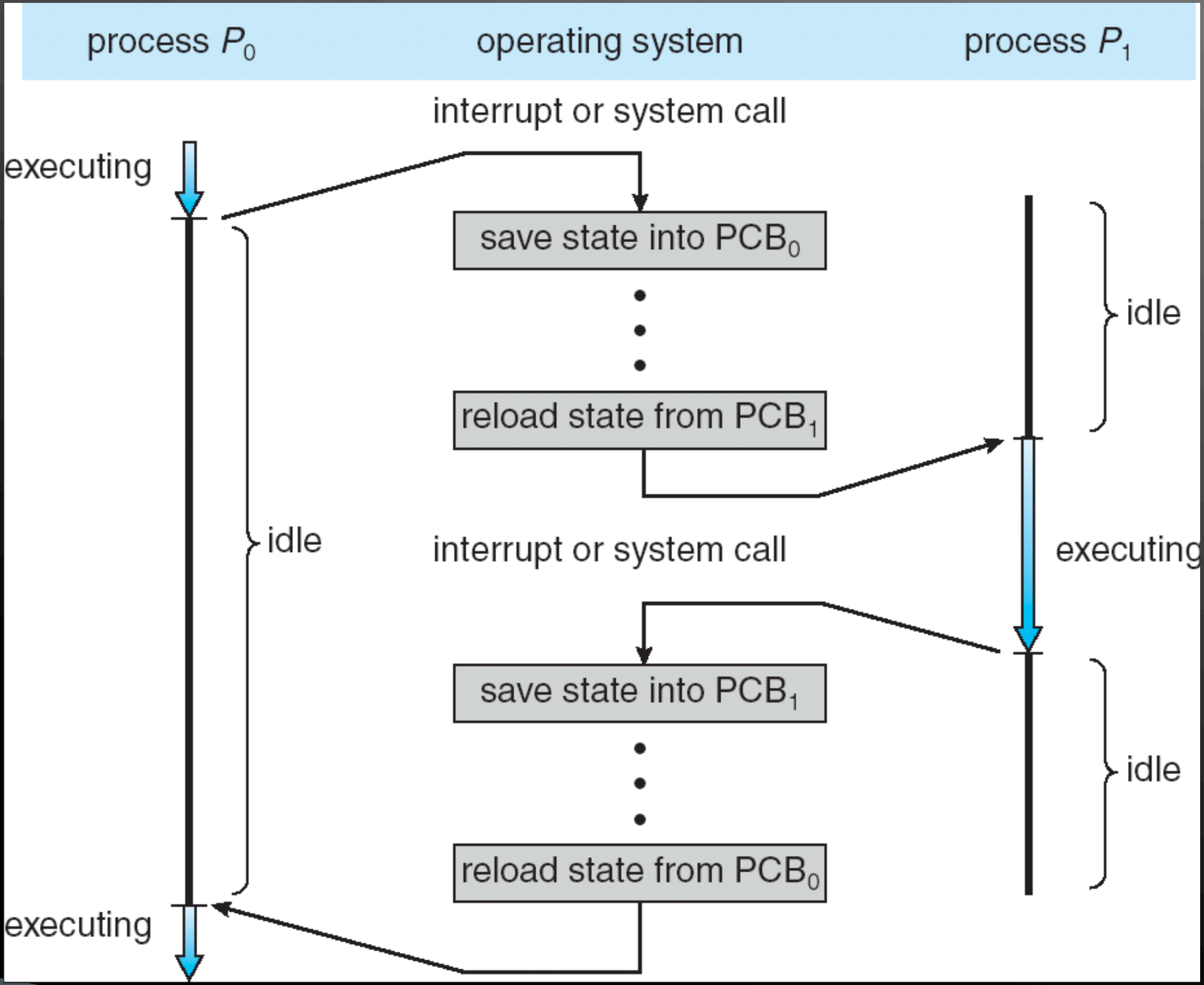
Chapter 3: Processes

Context Switch

- When an interrupt occurs or time slot of a process expires, the system needs to save the current context of the process (PCB) currently running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.
- We perform a state save of the process to be suspended (kernel or user mode) and then a state restore for the process to be resumed.
- **Context-switch time is overhead; the system does no useful work while switching**
- Context switch time varies from machine to machine, depending on the memory speed, the number of registers that must be copied etc Typical speeds are a few milliseconds.

