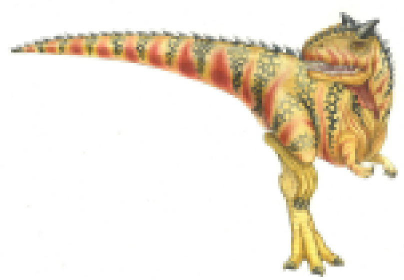
Chapter 3: Processes



Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes

- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems

Objectives

- Identify the separate components of a process and illustratehowtheyare represented and scheduled in an operating system.
- Describe how processes

are created and terminated inanoperatingsystem, including developing programs using the appropriatesystemcalls that perform these operations.

- Describe and contrast interprocess communication usingsharedmemory and message passing.
- Design programs that uses pipes and POSIX shared memorytoperform interprocess communication.
- Describe client-server communication using sockets andremoteprocedure calls.
  - Design kernel modules that interact with the Linux operatingsystem.

Process Concept

- An operating system executes a variety of programs that runasaprocess.

- **Process** – a program in execution; process execution must progressin sequential fashion. No parallel execution of instructionsof asingleprocess
  - Process memory is divided into following sections: • **Stack** containing temporary data
    - 4 Function parameters, return addresses, local variables
  - • **Heap** containing memory dynamically allocated duringruntime• The program code, also called **text section** • **Data section** containing global variables
    - • Current activity including **program counter**, processor registers

Process Concept (Cont.)

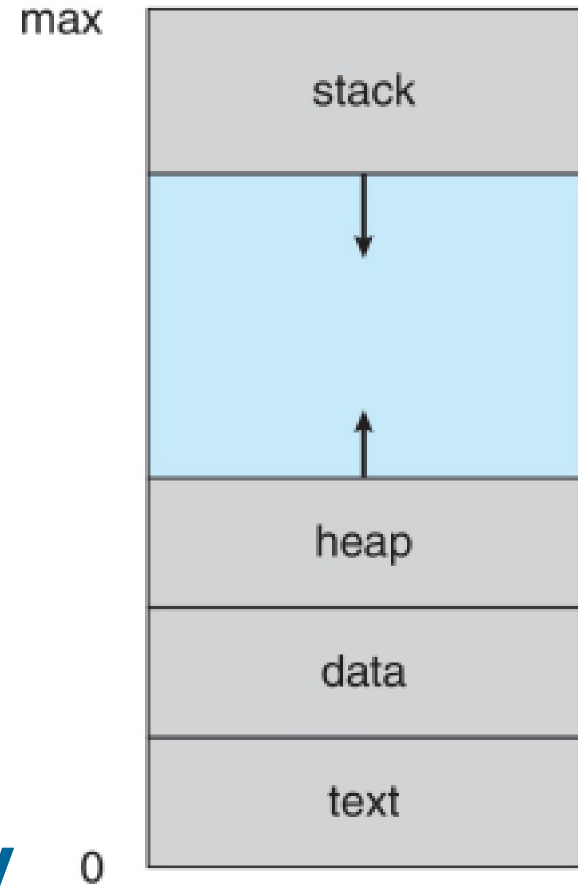
- Program is **passive** entity stored on disk (**executable file**); process is **active**

