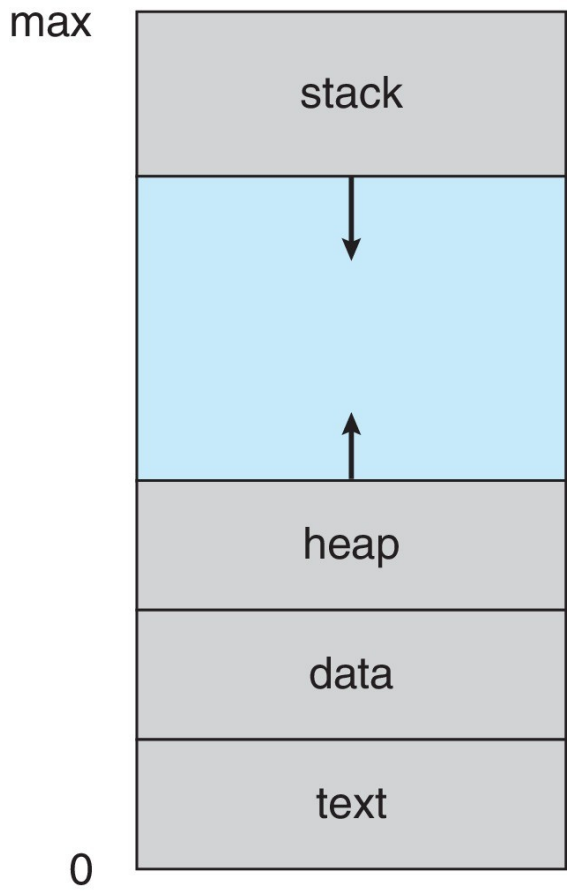
Processes

Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution
- Process execution must progress in **sequential** fashion. No parallel execution of instructions of a single process
- 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