CPU Switch From Process to Process



Operations on processes

- The processes in most systems can execute concurrently, and they may be created and deleted dynamically. These systems must provide a mechanism for process creation and termination.

Process Creation

- A process may create several new processes during the course of execution. The creating process is called a parent process and the new processes are called the children of that process.
- Each child process can further create other processes, forming a tree of processes.
- Most operating systems identify processes according to a unique process identifier (or pid), which is typically an integer number.

Fork() System Call

- The fork() function is used to create a new process by duplicating the existing process from which it is called.

- The existing process from which this function is called becomes the parent process and the newly created process becomes the child process.

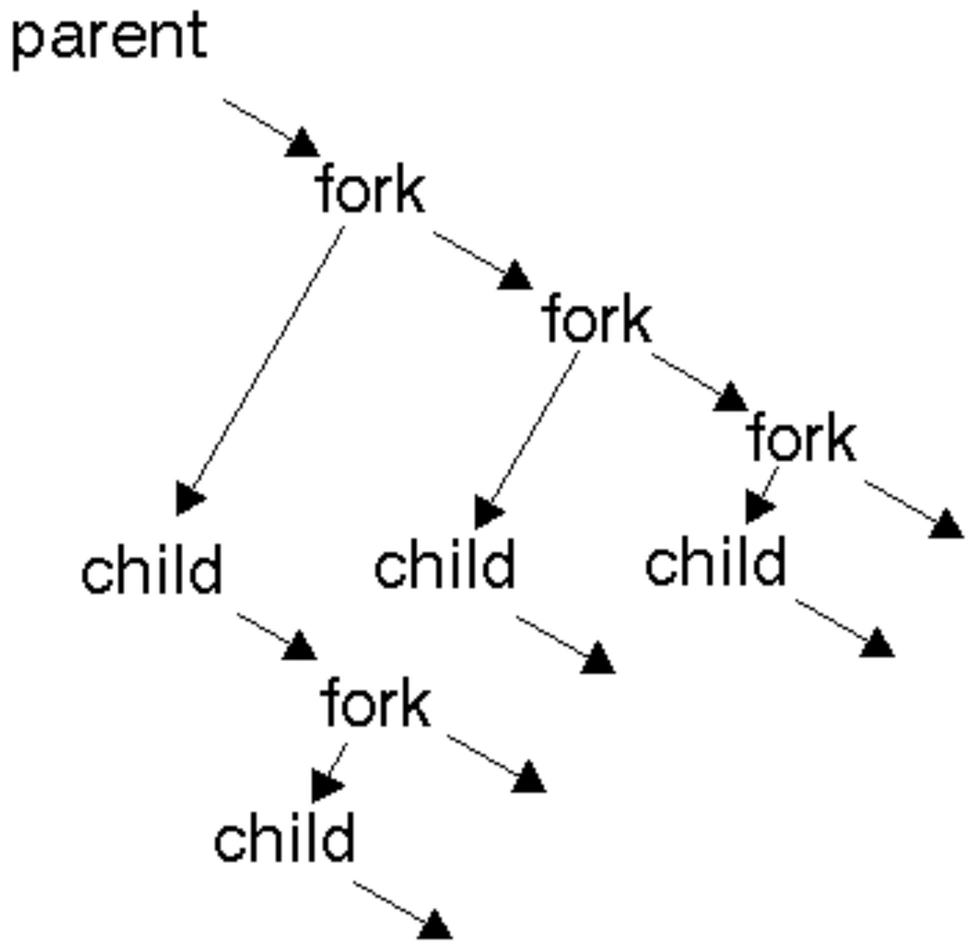
- As already stated that child is a duplicate copy of the parent but there are some exceptions to it.

  - The child has a unique PID like any other process running in the operating system.
  - The child has a parent process ID which is same as the PID of the process that created it.
  - Resource utilization and CPU time counters are reset to zero in child process.

