Process (contd.)

Saptarshi Ghosh and Mainack Mondal

CS39002

Spring 2020-21

So far on processes

- What is a process?
- Structure of a process in memory
- Process control block (PCB)
- Process states
- Context switch

Process queues (for scheduling)

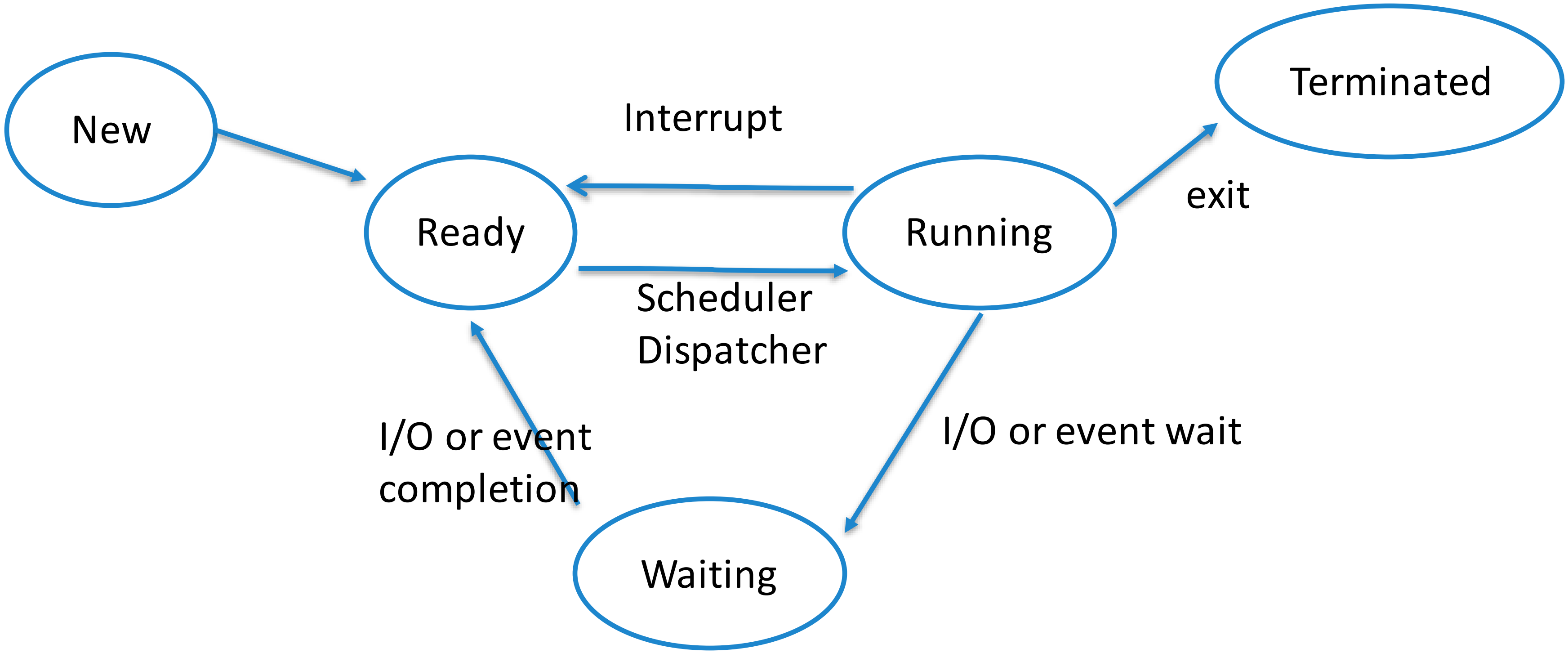
Process queues for scheduling

- 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

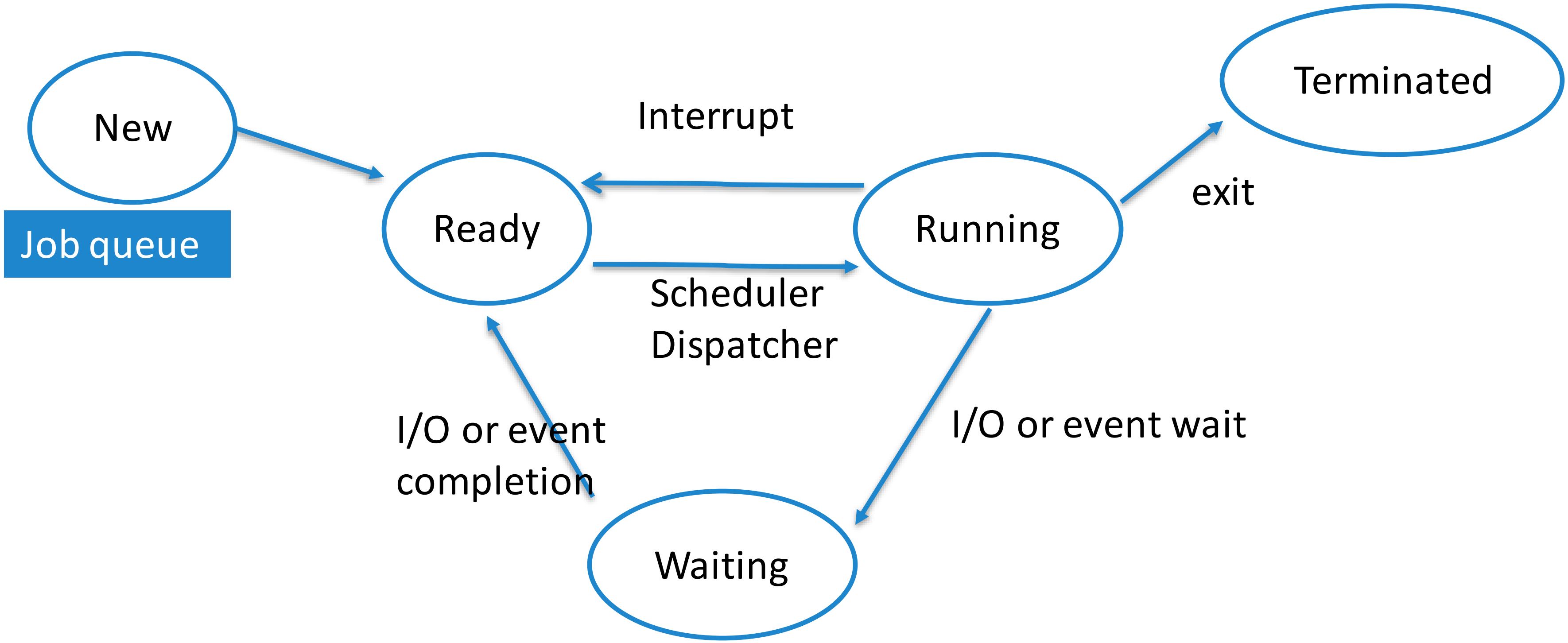
Characteristics of process queues

- Each I/O device has its own device queue
  - Contains (PCBs of) processes waiting for this device
- Each event also has its own queue
  - Contains processes waiting for this event
- Process scheduling can be represented as a queueing diagram
  - Queueing diagram represents queues, resources, flows
  - We will discuss actual scheduling algorithms later