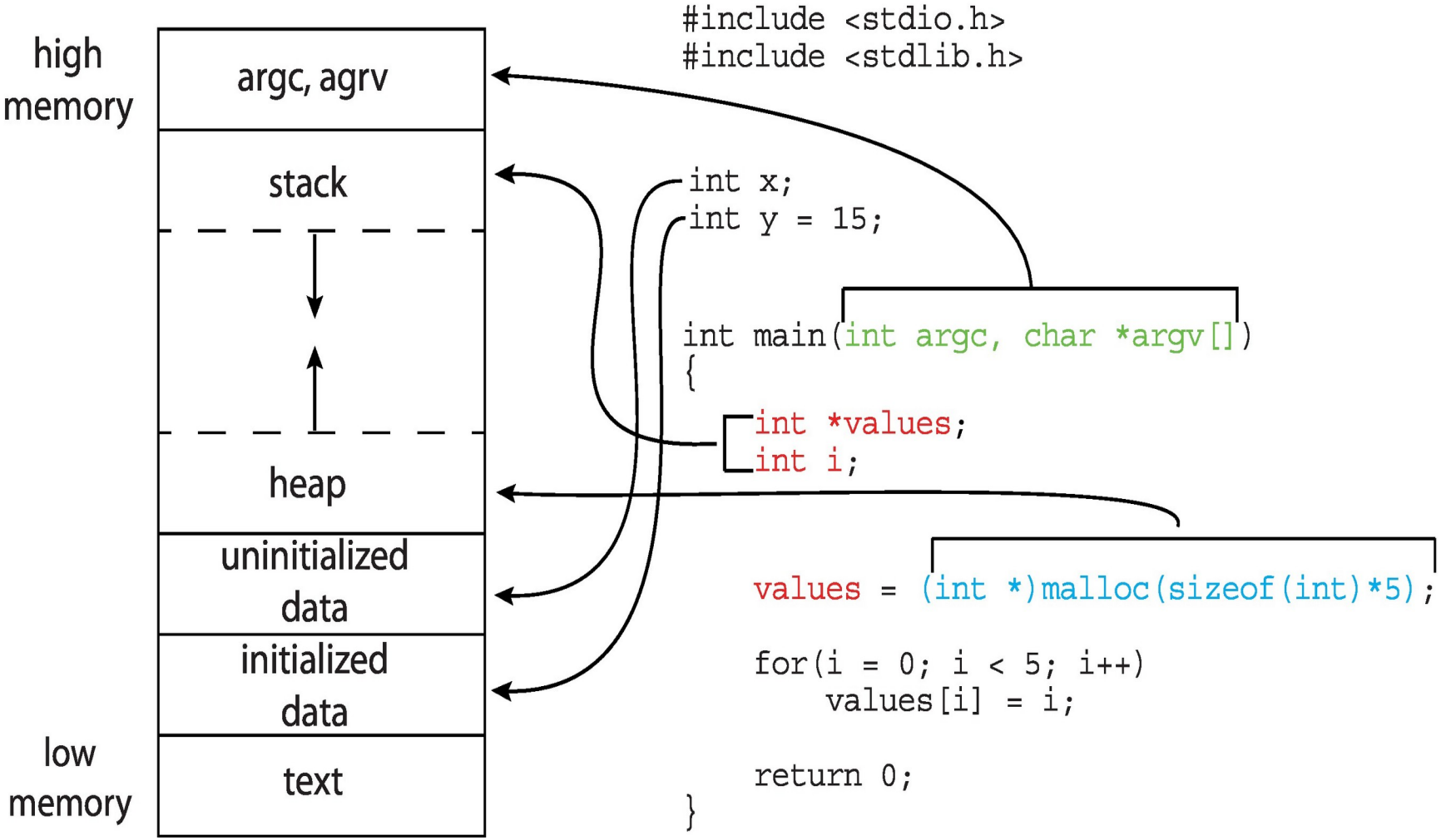
Process in Memory



Process Concept

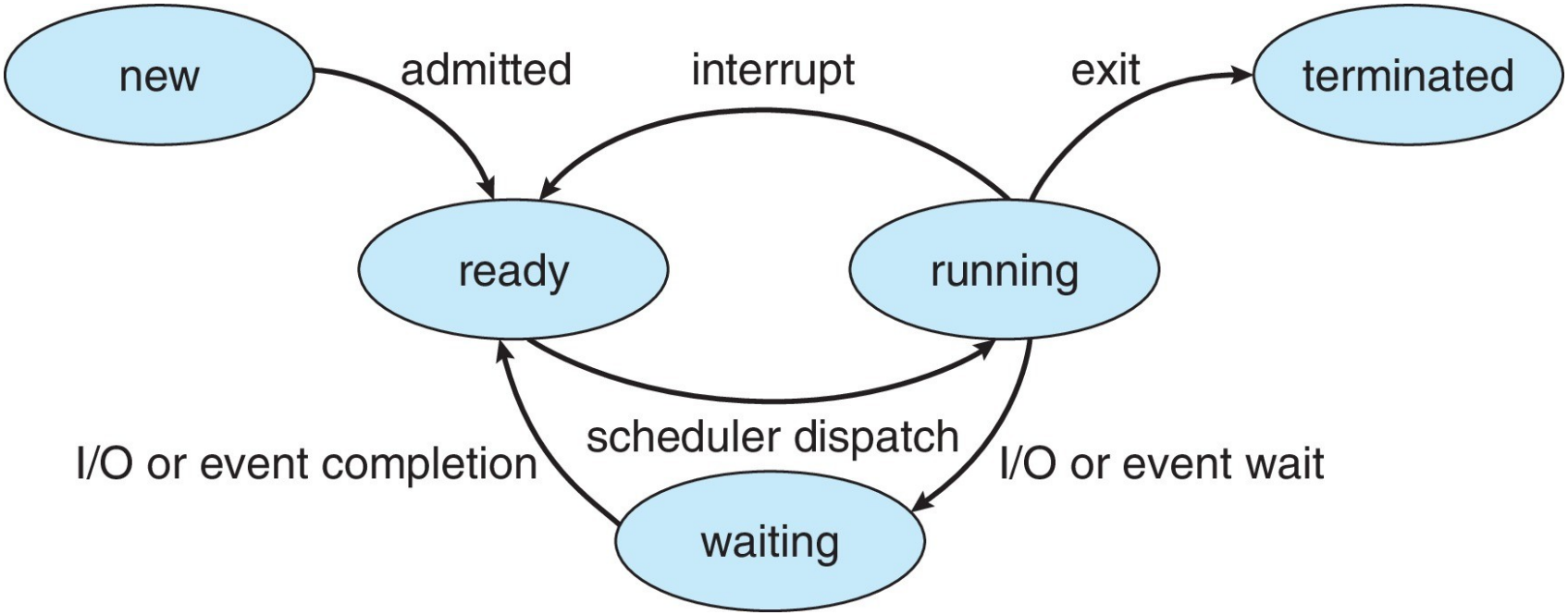
- Program is **passive** entity stored on disk (**executable file**);
- Process is **active**
  - ✓Program becomes process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - ✓Consider multiple users executing the same program