- Note that the above list is not exhaustive.

Operations on processes (cntd)

Process Creation



Process Creation (cntd)

- In general, a process will need certain resources (**CPU time, memory, files,I/O devices**) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the OS or it may be constrained to a subset of the resources of the parent process.
- The parent may have to partition its resources among its children or it may be able to share some resources (such as memory or files) among several of its children.
- Restricting a child process to a subset of the parent’s resources prevents any process from overloading the system by creating too many subprocesses.

Process Creation (cntd)

- When a process creates a new process, two possibilities exist in terms of execution:

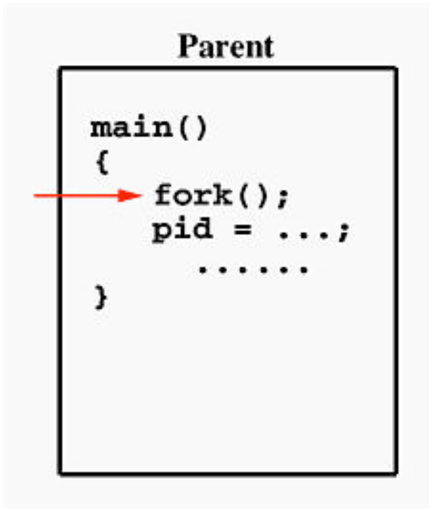
1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

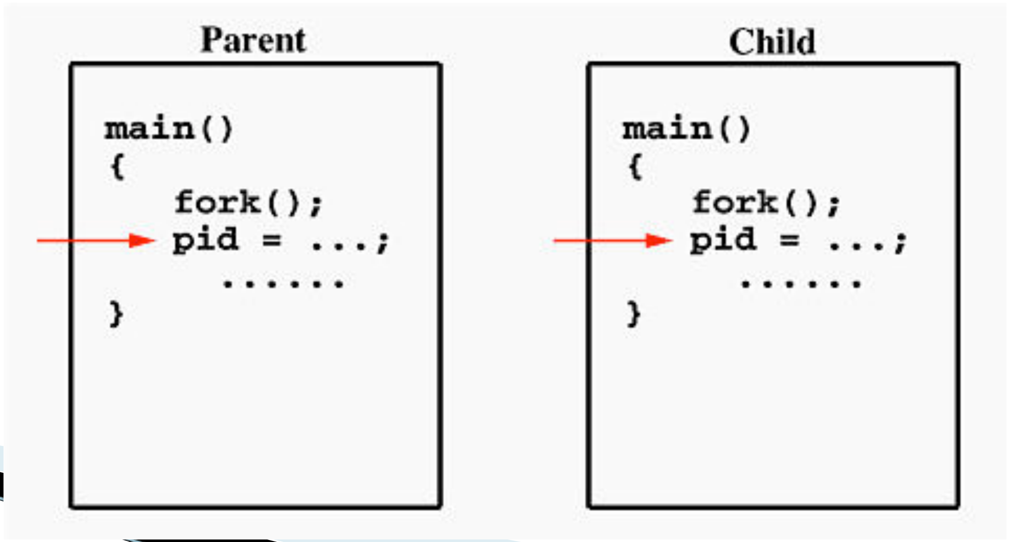
- 1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
- 2. The child process has a new program loaded into it.

Program using Fork()

Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: The return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.



```
main()
{
   fork();
   pid = ...;
      ......
}
```



```
main()
{
   fork();
   pid = ...;
      ......
}
```

```
main()
{
   fork();
   pid = ...;
      ......
}
```