- Program becomes process when an executable fileisloaded into memory

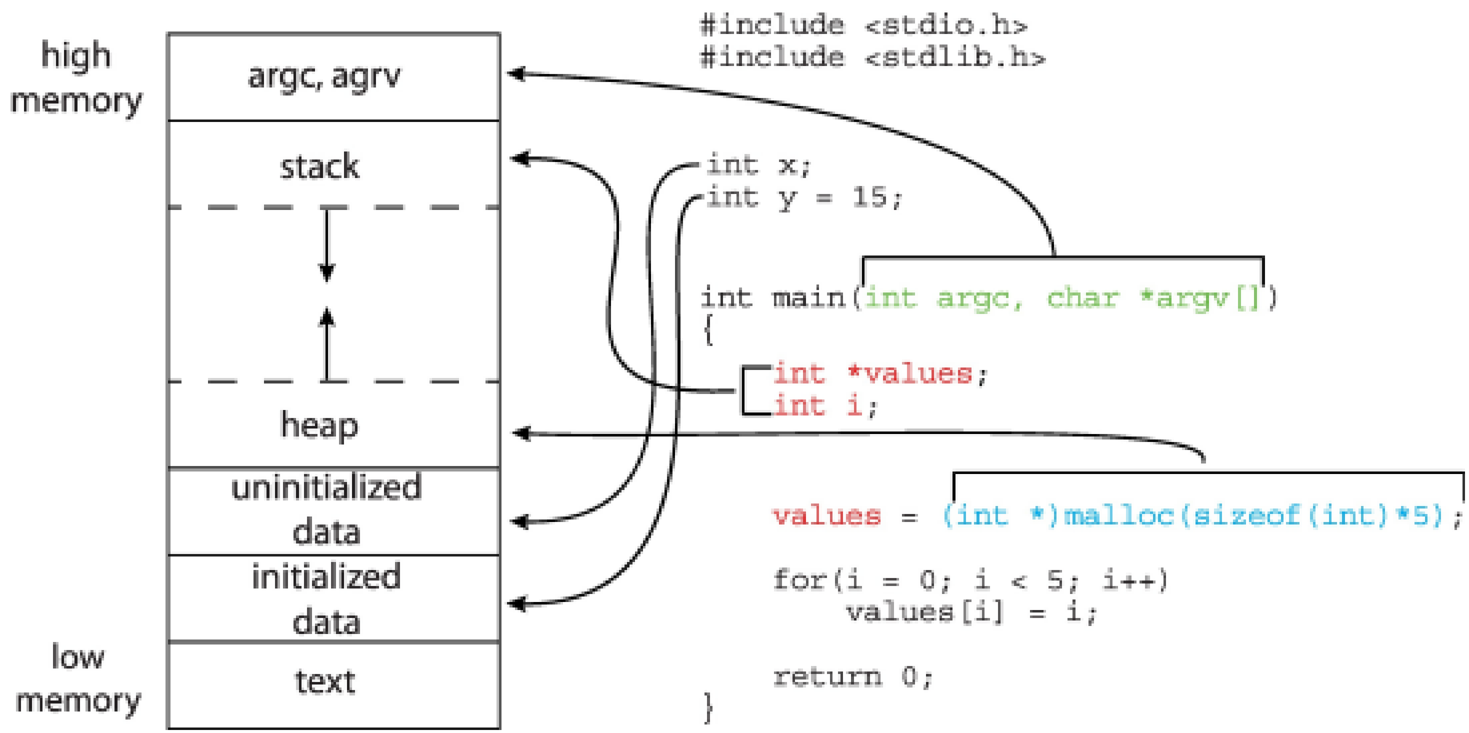
- Execution of program started via GUI mouse clicks, commandline entry of its name, etc.
- One program can be several processes
  - Consider multiple users executing the same program4 Compiler

    4 Text editor

Process inMemory



Memory Layout of aCProgram



Process State

- A Process has 5 states. Each process may be in one of thethesestates
- As a process executes, it changes **state** • **New**: The process is

  being created

  - **Ready**: The process has all the resources it needs torun.Theprocess is waiting to be assigned to a processor
  - **Running**: Instructions are being executed • **Waiting**: The process is waiting for some event tooccur. Forexample the process may be waiting for keyboardinput, diskaccess request, inter-process messages, a timer togooff, orachild process to finish.
  - **Terminated**: The process has finished execution

Diagramof ProcessState

Process Control Block(PCB)

Information associated with each process(also called **taskcontrol block**)

- Process state – state of the process. Eg. running, waiting, etc.
- Program counter – location of instruction to next execute

- CPU registers – contents of all process-centric registers. Eg. Accumulator, index regs., SP, GPRs
- CPU scheduling information- priorities, scheduling queue pointers, other scheduling parameters
- Memory-management information – memory allocated to the process, page/segment tables
- 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

Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - 4 Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple programcounters in PCB
- Explore in detail in Chapter 4

Process Scheduling

- **Process scheduler** selects among available processes for next executionon CPU core.
- Goal -- Maximize CPU use by keeping CPU busy all thetime, quicklyswitch processes onto CPU core
- Maintains **scheduling queues** of processes • **Job queue**– set of processes admitted to the system• **Ready queue** – set of all processes residing in

mainmemory, readyand waiting to execute

• **Wait queues** – set of processes waiting for an event (i.e., I/O)• Processes migrate among the various queuesore

- Queues are stored as linked list of PCBs. Queue header contains2pointers- head pointer pointing to first PCB and tail pintingtothelastPCBin the queue.