Memory Layout of a C Program



Process State

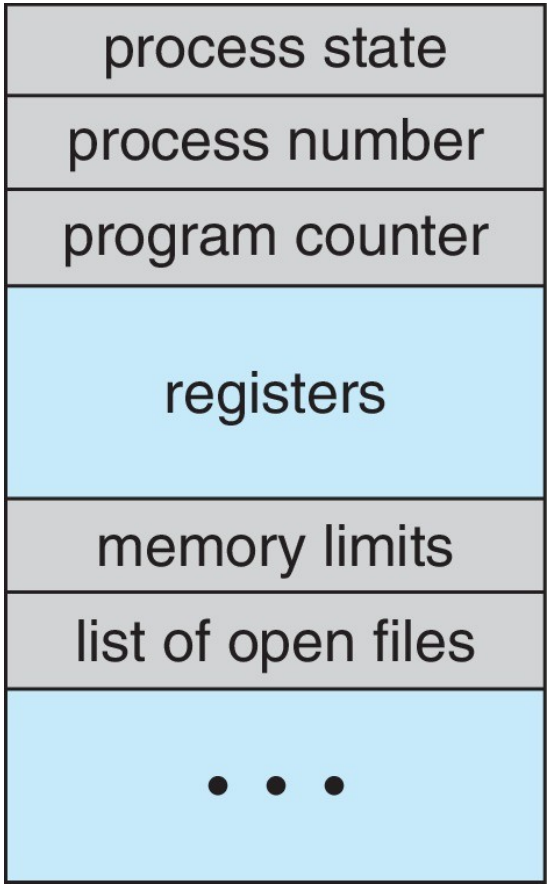
- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution



Process Control Block (PCB)

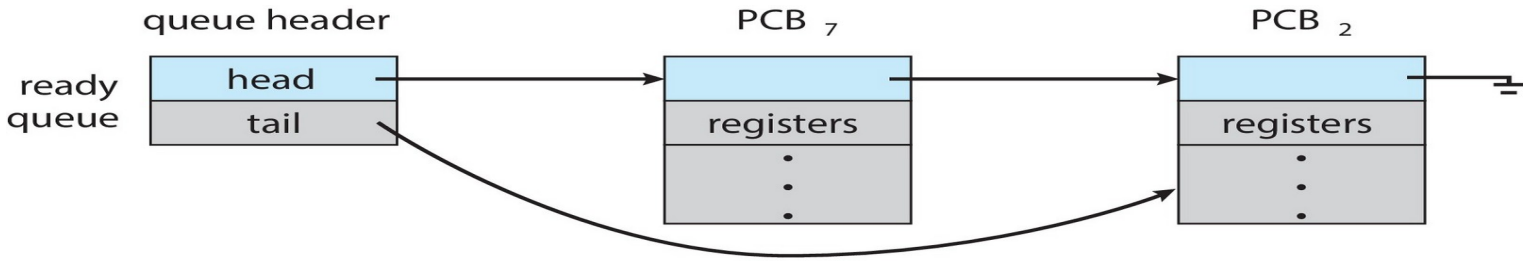
PCB contains Information associated with each process

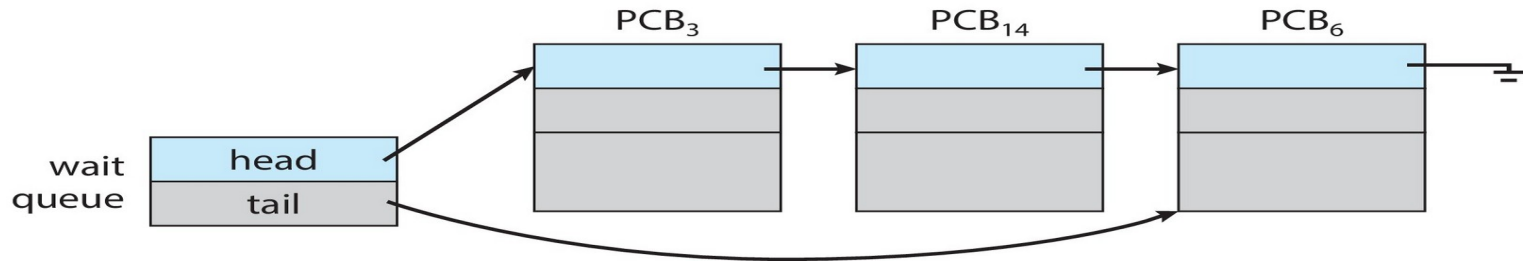
- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- process priorities, scheduling queue pointers
- Memory management information – memory allocated to the process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