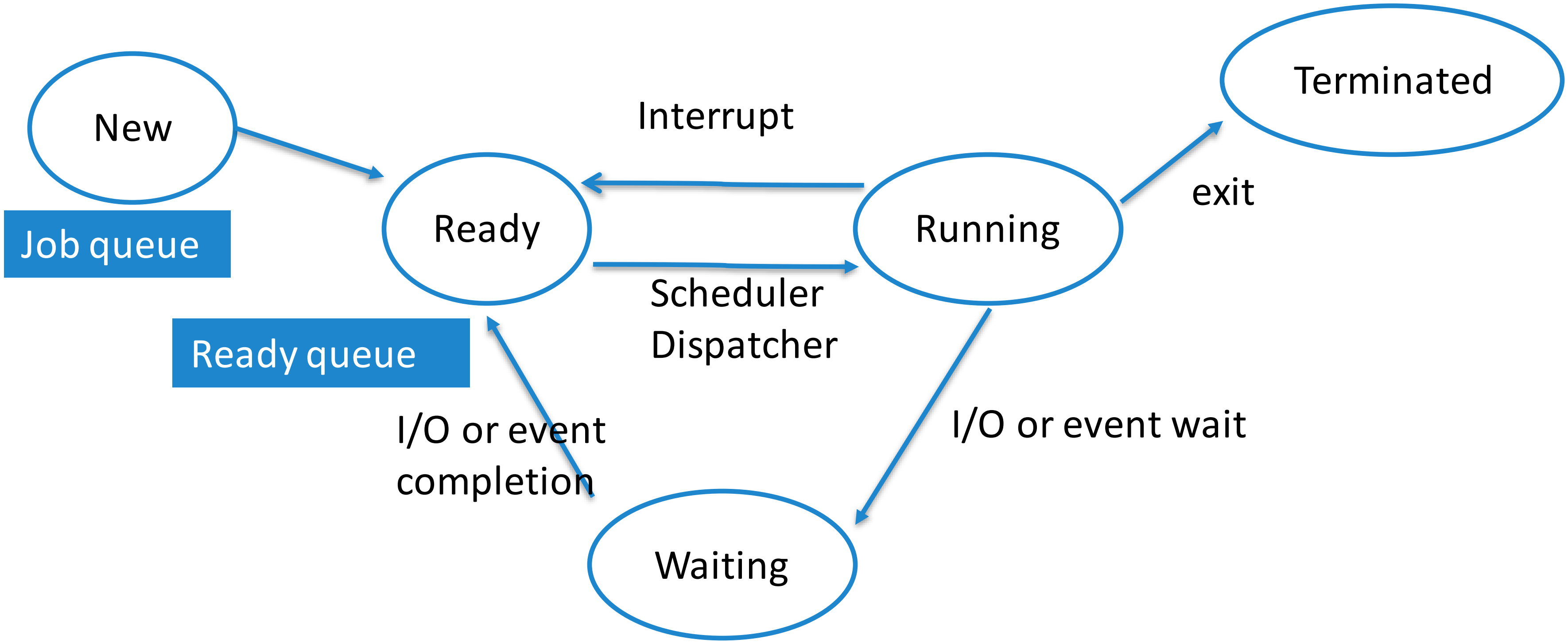
Recap: Process state diagram



Recap: Process state diagram

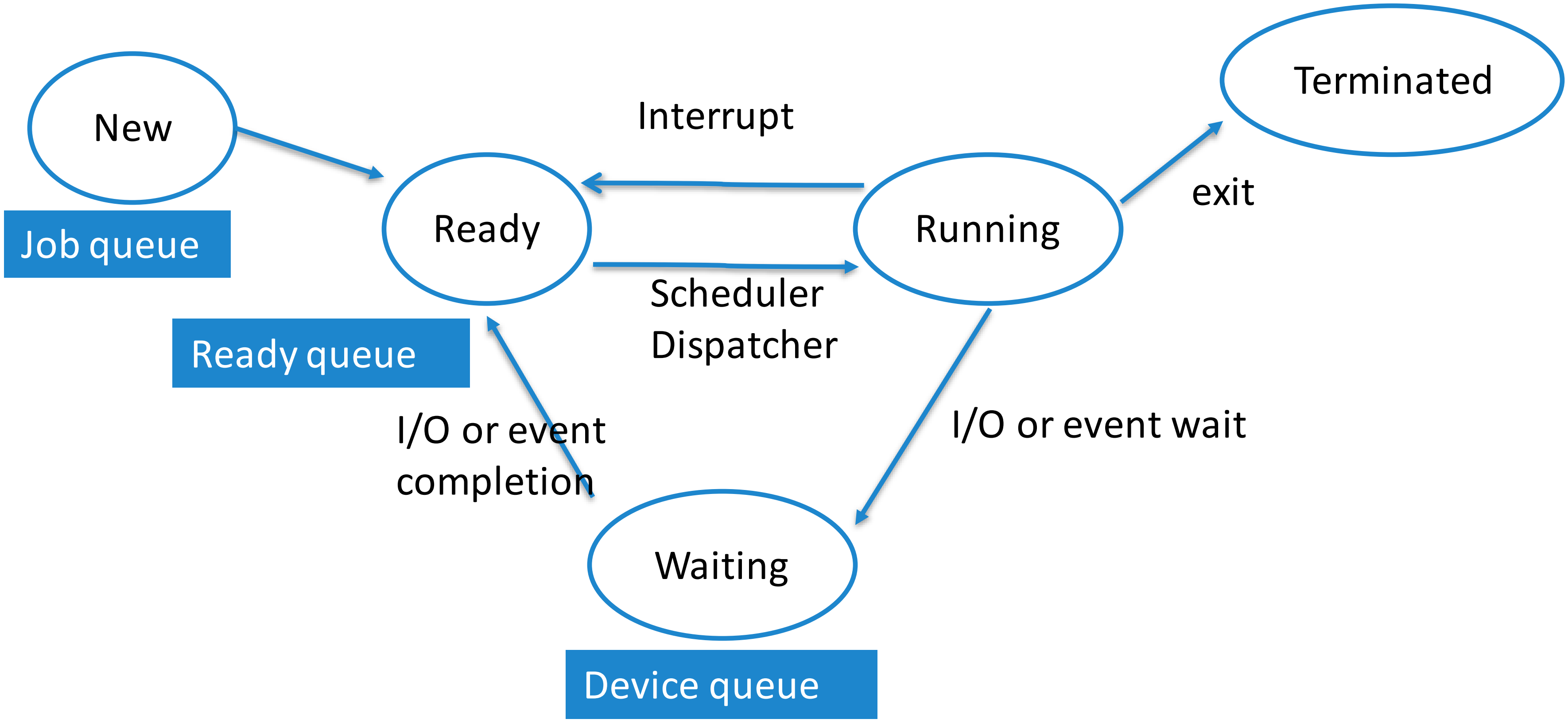


Recap: Process state diagram



- The process scheduler selects a process from the ready queue for execution on the CPU
- Dispatcher: The kernel process that assigns the CPU to a process selected by the scheduler

Recap: Process state diagram

