Chapter 3: Processes

Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems

Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.

- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.

- Describe and contrast interprocess communication using shared memory and message passing.
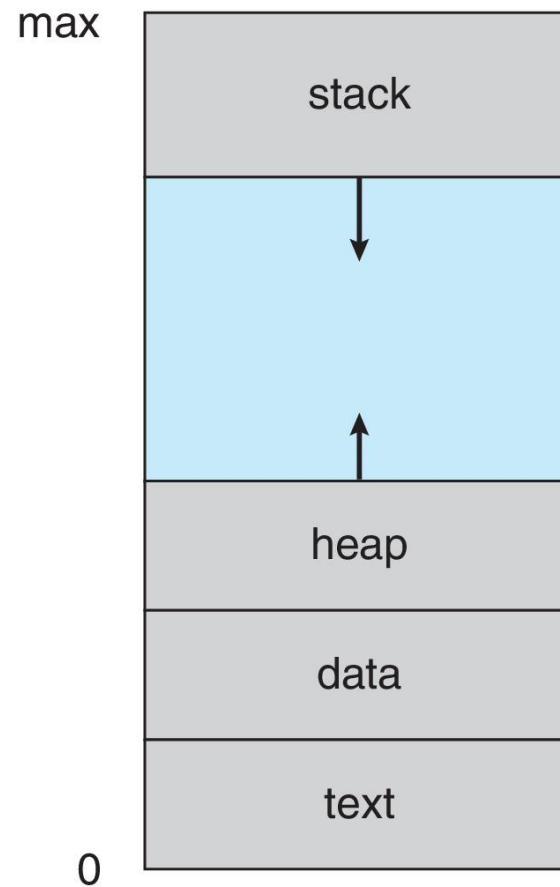
Process Concept

- 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
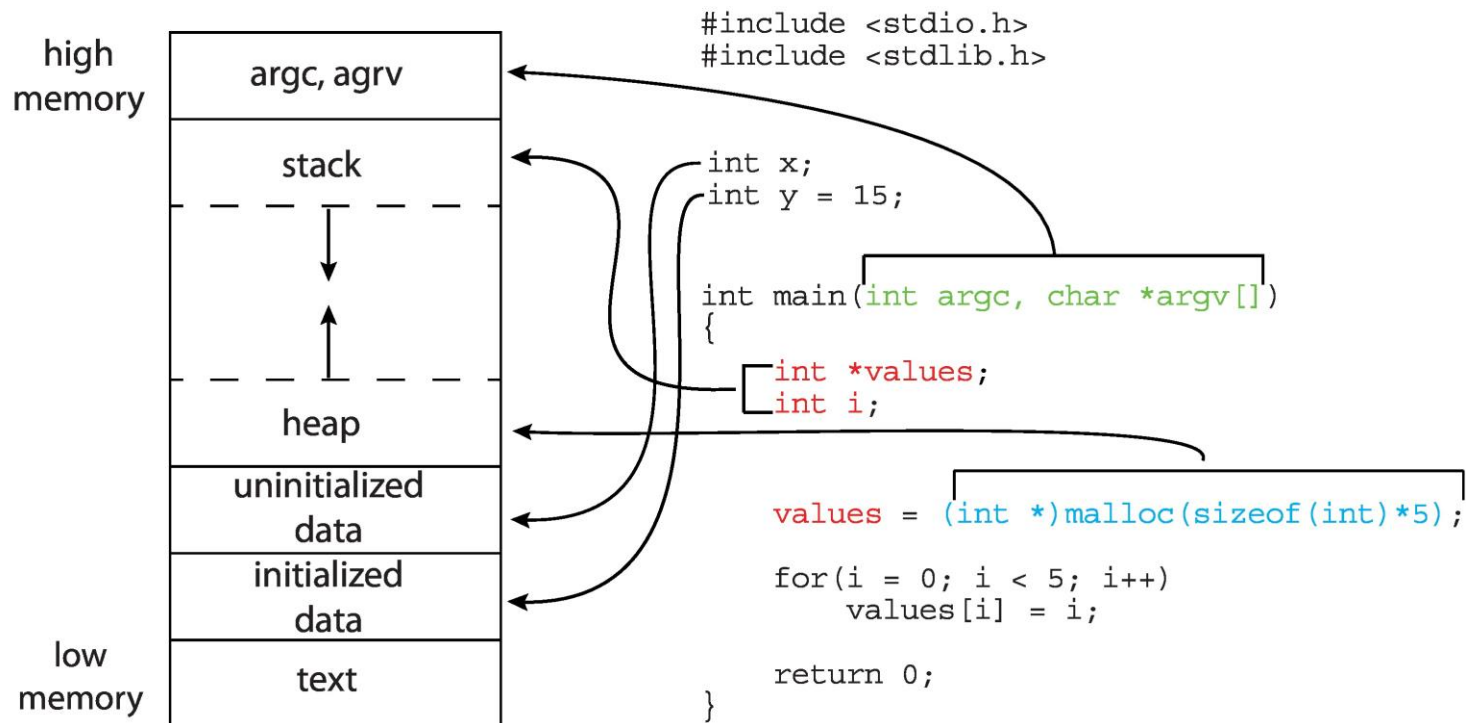
Process Concept (Cont.)