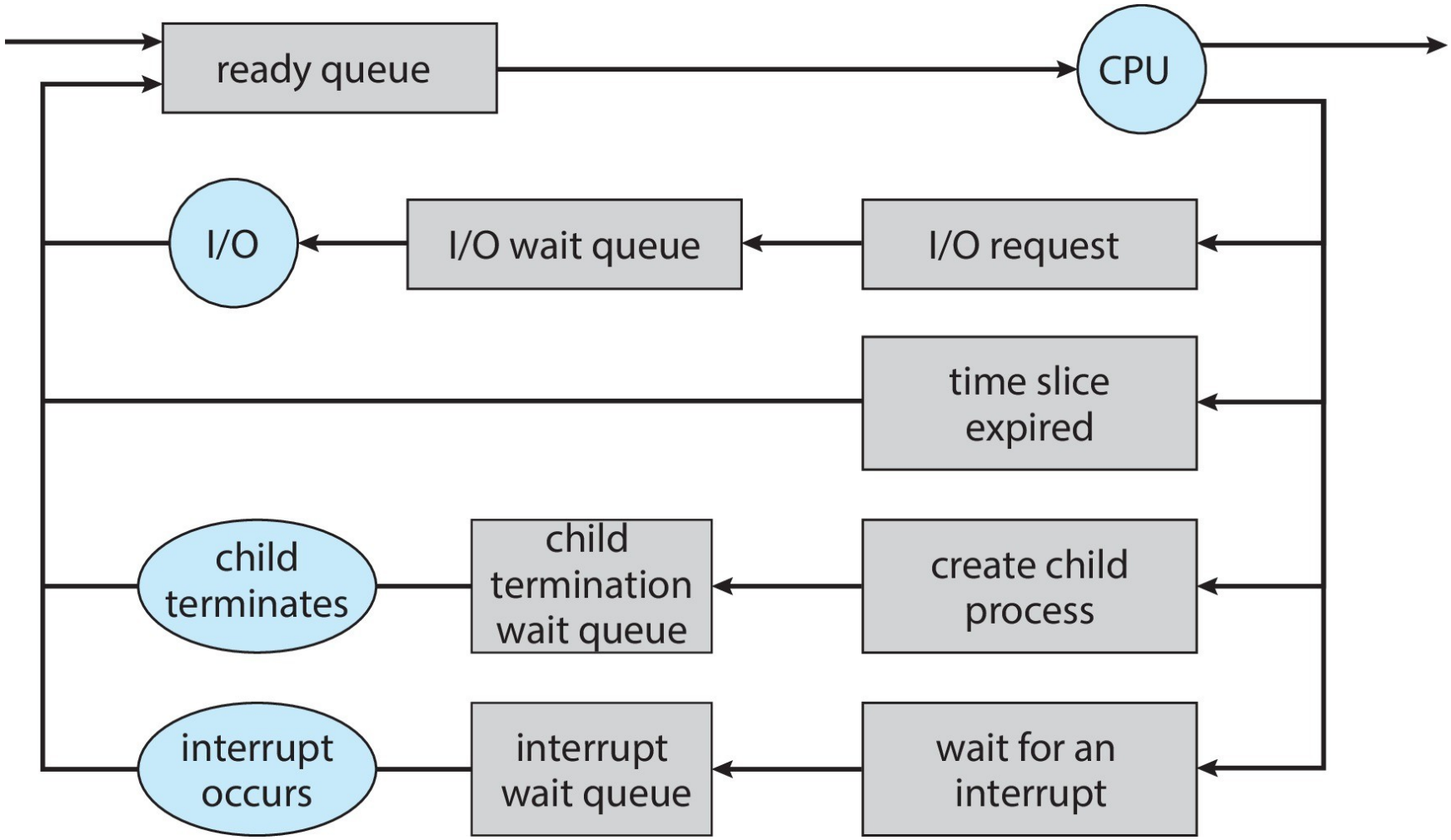
Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU core.
- Goal -- Maximize CPU use, quickly switch processes onto CPU core.
- Maintains **scheduling queues** of processes.
  - ✓ **Ready queue** – set of all processes residing in main memory, ready and waiting to execute.
  - ✓ **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - ✓ Processes migrate among the various queues.