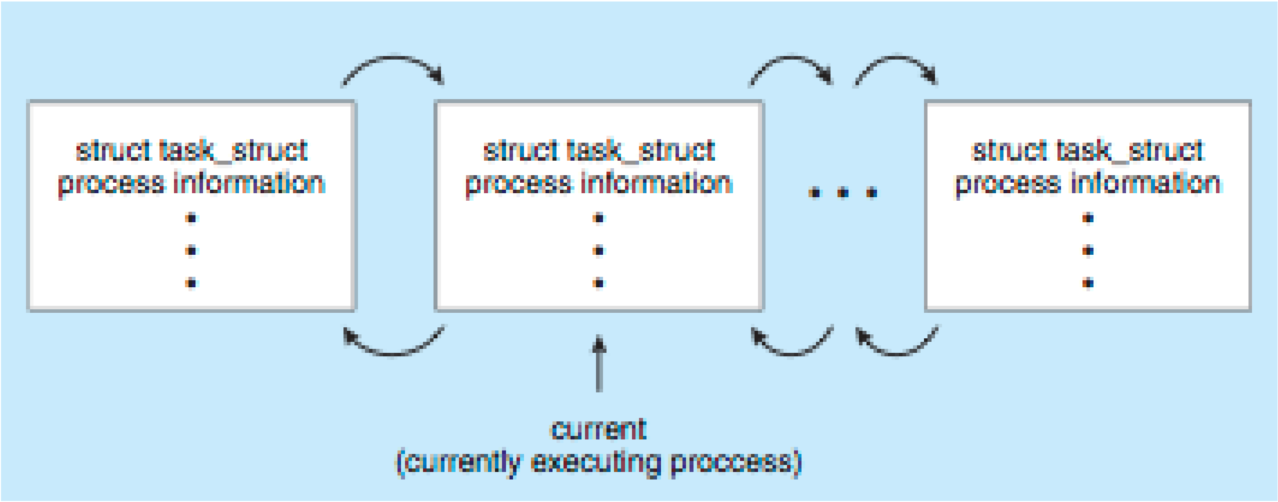


Process queue representation in Linux

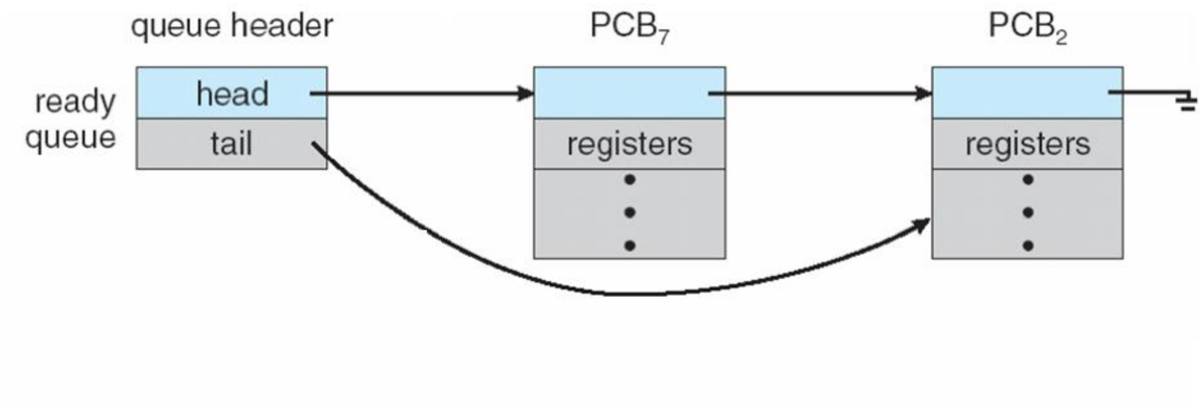
Represented by the C structure task_struct

```
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 task_struct *children; // this process's children
struct files_struct *files; // list of open files
struct mm_struct *mm; // address space of this pro
```