Ready and Wait Queues

Representation of ProcessScheduling

- A common representation of process scheduling is a queueingdiagram.▪ Each rectangular box in the diagram represents a queue. Twotypesofqueues are present: the ready queue and a set of devicequeues. ▪ The circles represent the resources that serve the queues, andthearrowsindicate the flow of processes in the system.

- A new process is initially put in the ready queue. It waits inthereadyqueue until it is selected for execution and is given theCPU. Oncetheprocess is allocated the CPU and is executing, one of several eventscouldoccur:

 The process could issue an I/O request, and then be placedinanI/Oqueue.

 The process could create a new subprocess and wait for itstermination. The process could be removed forcibly fromthe CPU, asaresult ofaninterrupt, and be put back in the ready queue.

Representation of ProcessScheduling

CPU Switch FromProcesstoProcess

A **context switch** occurs when the CPU switchesfromone process to another.

Context Switch

- When CPU switches to another process, the systemmust **savethe state** of the old process and load the **saved state**for thenew process via a **context switch**
- **Context** of a process represented in the PCB ▪ Context-switch time is pure overhead; the systemdoes nouseful work while switching
    - The more complex the OS and the PCB the longer thecontext switch
- Time dependent on hardware support
    - Some hardware provides multiple sets of registers per CPU multiple contexts loaded at once

Schedulers

Schedulers are software which selects an available programtobeassignedto CPU.

- **Long term scheduler or Job scheduler**: selects processesfromthepool (of secondary memory disk) and loads theminto thememoryforexecution.

□ It runs infrequently

□ will be invoked only when a process leaves the system□ can take time to select the next process because of thelonger timebetween the executions.

□ controls **degree of multiprogramming** - number of processesinmemory

- **Short-term scheduler, or CPU Scheduler:** selects fromamongtheprocesses that are ready to execute and assigns the CPUtoit. □ It must select the new process for CPU frequently. □ Must execute atleast once every 100ms □ Must execute very fast.

Schedulers

Processes can be described as either:

• I/O-bound process – spends more time doing I/Othancomputations,• CPU-bound process – spends more time doing computationsandfewI/Ooperations.

An efficient scheduling system will select a good mix of CPU-boundprocesses and I/O bound processes. If the scheduler selectsmoreI/Oboundprocess, then I/O queue will be full and ready queue will beempty. • On some systems (time sharing UNIX), long termscheduler maybe absent, all new processes are submitted to memory for theshort termscheduler.

 the system stabilty depends on physical limitations (no. of terminals)oradjusting nature of human users

▪ Time sharing systems employ a **Medium-termscheduler:** It swapsoutthe process from ready queue and swap in the process toreadyqueue. When system loads get high, this scheduler will swap oneor moreprocesses out of the ready queue for a few seconds, inorder toallowsmaller faster jobs to finish up quickly and clear the system.

Schedulers

Advantages of medium-term scheduler –

- To remove process from memory and thus reduce thedegreeof multiprogramming (number of processes in memory).
- To make a proper mix of processes(CPU bound and I/Obound)