Representation of Process Scheduling

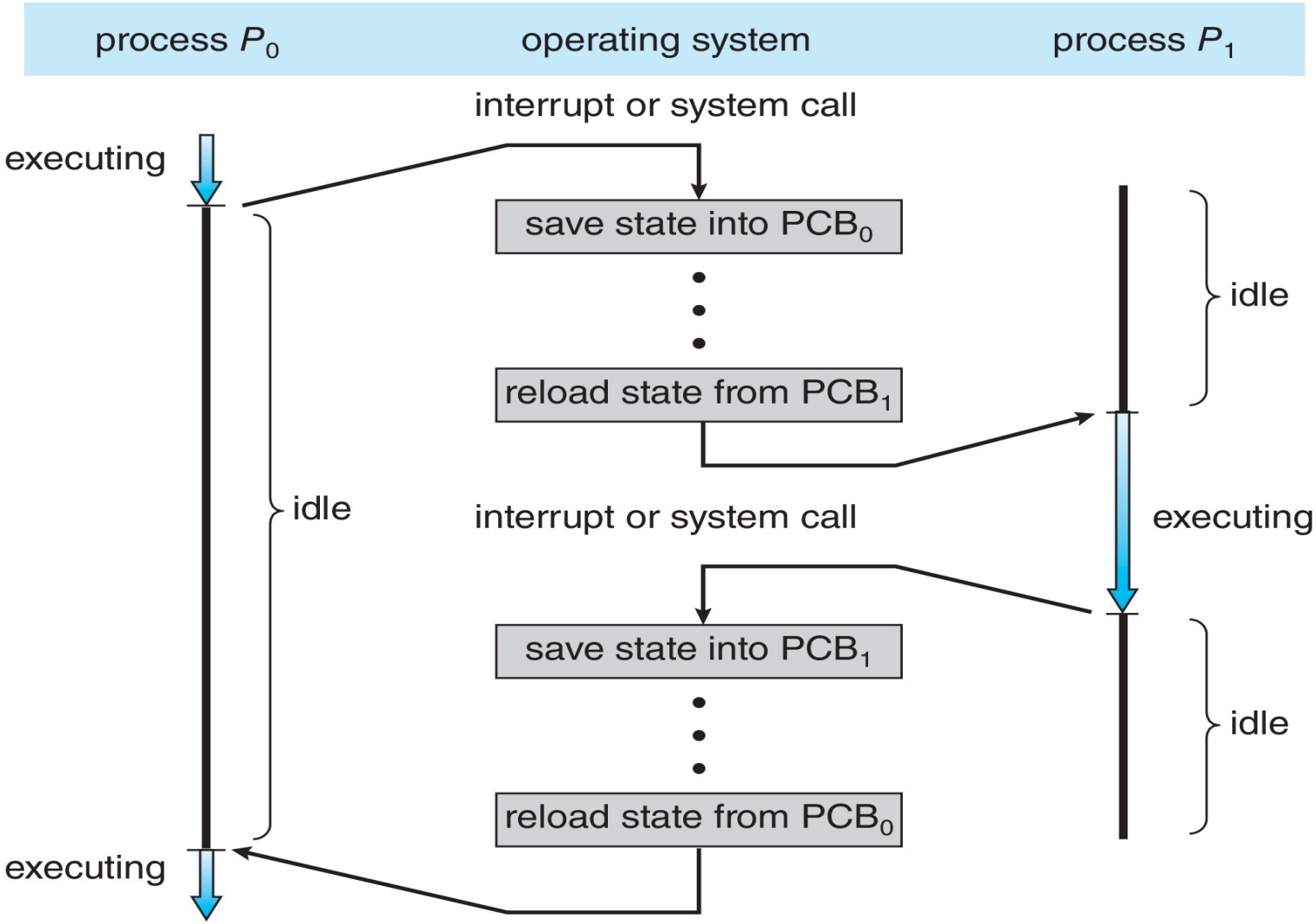


Context Switch

- 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 pure 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.

CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.



Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - ✓Single **foreground** process- controlled via user interface
  - ✓Multiple **background** processes– in memory, running, but not on the display, and with limits
  - ✓Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - ✓Background process uses a **service** to perform tasks
  - ✓Service can keep running even if background process is suspended
  - ✓Service has no user interface, small memory use

Operations on Processes

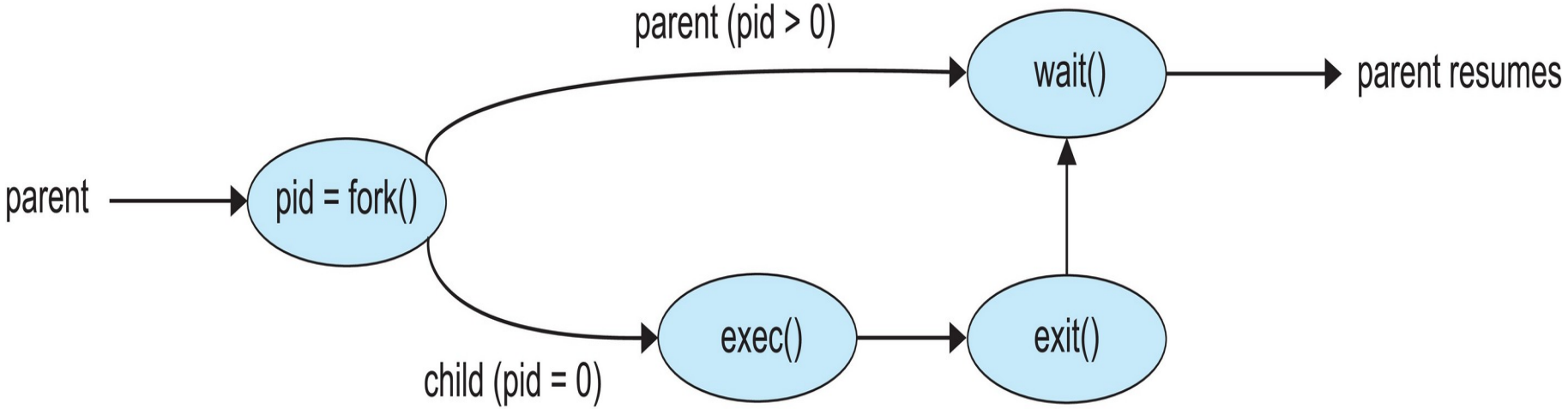
- System must provide mechanisms for:
  - Process creation
  - Process termination

