /*  consume the item in
     nextConsumed */
}
```

Message passing system

Basics of 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

Communication model



Ways for message passing

- Pipes
- Named pipes
- Covered in last class
    - Also in the assignments

Finally for communication of two processes over network

- Sockets API
- Remote procedure call

Summary

- What is a process?
  - Structure of a process
  - Process states
  - Process control block
  - Context switch
- Why is process scheduling necessary?
  - Ready queues, event queues, queueing diagram
- How does two processes talk?
  - Shared memory, pipe, named pipe