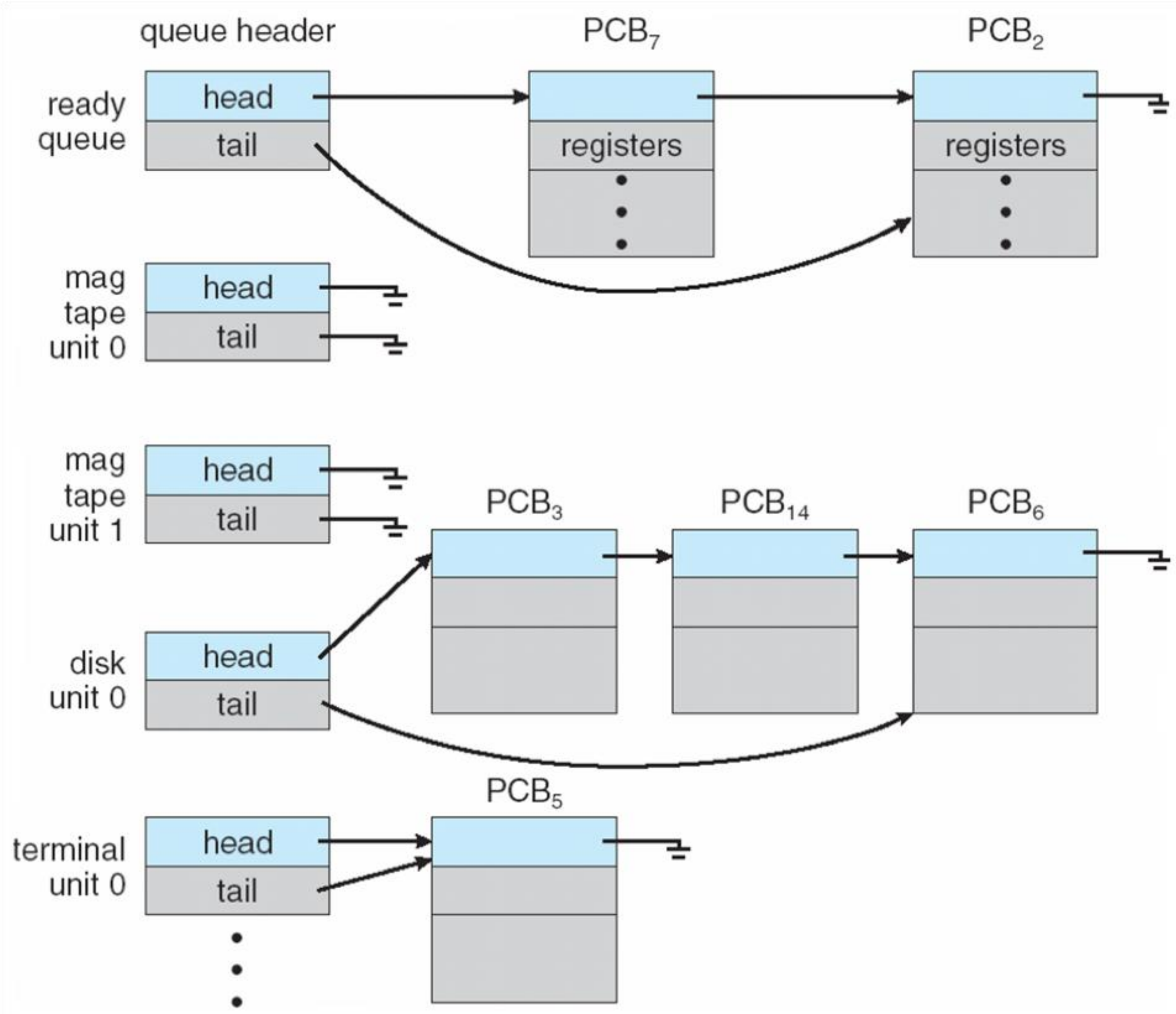
Doubly linked list



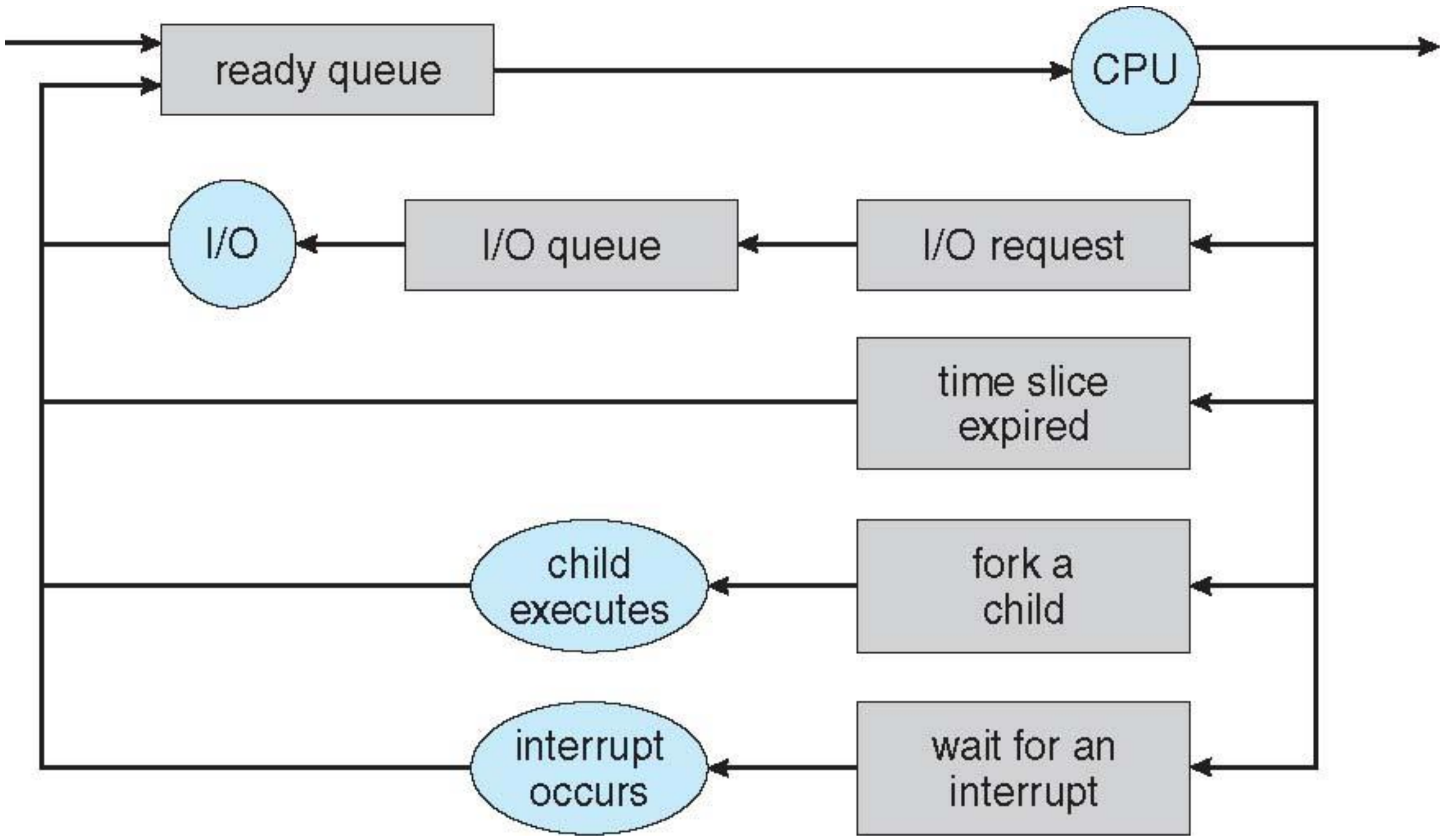
Structure of process queues



Structure of process queues



Representation of process scheduling



Operations on processes

Process creation

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

Process creation (contd.)

- Address space
  - Initially child is duplicate of parent
  - Child can later have a different program loaded into it