Operations onProcesses

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

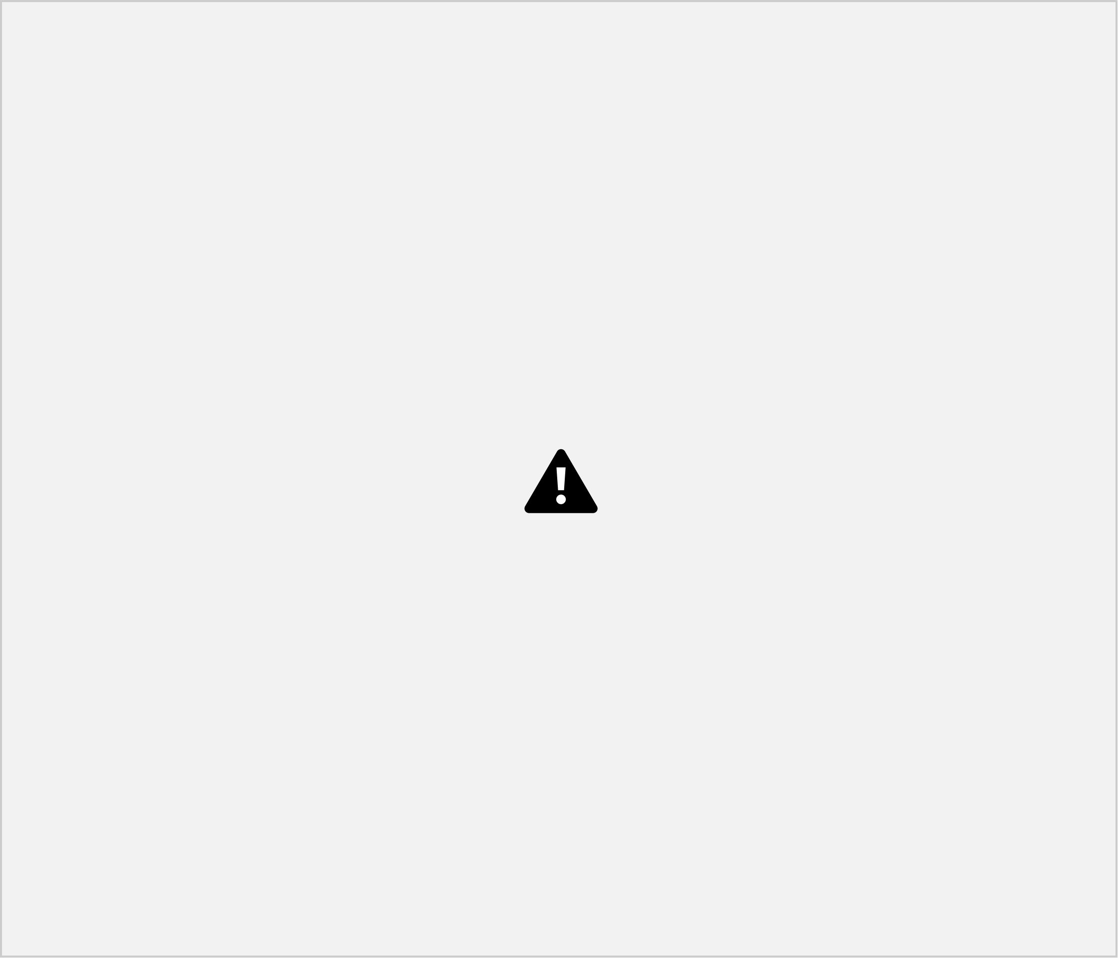
Process Creation

- **Parent** process create **children** processes, which, in turncreate other processes, forming a **tree** of processes

- Generally, process identified and managed via a **processidentifier** (**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

- Process tree on Solaris system



A Tree of ProcessesinLinux

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 replacetheprocess'memory space with a new program
- Parent process calls **`wait()`** waiting for the child toterminate

Process Creation(Cont.)

C ProgramForkingSeparateProcess

Creating a Separate ProcessviaWindowsAPI

Process Termination

- Process executes last statement and then asks the operatingsystem to delete it using the **`exit()`** systemcall.
  - Returns status data from child to parent (via **`wait()`**)
  - Process'resources are deallocated by operating system
- Parent may terminate the execution of children processesusingthe **`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 doesnot allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its childrenmust also be terminated.
  - **cascading termination.** All children, grandchildren, etc., areterminated.
    - The termination is initiated by the operating system.
- The parent process may wait for termination of a child processbyusing the **`wait()`** system call. The call returns status information and the pid of the terminated process **`pid = wait(&status);`**
- If no parent waiting (did not invoke **`wait()`**) process is a**zombie.** It has completed but it still has an entry in theprocesstable. They do not use any resources but retain their processid.
- If parent terminated without invoking **`wait()`**, processisan**orphan.** These are the processes that are still runningeventhough their parent process has terminated.

Interprocess Communication

- Processes executing may be either **co-operative** or **independent**processes.

 Independent Processes – processes that cannot affect other processes or be affected by other processes executinginthesystem. Cooperating Processes – processes that can affect other processesorbe affected by other processes executing in the system; includingsharing data

- Reasons for cooperating processes:
  - Information sharing: There may be several processeswhichneedto access the same file. So the information must beaccessibleatthe same time to all users.
  - Computation speedup: Often a solution to a problemcanbesolved faster if the problem can be broken down intosub-tasks,which are solved simultaneously (particularly whenmultipleprocessors are involved)

Interprocess Communication

- Modularity: A system can be divided into cooperatingmodulesandexecuted by sending information among one another. Asystemcan be constructed in a modular approach divngthesysstemfunctions into separate processes or threads.
- Convenience: Even a single user can work on multipletaskbyinformation sharing. Eg: A user may be editing, compilingandprinting at the same time.