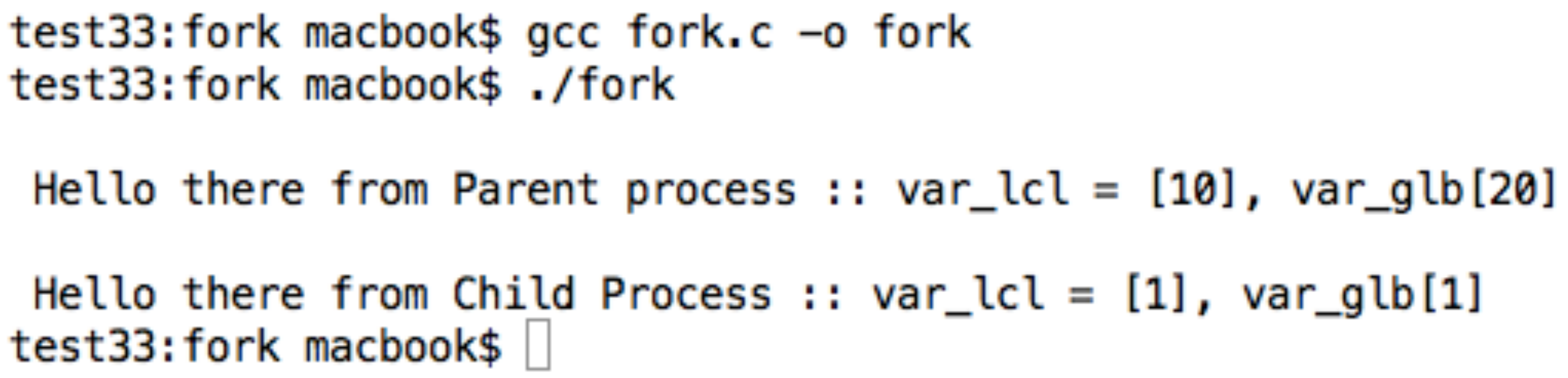
Program using Fork()

```
#include <stdio.h>

int var_glb=0; /* A global variable*/

int main(void)
{
    pid_t forkid;
    int var_lcl = 0;

    forkid = fork();

    if(forkid >= 0) // fork was successful
    {
        if(forkid == 0) // This is the child process
        {
            var_lcl++;
            var_glb++;
            printf("\n Hello there from Child Process :: var_lcl = [%d], var_glb[%d]\n",
var_lcl, var_glb);
        }
```

Program using Fork()

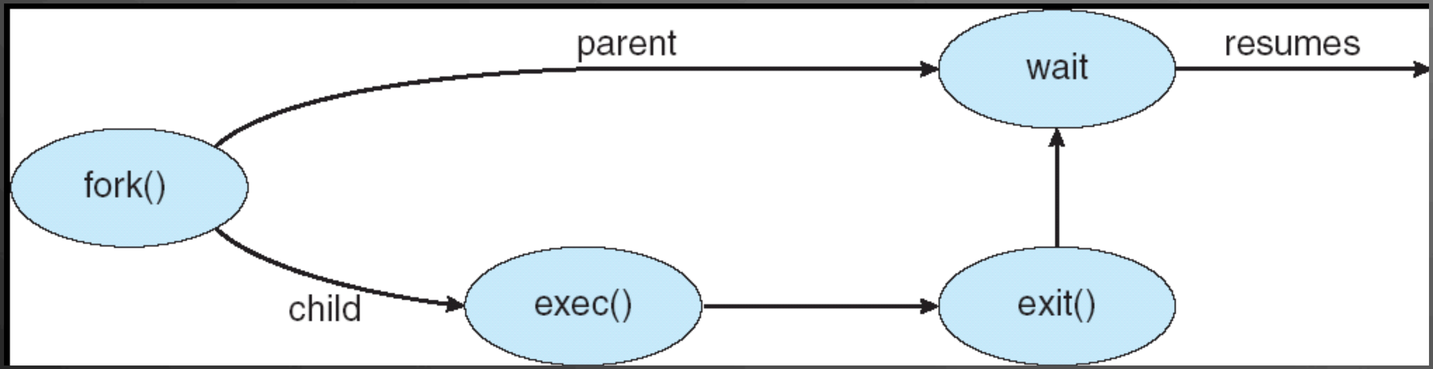
```
 else //Parent process
        {
            var_lcl = 10;
            var_glb = 20;
            printf("\n Hello there from Parent process :: var_lcl = [%d],
var_glb[%d]\n", var_lcl, var_glb);
        }
    }
    else // fork failed
    {
        printf("\n Fork failed, quitting!!!!!!\n");
        return 1;
    }

    return 0;
 }
```

Program using Fork()



```
test33:fork macbook$ gcc fork.c -o fork
test33:fork macbook$ ./fork

 Hello there from Parent process :: var_lcl = [10], var_glb[20]

 Hello there from Child Process :: var_lcl = [1], var_glb[1]
test33:fork macbook$ 
```

Process Creation



-> Typically, the exec() system call is used after a forkO to replace the process's memory space with a new program.
-> It loads a binary file into memory (destroying the memory image of the program containing the execO system call) and starts its execution.
->The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait () system call to move itself off the ready queue until the termination of the child.