- In UNIX / Linux
  - fork() : creates a new process
  - exec(): replace new process’s memory with new code

fork( ) system call

- If **fork()** returns a negative value, the creation of a child process was unsuccessful.

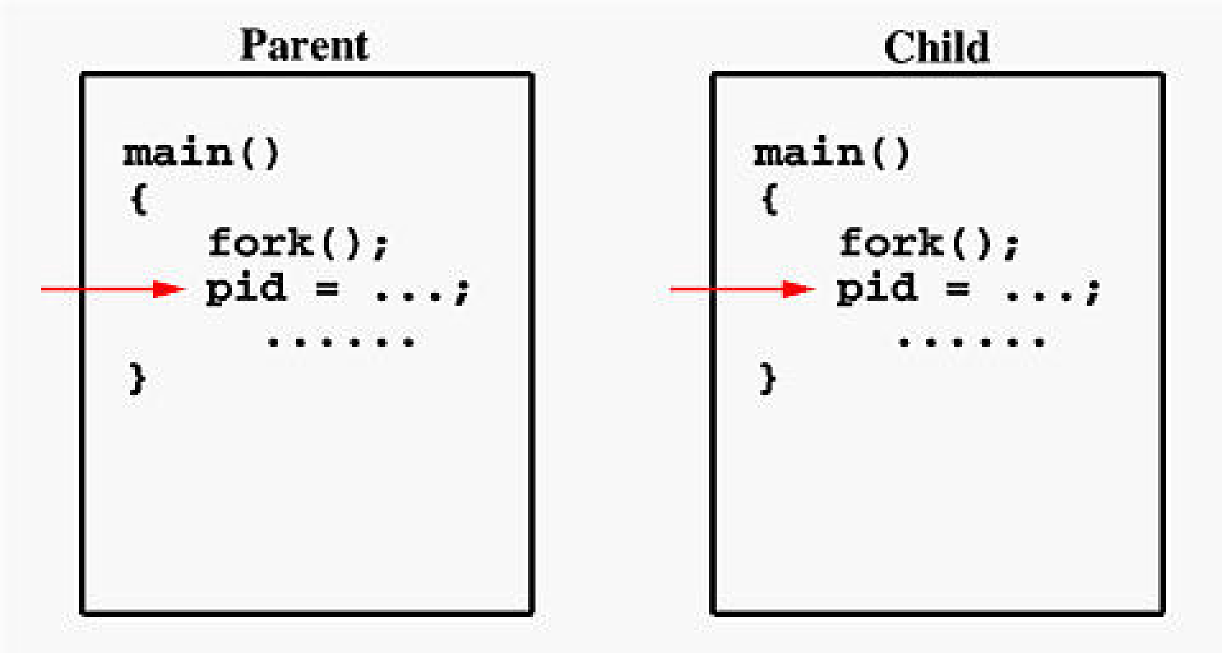
- **fork()** returns a zero to the newly created child process.

- **fork()** returns a positive value, the ***process ID*** of the child process, to the parent.