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
  - ✓A child process may be able to obtain its resources directly from OS
  - ✓Children share subset of parent's resources
- Execution options
  - ✓Parent and children execute concurrently
  - ✓Parent waits until children terminate

Process Creation

- 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
  - ✓Parent process calls wait()waiting for the child to terminate



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;
}
```

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 (either normally or abnormally), then all its children must also be terminated.
  - ✓**cascading termination.** All children, grandchildren, etc., are terminated.
  - ✓The termination is initiated by the operating system.
- We can terminate a process by using the exit() system call, providing an exit status as a parameter:

  exit(1);
- In fact, under normal termination, exit() will be called either directly (exit(1)) or
- indirectly, as the C run-time library will include a call to exit() by default.

Process Termination

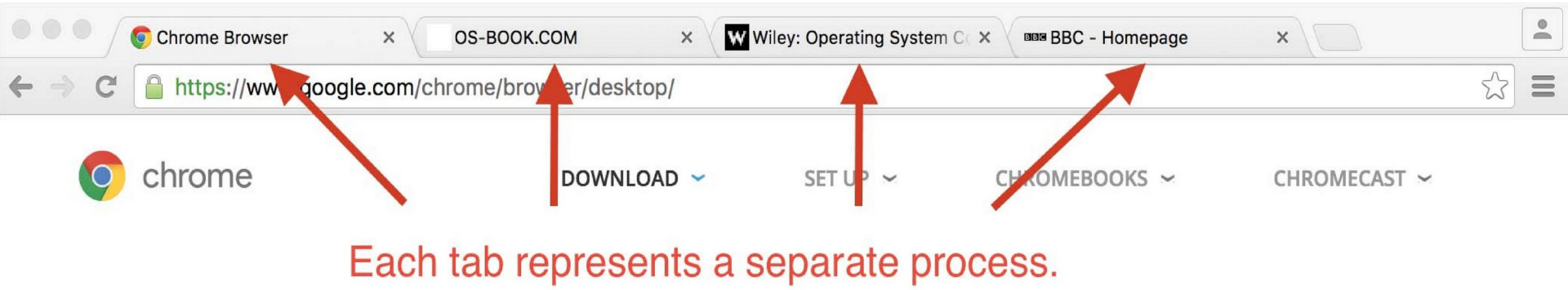
- The parent process may wait for termination of a child process by using the **wait()**system call**.**
- This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated.

  pid = wait(&status);

- When a process terminates, its resources are deallocated by the operating system.
- However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status.
- A process that has terminated, but whose parent has not yet called wait(), is known as a zombie process.
- if a parent did not invoke wait() and instead terminated, leaving its child processes as orphans.

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - ✓ If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - ✓ **Browser** process manages user interface, disk and network I/O
  - ✓ **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
  - ✓ **Plug-in** process for each type of plug-in



Interprocess Communication

- Processes executing concurrently in the operating system may be either **independent** or **cooperating.**

➢ **Independent** process cannot affect or be affected by the other processes executing in the system.

- A process is independent if it does not share data with any other processes executing in the system.

➢ **Cooperating** process can affect or be affected by other processes executing in the system.

- Any process that shares data with other processes is a cooperating process.

Interprocess Communication

There are several reasons for providing an environment that allows process cooperation:

- **Information sharing:** Since several applications may be interested in the **same piece of information**, we must provide an environment to allow concurrent access to such information.
- **Computation speedup:** If we want a particular task to run faster, we must break it into **subtasks**, each of which will be executing in parallel with the others.