▪ Cooperating processes need **interprocess communication**(**IPC**)▪

Two models of IPC

- **Shared memory** (under the control of users): shareacommonbuffer pool, and the code for implementing buffer isexplicitlywritten by user.
- **Message passing** (under the control of OS): OSprovidesmeansof communication between cooperating processes.

CommunicationsModels

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

CommunicationsModels

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

Shared Memory Vs. Messagepassing

Shared Memory

• A region of memory is shared by communicatingprocesses,into which the information is written andread• Useful for sending large block of data • System call is used only to create sharedmemory• Message is sent faster, as there are no systemcalls

Message passing

• Message exchange is done among the processesbyusingobjects.
• Useful for sending small data.
• System call is used during every read andwriteoperation.• Message is communicated slowly.

Shared Memory

- An area of memory shared among the processesthat wishtocommunicate
- A region of shared-memory is created within theaddressspaceof a process, which needs to communicate.
- Other processes that needs to communicateusesthissharedmemory.
- The form of data and position of creating sharedmemoryareais decided by the process.
- The communication is under the control of theusersprocessesnot the operating system.
- Major issues is to provide mechanismthat will allowtheuserprocesses to synchronize their actions whentheyaccessshared memory.

Producer-Consumer Problem

- Paradigm for cooperating processes:
  - The data is passed via an intermediary buffer (shared memory).
  - ***producer*** process produces information that is consumedbya***consumer***process.
  - A producer can produce one item while the consumer is consuminganother item.
  - The producer and consumer must be synchronized, sothat theconsumerdoes not try to consume an item that has not yet been produced.
- Two variations:
  - **unbounded-buffer** places no practical limit on the sizeof thebuffer:
    - 4 Producer never waits
    - 4 Consumer may have to wait for new items.
  - **bounded-buffer** assumes that there is a fixed buffer size
    - 4 Producer must wait if all buffers are full
    - 4 Consumer waits if there is no item to consume

Bounded-Buffer – Shared-MemorySolution

- This example uses shared memory as a circular queue. Theinandoutaretwo pointers to the array. Note in the code belowthat onlytheproducerchanges "in", and only the consumer changes "out".
- Set up Shared Memory

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

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

Producer Process–SharedMemory

Note that the buffer is full when [(in+1)%BUFFER_SIZE==out]

```
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–SharedMemory

Note that the buffer is empty when [in == 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 */ }
```

This scheme allows at most BUFFER_SIZE-1 i.e., 9 items in the bufferatthesame time .

What about Fillingall theBuffers?

- 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 keepstrackof 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– MessagePassing

- Processes communicate with each other without resortingtosharedvariables
- Particularly useful in distributed environment wherethecommunicating processes may reside on different computersconnected with a network.
- IPC provides message passing with two operations: •

send(*message*)

- **receive**(*message*)

- The *message* size is either fixed or variable ▪ fixed size message : the s/m level implementation is simplebut thetask of programming becomes difficult.
- variable sized message : more complex s/mlevel implementationbutprogramming task becomes simpler.

Message Passing(Cont.)

- If processes *P* and *Q* wish to communicate, they needto: • 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 accommodatefixed or variable?
- Is a link unidirectional or bi-directional?

Implementation of CommunicationLink

- Physical:
  - Shared memory
  - Hardware bus
  - Network

- Logical: methods to logically implement a link and send/receiveoperations are
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering

Direct Communication

- Naming: The processes that want to communicate shouldhaveawayto refer eachother. (using some identity)
- Processes must name each other explicitly:

- **`send`** (*P, message*) – send a message to process P• **`receive`**(*Q, message*) – receive a message fromprocessQ▪ 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

Direct Communication

▪ Types of addressing in direct communication – ▪ Symmetric addressing – the above described communicationissymmetric communication. Here both the sender and thereceiverprocesses have to name each other to communicate.

- Asymmetric addressing – Here only the sender name is mentioned,but the receiving data can be from any system.

send(P, message) --- Send a message to process Preceive(id, message). Receive a message fromanyprocess- Disadvantages of direct communication – any changes intheidentifierof a process, may have to change the identifier inthewholesystem(sender and receiver), where the messages aresentandreceived.

Indirect Communication

- uses shared mailboxes or ports
- Messages are directed and received frommailboxes (alsoreferredto as ports)

- Each mailbox has a unique id
- Processes can communicate only if they share a mailbox▪ Mailbox is an object into which messages can be sent andreceived.It has a unique ID. Using this identifier messages are sent andreceived.