- 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.

Memory Layout of a C Program


```c
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

Memory regions (high to low memory): argc, agrv | stack | heap | uninitialized data | initialized data | text
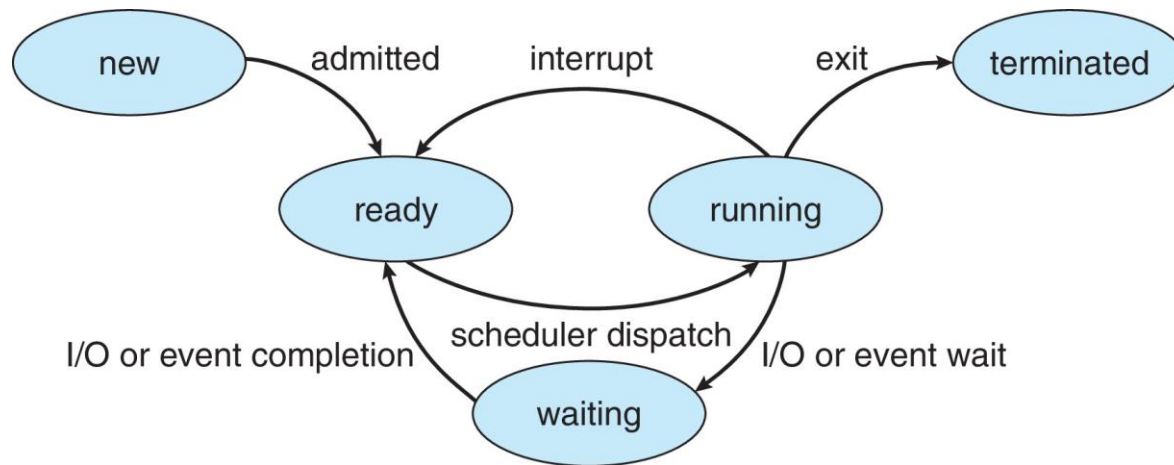
Process State

- 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(also called **task control block**)

- Process state – running, waiting, etc.

- Program counter – location of instruction to next execute

- CPU registers – contents of all process-centric registers

- CPU scheduling information- priorities, scheduling queue pointers

- Memory-management information – memory allocated to the process

- Accounting information – CPU used, clock time elapsed since start, time limits