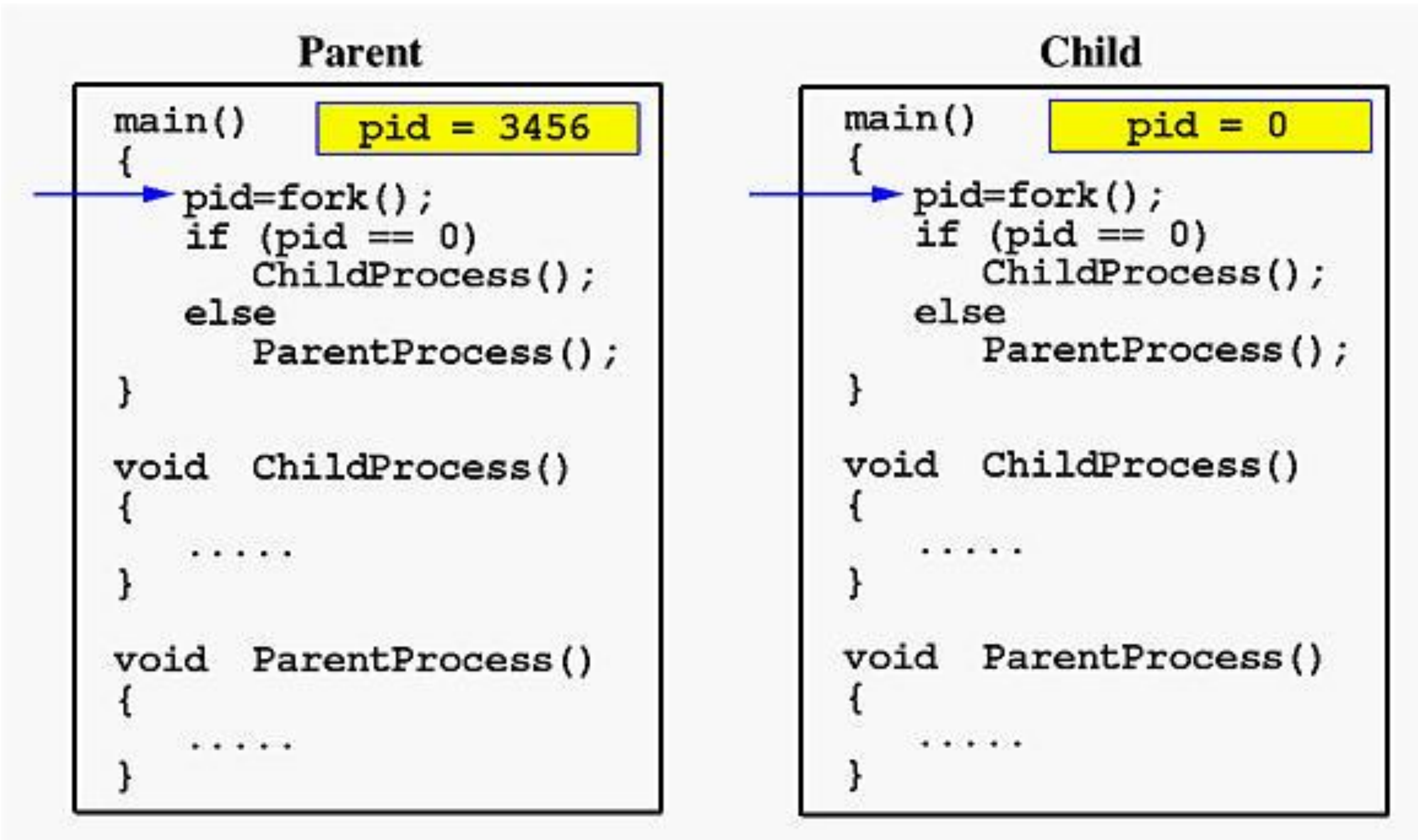
fork( ) system call

- If the call to **fork()** is executed successfully, the kernel will make an identical copy of the parent's address space, for use by the child.

- Both processes will start their execution at the next statement following the **fork()** call.



Make a child do something different than the parent – method 1



Make a child do something different than the parent – method 2

- exec( ) system call and its variants
  - Replaces the calling process (that calls exec) with another program
  - Calling process terminated; the specified program is loaded in the same address space and then executed
  - New process image has same process ID (pid), same file descriptors, etc
  - Several variants – execl, execle, execlp, execv, execve, execvp

Process waiting for a child

- A parent process can wait for a child to complete
  - Parent indicates to the kernel that it wants to wait, using the wait() system call
  - Parent execution is suspended; will be "ready" only after child terminates
  - If a child has already terminated, then the call returns immediately. Otherwise, block until a child terminates

Process waiting for a child

- A parent process can wait for a child to complete
  - Parent indicates to the kernel that it wants to wait, using the wait() system call
  - Parent execution is suspended; will be "ready" only after child terminates
  - If a child has already terminated, then the call returns immediately. Otherwise, block until a child terminates

- Two variants
  - wait() - suspends execution of the current process until one of its children terminates
  - waitpid(pid) - suspends execution of the current process until the child specified by *pid* terminates

Process termination

- A child process executes last statement
  - exit() call for deleting the process
  - Return status data to parent via wait() if any parent is waiting for the termination of this process
  - Deallocate the resources

Process termination

- A child process executes last statement
  - exit() call for deleting the process
  - Return status data to parent via wait() if any parent is waiting for the termination of this process
  - 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 creation and exec: 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) {
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) {
       execlp("/bin/ls","ls",NULL);
    }
    else {
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Error condition

child process

parent process

Process termination: Special cases

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

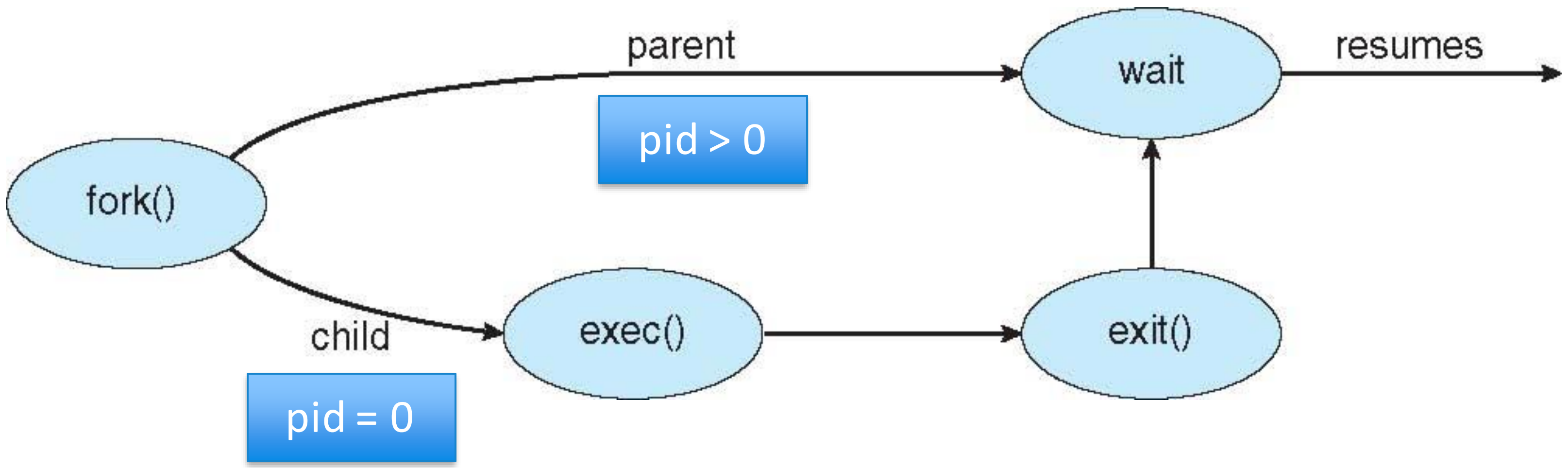
- Zombie process
  - A process that has terminated, but whose parent has not (yet) called wait()
  - All processes move to this state when they terminate, and remain there until parent calls wait()
  - Entry in process table removed only after calling wait()