- **Modularity:** To construct the system in a modular fashion, dividing the system functions into separate processes or threads.

Interprocess Communication

- Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data— that is, send data to and receive data from each other.
- Two fundamental models of interprocess communication:
  - ✓Shared memory
  - ✓Message passing

Communications Models

- **Shared-memory model**: A region of memory that is shared by the cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.

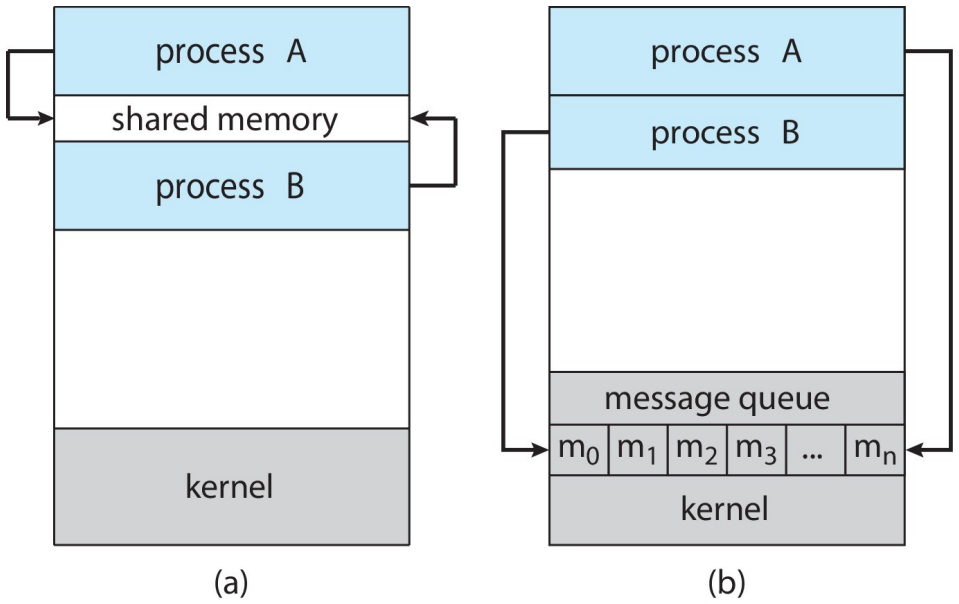


Fig: (a) Shared memory. (b) Message passing

- **Message-passing model**: communication takes place by means of messages exchanged between the cooperating processes.

Communications Models

- Message passing is useful for exchanging smaller amounts of data.
- Message passing is also easier to implement in a distributed system than shared memory.

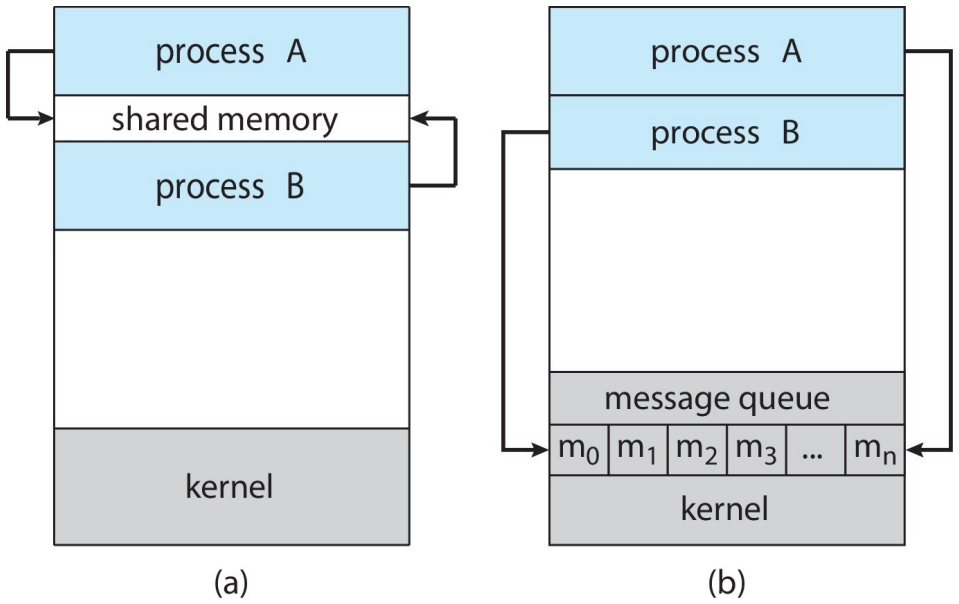


Fig: (a) Shared memory. (b) Message passing

Communications Models

- Shared memory can be faster than message passing, since message-passing systems are implemented using system calls, and thus require the more time-consuming task of kernel intervention.

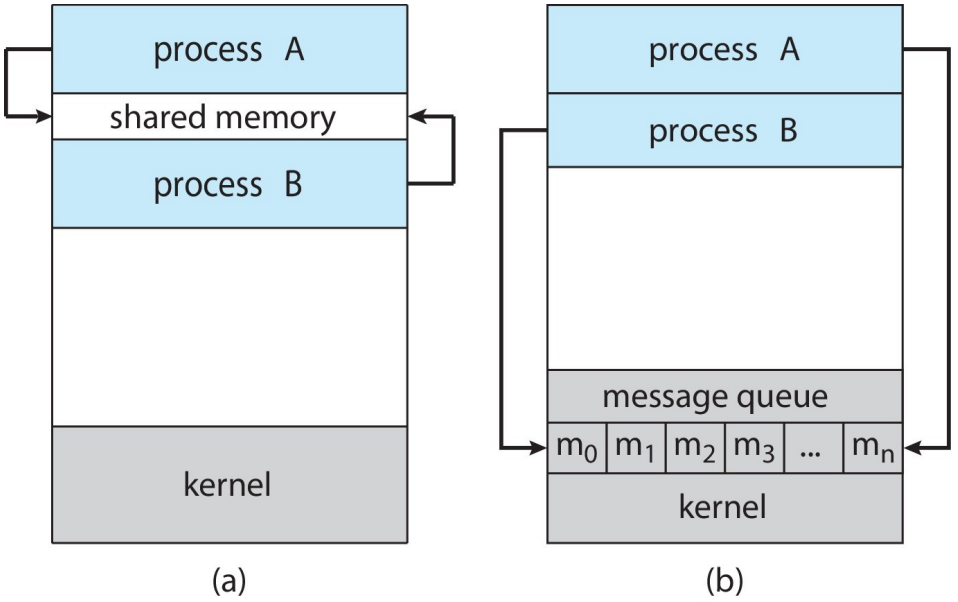


Fig: (a) Shared memory. (b) Message passing

- In shared-memory systems, system calls are required only to establish shared memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Communications Models

- Message passing: For example, an Internet **chat** program could be designed so that chat participants communicate with one another by exchanging messages.

- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- The *message* size is either fixed or variable

Producer–consumer problem

- To illustrate the concept of cooperating processes, let's consider the producer–consumer problem, which is a common paradigm for cooperating processes.
- Exchange information by reading and writing data in the shared areas.
- To allow producer and consumer processes to run concurrently, we must have available a **buffer of items** that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.

Producer–consumer problem

- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

➢ Two types of buffers can be used:

- **Unbounded-buffer** places no practical limit on the size of the buffer:
  - ✓ Producer never waits
  - ✓ Consumer waits if there is no buffer to consume
- **Bounded-buffer** assumes, there is a fixed buffer size
  - ✓ Producer must wait if all buffers are full
  - ✓ Consumer waits if there is no buffer to consume

Bounded-Buffer – Shared-Memory Solution

- Shared data

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

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

- The shared buffer is implemented as a **circular array** with two logical pointers: in and out.

Bounded-Buffer – Shared-Memory Solution