- I/O status information – I/O devices allocated to process, list of open files

process state
process number
program counter
registers
memory limits
list of open files
• • •

Threads

- So far, process has a single thread of execution

- Consider having multiple program counters per process

  - Multiple locations can execute at once

    - Multiple threads of control -> **threads**

- Must then have storage for thread details, multiple program counters in PCB

- Explore in detail in Chapter 4

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)

- Most processes can be described as either I/O bound or CPU bound
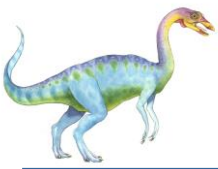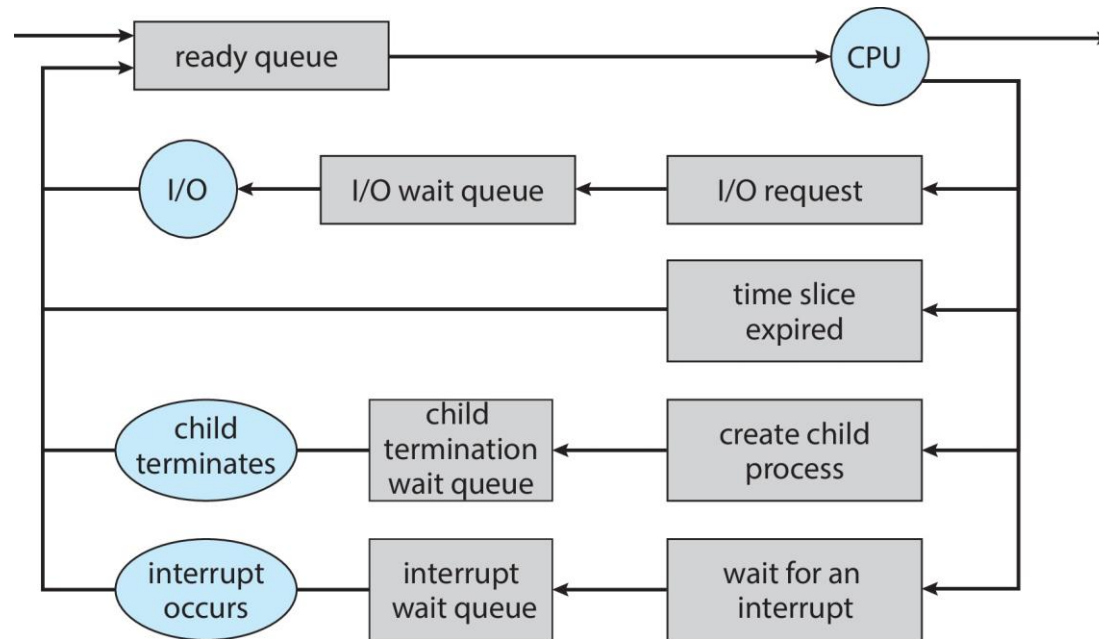
Ready and Wait Queues
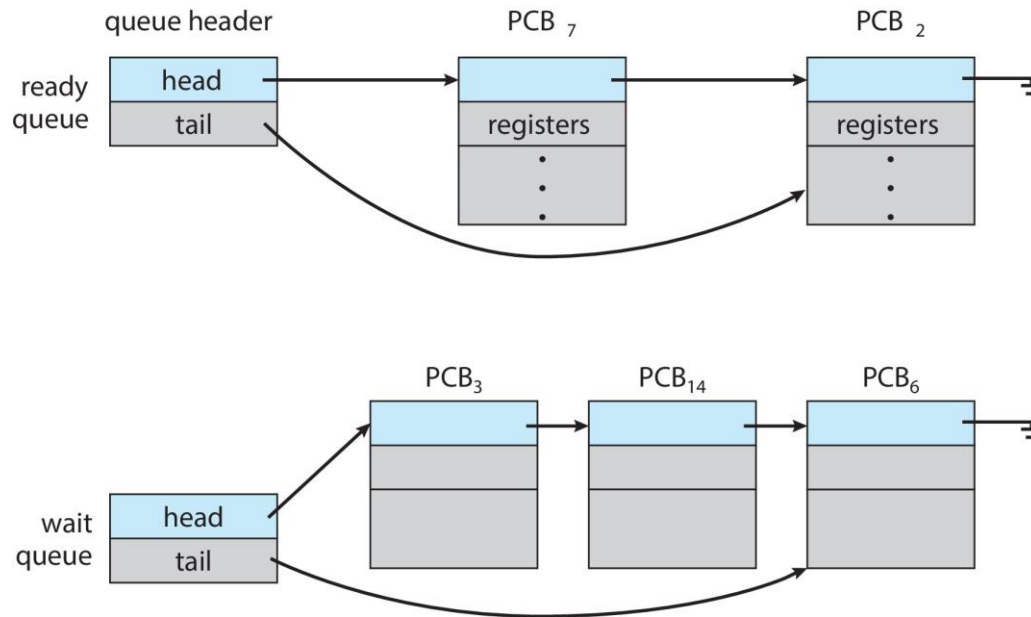
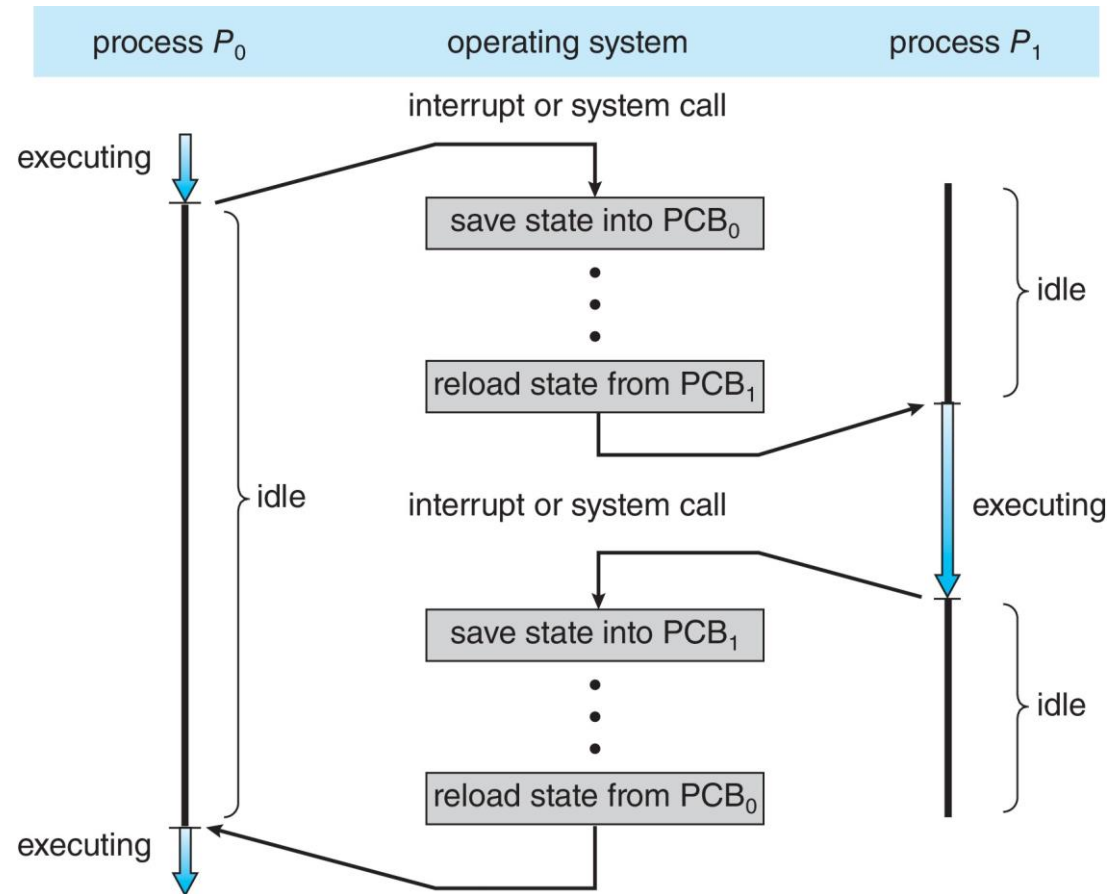# Representation of Process Scheduling