Process Termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit () system call.

- All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

- A process can cause the termination of another process via an appropriate system call. Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs.

Process Termination (cntd)

- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
  - The task assigned to the child is no longer required.
  - The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

- Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

Cooperating Processes

- Processes executing concurrently in the operating system may be either independent processes or 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

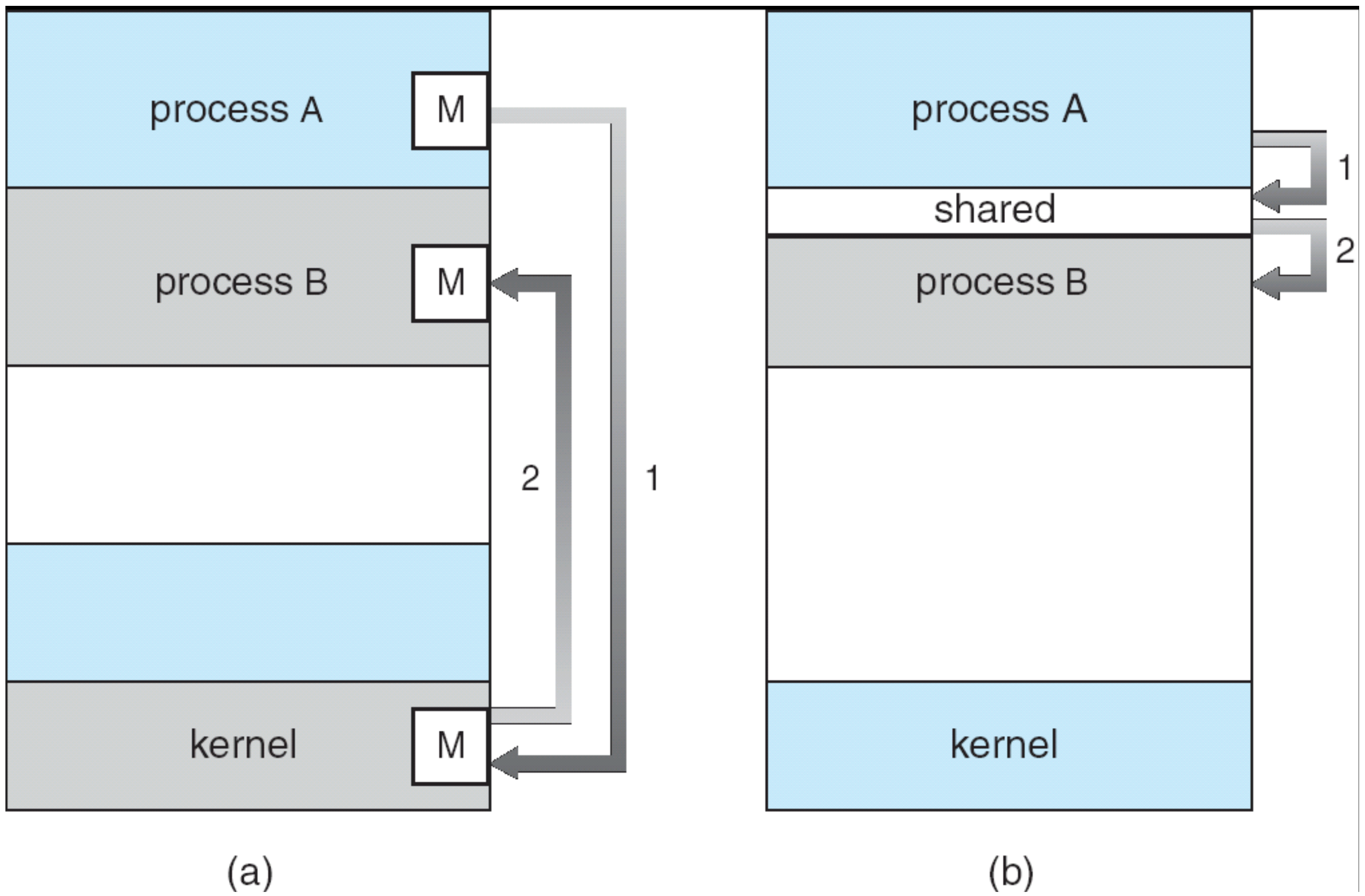
Models of IPC

- There are two fundamental models of interprocess communication:
- **shared memory**
- **message passing.**

- In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.

- In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.

Models of IPC (cntd)