Zombie and orphan process

- Zombie process
  - A process that has terminated, but whose parent has 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()

Usual workflow between parent & child



- Workflow can be different from what is shown
  - E.g., parent may or may not wait for the child
  - E.g., child may or may not use exec()

Inter Process Communication

Inter-process communication (IPC)

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

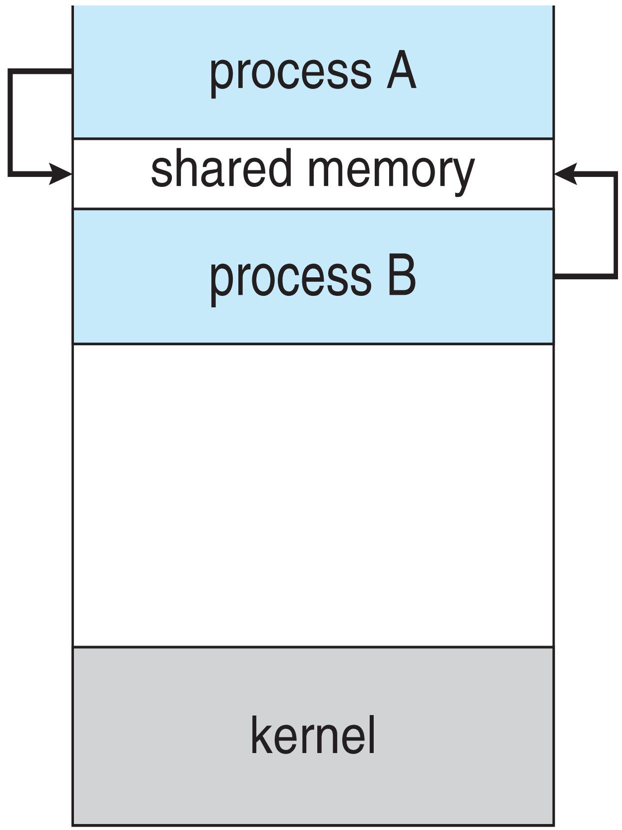
- Cooperating processes need IPC
  - Several types of IPC support provided by OS

Inter-process communication (IPC)

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

Shared memory system

Schematic for shared memory



Require the communicating processes to establish a region of shared memory

Relevant system calls in Linux / Unix
shmget( )
shmat( )
shmdt( )
shmctl( )

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

Create shared memory segment

```
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);

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

Create shared memory segment

Attach the memory segment to the address space of this process

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);
```

Create shared memory segment

Attach the memory segment to the address space of this process

Detach the memory segment from the address space of this process

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);
```

Create shared memory segment

Attach the memory segment to the address space of this process

Detach the memory segment from the address space of this process

Mark the segment to be destroyed

Example: 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 written to / read from a buffer

Example: Producer consumer problem

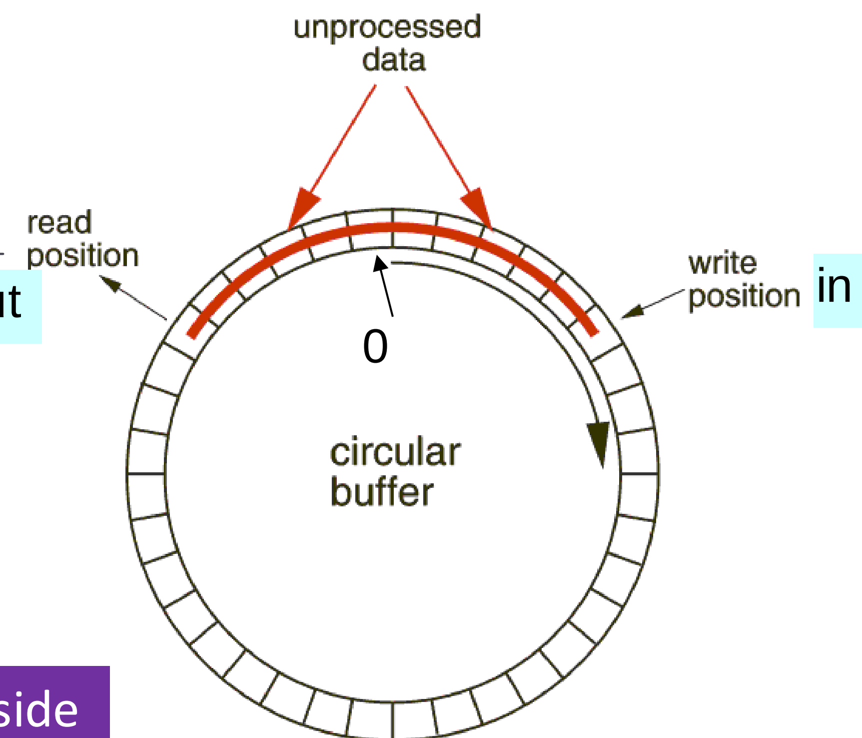
- A producer process produces information that is consumed by the consumer process
  - Compiler produces assembly code consumed by assembler
  - Program produces lines to print, print spool consumes
  - The information is written to / read from a buffer

- Two variants
  - Bounded buffer: producer waits when buffer is full, consumer waits when buffer is empty
  - Unbounded buffer: 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 {
  . . . // info to be shared
} item;

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



These variables reside in shared memory

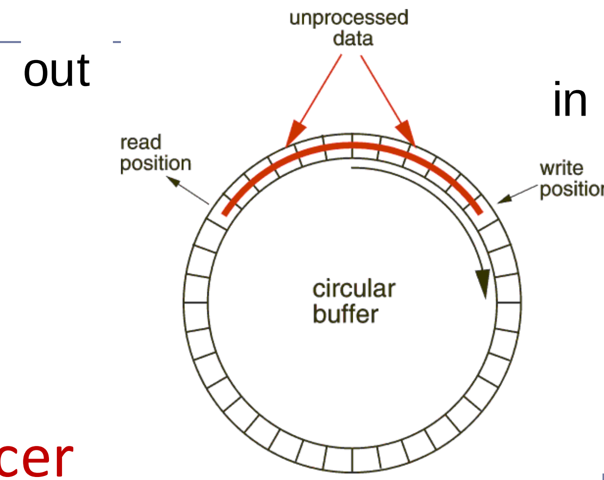
Key ideas

- Circular buffer
  - Index in: the next free position to write to
  - Index out: the next filled position to read from
- To check buffer full or empty:
  - Buffer empty: in==out
  - Buffer full: in+1 % BUFFER_SIZE == out
    - This scheme allows at most BUFFER_SIZE – 1 items in the buffer

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;

}
```