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

- 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.

| process P_0 | operating system | process P_1 |

interrupt or system call

executing

save state into PCB_0

reload state from PCB_1

idle

idle

interrupt or system call

executing

save state into PCB_1

reload state from PCB_0

idle

executing

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

  - Parent and children share all resources

  - Children share subset of parent's resources

  - Parent and child share no resources

- Execution options

  - 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
  - `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
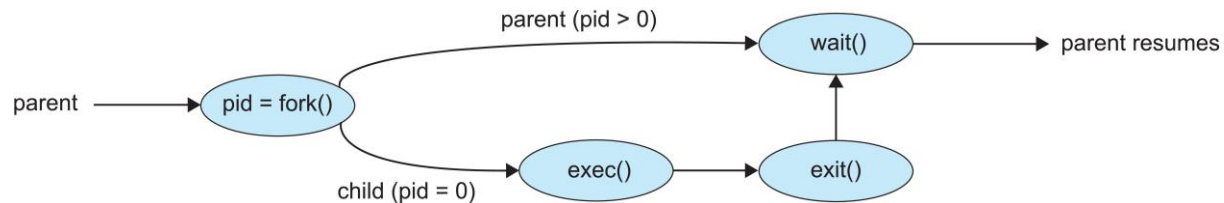
C Program Forking Separate Process