```
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;
}
```

- The variable **in** points to the next free position in the buffer; **out** points to the first full position in the buffer.

- The buffer is empty when **in** == **out**;
- the buffer is full when ((**in** + 1) % BUFFER SIZE) == **out**.

```
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 */

}
```

Bounded-Buffer – Shared-Memory Solution

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

Bounded-Buffer – Shared-Memory Solution

- our original solution allowed at most BUFFER SIZE − 1 items in the buffer at the same time.
- One possibility to overcome this deficiency, is to add an integer variable, **count**, initialized to 0.
- **count** is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.

Bounded-Buffer – Shared-Memory Solution

```
while (true) {
        /* produce an item in next produced */

        while (count == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
}
```

```
while (true) {
        while (count == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        /* consume the item in next consumed */
}
```

Bounded-Buffer – Shared-Memory Solution

- Although the producer and consumer routines shown are correct separately, they may not function correctly when executed concurrently.
- Suppose that the value of the variable **count** is currently **5** and that the producer and consumer processes **concurrently** execute the statements **count++** and **count—**.
- Following the execution of these two statements, the value of the variable count may be 4, 5, or 6! The only correct result, though, is count == 5

Bounded-Buffer – Shared-Memory Solution

- **count++** could be implemented as (**Producer**)

  register$_1$ = count

  register$_1$ = register$_1$ + 1

  count = register$_1$

- **count--** could be implemented as (**Consumer**)

  register$_2$ = count

  register$_2$ = register$_2$- 1

  count = register$_2$

- Consider this execution interleaving with "count = 5" initially:

  S_0: producer execute **register$_1$ = count** {register$_1$ = 5}

  S_1: producer execute **register$_1$ = register$_1$ + 1** {register$_1$ = 6}

  S_2: consumer execute **register$_2$ = count** {register$_2$ = 5}

  S_3: consumer execute **register$_2$ = register$_2$ – 1** {register$_2$ = 4}

Race Condition

- When two or more process cooperates with each other, manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.
- Such processes need to be **synchronized** so that their **order of execution** can be guaranteed.
- The procedure involved in preserving the appropriate order of execution of cooperative processes is known as Process Synchronization.
- There are various synchronization mechanisms that are used to synchronize the processes.