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 (shared memory)
- Suppose that we want to use all buffer space:
  - an integer count: the number of filled buffers (shared memory)
  - 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

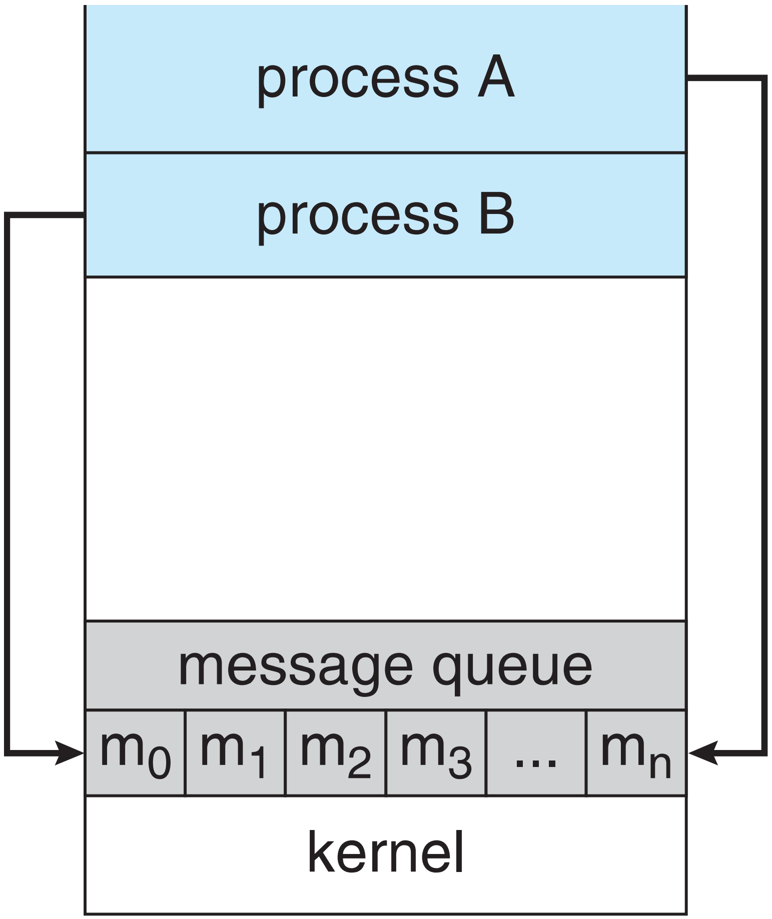
- Mechanism for processes to communicate and to synchronize their actions

- 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

Pipes

- Acts as a medium to allow two processes to communicate
  - Communication can be uni/bi-directional
  - Must there exist a relationship (i.e., parent-child) between the communicating processes?
  - Can the pipes be used over a network?

Ordinary pipes

- A message passing medium between related processes
  - Cannot be accessed from outside the process
  - Typically, a parent process creates a pipe and uses it to communicate with its child process
  - Pipes behave like FIFO queues
  - Read-write in pipe == producer-consumer
  - Producer writes to one end (the write-end of pipe)
  - Consumer reads from the other end (the read-end of pipe)
  - Unidirectional

Producer consumer in pipes

Producer

```
message next_produced;

while(TRUE){
   <produce and put data in next produced>
   ...
   send( next_produced);
}
```

Consumer

```
message next_consumed;

while(TRUE){
  receive(next_consumed);
  ...
  <data in next consumed>
}
```

Named pipes

- Accessed as files by processes
  - No parent-child relation is necessary
  - Still behave like FIFO queues (even called fifo)
  - Several processes can use the named pipe

Named pipes

- Accessed as files by processes
  - No parent-child relation is necessary
  - Still behave like FIFO queues (even called fifo)
  - Several processes can use the named pipe

```
char* myfifo = '/tmp/myfifo';
mkfifo (myfifo, 0666); // creates the fifo or named pipe (a special file)
…
fd = open(myfifo, O_WRONLY);  // Process A
write(fd, …);    // Process A
close(fd);    // Process A
…
fd = open(myfifo, O_RDONLY);  // Process B
read(fd, …);    // Process B
close(fd);    // Process B
```

Named pipes

- Once you have created a FIFO special file / named pipe, any process can open it for reading or writing, in the same way as an ordinary file.
- However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

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
- Queues for process scheduling
  - Ready queues, event queues, queueing diagram
- How does two processes talk?
  - Shared memory, pipe, named pipe