```c
#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, then all its children must also be terminated.

  - **cascading termination.** All children, grandchildren, etc., are terminated.

  - The termination is initiated by the operating system.

- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process

  ```
  pid = wait(&status);
  ```

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
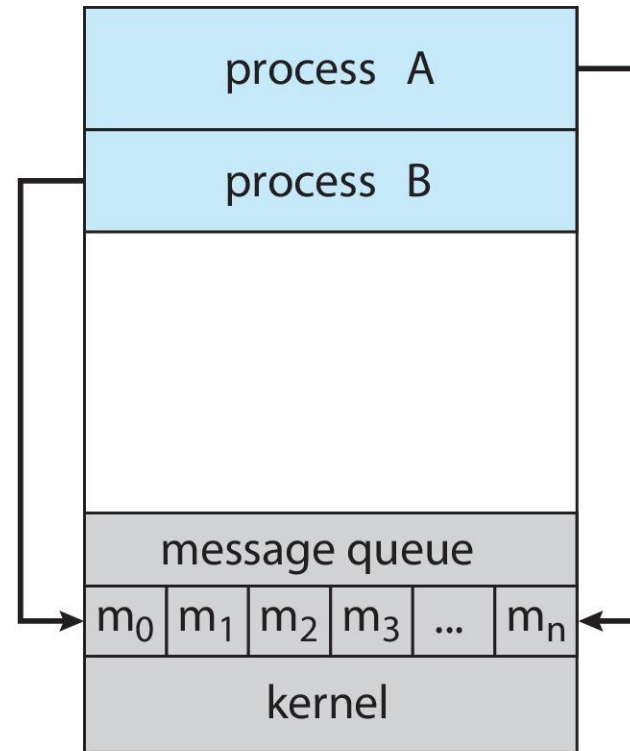
  - **Shared memory**

  - **Message passing**

Communications Models

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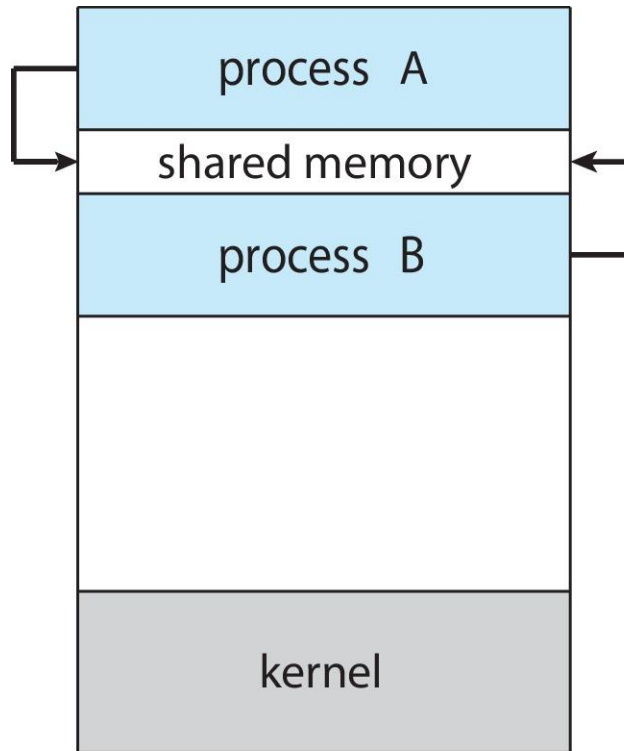
(a)   (b)

Producer-Consumer Problem

- Paradigm for cooperating processes:

  - *producer* process produces information that is consumed by a *consumer* process

- Two variations:

  - **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 that 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;
```

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

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

Producer Process – Shared Memory

```
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;
}
```

Consumer Process – Shared Memory

```
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 */

}
```

What about Filling all the Buffers?

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.

- We can do so by having an integer `counter` that keeps track of the number of full buffers.

- Initially, `counter` is set to 0.

- The integer `counter` is incremented by the producer after it produces a new buffer.

- The integer `counter` is and is decremented by the consumer after it consumes a buffer.

Producer

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

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

Consumer

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

IPC – Message Passing

- 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

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive

- Implementation issues:

  - 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?

Implementation of Communication Link

- Physical:
  - Shared memory
  - Hardware bus
  - Network
- Logical:
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering

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 (Cont.)

- Operations
    - Create a new mailbox (port)
    - Send and receive messages through mailbox
    - Delete 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 (Cont.)

- 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** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message

Producer-Consumer: Message Passing

- Producer
```
message next_produced;
while (true) {
   /* produce an item in next_produced */

   send(next_produced);
}
```

- Consumer
```
message next_consumed;
while (true) {
   receive(next_consumed)

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

Buffering

- Queue of messages attached to the link.

- Implemented in one of three ways

    1. Zero capacity – no messages are queued on a link.
       Sender must wait for receiver

    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