Models of IPC (cntd)

Both of the models just discussed are common in operating systems.

- Message passing is useful for exchanging smaller amounts of data.

- Shared memory allows maximum speed and convenience of

communication, as it can be done at memory speeds when within a computer and requires no kernel intervention.

Shared Memory Systems

Producer-consumer problem

- To illustrate the concept of cooperating processes, let's consider the Producer-consumer problem, which is a <u>common paradigm for cooperating processes</u>.

- A producer process produces information that is consumed by a consumer process. For example 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. 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.

Shared Memory Systems

Producer-consumer problem (cntd)

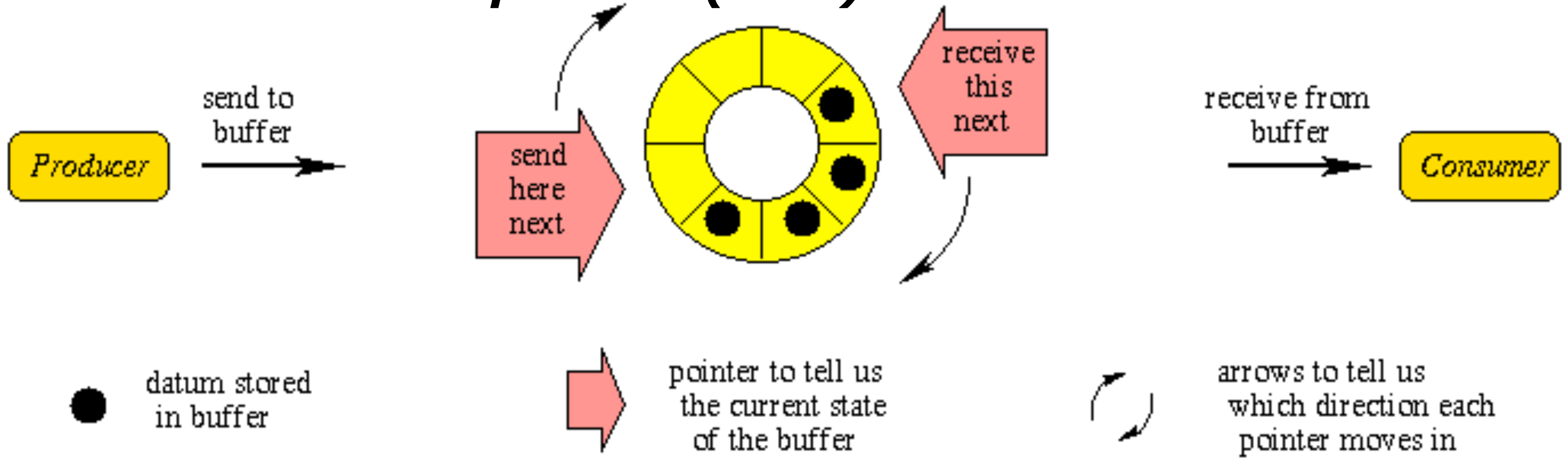
Now we look at how the bounded buffer can be used to enable processes to share memory. The following variables reside in a region of memory shared by the producer and consumer processes:

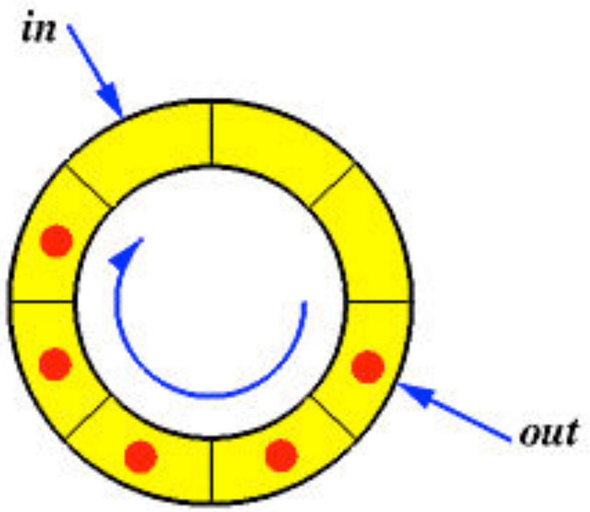
```
#define BUFFER_SIZE 10
int buffer [BUFFER_SIZE] ;
int in = 0 ,
int out = 0 ;
```

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

Shared Memory Systems

Producer-consumer problem(cntd)


Producer
send to buffer
send here next
receive this next
receive from buffer
Consumer
datum stored in buffer
pointer to tell us the current state of the buffer
arrows to tell us which direction each pointer moves in



Shared Memory Systems

Producer-consumer problem (cntd)

Producer

```
    while (true) {
      /* Produce an item */
           while (((in + 1) % BUFFER SIZE)  == out)
            ;   //no free buffers
           buffer[in] = item;
           in = (in + 1) % BUFFER SIZE;
        }
```

Consumer

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

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

Producer-consumer problem (cntd)

- How many items can be stored in a buffer of size in previous example?

Producer-consumer problem (cntd)

```
#define BUFFER_SIZE 10
int buffer [BUFFER_SIZE] ;
int in = 0 ,
int out = 0 ;
int count = 0;
```

Producer-consumer problem (cntd)

- How many items can be stored in a buffer of size in previous example?
    - Our solution allowed at most BUFFER.SIZE - 1 items in the buffer at the same time (refer to old slides).
    - To remedy this deficiency we can 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.

Producer-consumer problem (cntd)

```
while (true) {
        /* Produce an item */
        while (count == BUFFER SIZE)
         ;   //no free buffers
        buffer[in] = item;
        in = (in + 1) % BUFFER SIZE;
        count = count+1;
}
```

Producer

```
while (true) {
        while (count == 0)
                ; //nothing to consume

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

Consumer

Message Passing Systems

- Message passing provides a mechanism to allow processes to communicate without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- For example, a chat program used on the World Wide Web could be designed so that chat participants communicate with one another by exchanging messages.

- A message-passing facility provides at least two operations: send(message) and receive(message).If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them.

Message Passing Systems

- We look at issues related to implementation of message passing systems:

1.Naming: In message passing communication can be either direct or indirect.

- In *direct communication*, each process must explicitly name the recipient or sender of the communication.

- send(P, message)—Send a message to process P.
- receive (Q, message)—Receive a message from process Q.

- The processes need to know only each other’s identity to communicate.

Message Passing Systems

- A link is associated with exactly two processes.

- This scheme exhibits **symmetry** in addressing; that is, both the sender process and the receiver process must name the other to communicate.

- A variant of this scheme employs **asymmetry** in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender.
- send(P, message)—Send a message to process P.
- receive(id, message)—-Receive a message from any process

- Both approaches have disadvantage that changing the identifier of a process may necessitate updation of all references to the old identifier.

Message Passing Systems

With indirect communication, the messages are not sent directly from sender to receiver but to a shared data structure consisting of queues (mailbox) that can temporarily hold.

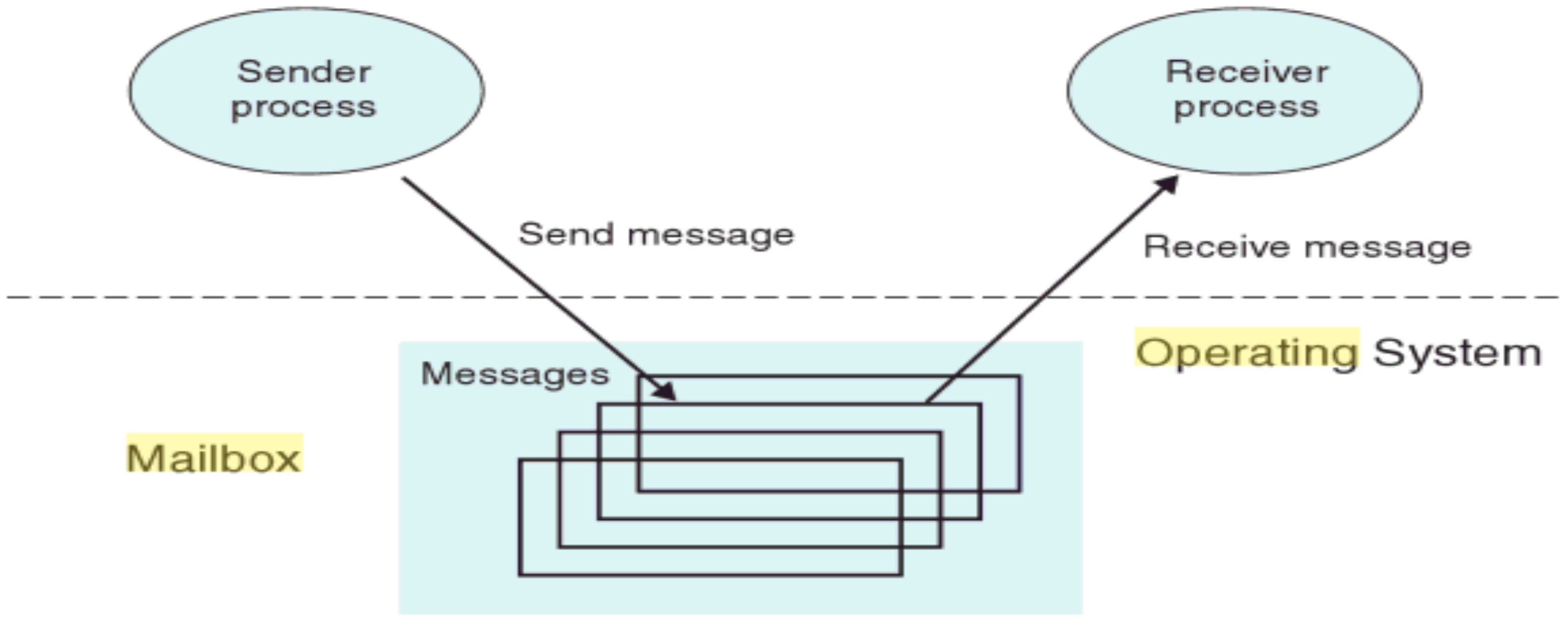


Figure 6.10 Two processes communicating via a mailbox.

Message Passing Systems

- When communicating one process sends message to a mailbox and other process receives it from mailbox.
- Each mailbox has a unique identification. For example pipes

- send(A, message)—Send a message to mailbox A.
- receive(A, message)—Receive a message from mailbox A.

Message Passing Systems

2.Synchronization: Message passing may be either blocking or nonblocking— also known as synchronous and asynchronous.

- ***Blocking send***. The sending process is blocked until the message is received by the receiving process or by the mailbox.
- ***Nonblocking send***. The sending process sends the message and resumes operation.
- ***Blocking receive***. The receiver blocks until a message is available.
- ***Nonblocking receive.*** The receiver retrieves either a valid message or a null.

Message Passing Systems

3.Buffering: Direct or indirect, messages exchanged by communicating processes reside in a temporary queue which can be implemented in three ways:

- ***Zero capacity***. The queue has a maximum length of zero.In this case, the sender must block until the recipient receives the message.
- ***Bounded capacity***. The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting, however. If the link is full, the sender must block until space is available in the queue.
- ***Unbounded capacity.*** The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

Message Passing Systems

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.