▪ Two processes can communicate only if they have a sharedmailbox.

▪ The send and receive functions are –

send(A, message) – send a message to mailbox Areceive(A,

message) – receive a message frommailboxA

Indirect Communication(Cont.)

▪ Properties of communication link

- A link is established between a pair of processes onlyif theyhave a shared mailbox

- A link may be associated with many processes • Each pair of processes may share several communicationlinks• Link may be unidirectional or bi-directional

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

Indirect Communication(Cont.)

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 each execute *receive* fromA• 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 receiveoperation• Allow the system to select arbitrarily the receiver (either P2orP3). 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 themessageisreceived

- **Blocking receive** -- the receiver is blocked until amessageisavailable

- Non-blocking is considered **asynchronous** • **Non-blocking send** -- the sender sends the messageandresumes opeeration

- **Non-blocking receive** -- the receiver receives: 4 A valid

  message, or

  4 Null message

- Different combinations possible

- If both send and receive are blocking, we havea**rendezvous**

Producer-Consumer: MessagePassing

- 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

- When messages are passed, a temporary queue is created. Suchqueue can be of three capacities:

- Queue of messages attached to the link.
- Implemented in one of three ways
    1. Zero capacity – The buffer size is zero (buffer does not exist). Messages are not stored in the queue. The senders must blockuntil receivers accept the messages.
    2. Bounded capacity – The queue is of fixed size(n). Sendersmustblock if the queue is full. After sending ‘n’ bytes thesender isblocked.
    3. Unbounded capacity – The queue is of infinite capacity. Thesender never blocks.

Examples of IPCSystems-POSIX

- POSIX Shared Memory

- Process first creates shared memorysegment`shm_fd = shm_open(name, O CREAT | ORDWR,0666);`
  - O_CREAT | O_RDWR : the shared-memory object istobecreated if it does not yet exist (O_CREAT) and that theobjectisopen for reading and writing (O_RDWR).
  - O_RDONLY: Open for read access only.
  - O_RDWR: Open for read or write access.
  - The last parameter establishes the directory permissionsof theshared-memory object.
- Set the size of the object

```
ftruncate(shm_fd, 4096);
```


Examples of IPCSystems- POSIX

Mapping a Shared Memory Object • Use **`mmap()`** to memory-map a file pointer tothesharedmemory object

• Reading and writing to shared memory is donebyusingthepointer returned by **`mmap()`**.

pa=mmap(addr, len, prot, flags, fildes, off) • The mmap() - establishes a mapping betweentheaddressspace of the process at an address “pa” for “len” bytestothememory object represented by the file descriptor “fildes”atoffset “off” for “len” bytes.

Eg: psm = mmap(0, 1024, PROT_WRITE, MAP_SHARED, shmid1,0);

Examples of IPCSystems- POSIX

- Suppose process 1 and process 2 have successfully attached the shared memory segment.
- This shared memory segment will be part of their address space, although the actual address could be different (i.e., the starting address of this shared memory segment in the address space of process 1 may be different from the starting address in the address space of process 2).

IPCPOSIXProducer

IPC POSIXConsumer

Operating

shmget()

- **shmget()** is used to obtain a shared memory identifier ▪ #include<sys/types.h>
- #include<sys/ipc.h>
- #include<sys/shm.h>
- int shmget(key_t key, int size, int flag);
  - shmget() returns a shared memory ID if OK, -1 on error ▪ Key is typically the constant “IPC_PRIVATE”, which lets thekernel choose a new key – keys are non-negative integer identifier, but unlikefds they are system-wide, and their value continually increasestoamaximum value, where it then wraps around to zero ▪ Size is the size of shared memory segment in bytes ▪ Flag can be “SHM_R”, “SHM_W” or “SHM_R | SHM_W”

shmat()

- **shmat()**
- Once a shared memory segment has been created, a processattachesit to its address space by calling shmat():
- void *shmat(int shmid, void* addr, int flag);
- shmat() returns a pointer to shared memory segment if OK, -1onerror▪

shmdt()

- shmdt() detaches the shared memory segment locatedat theaddressspecified by shmaddr from the address space of the callingprocess.▪ **shmctl()** performs the control operation specified by cmdontheshared memory segment whose identifier is given in shmid. ▪ if cmd is IPC_RMID then Mark the segment to be destroyed. Thesegment will actually be destroyed only after the last processdetachesit. The caller must be the owner or creator of the segment, or beprivileged.

Endof Chapter3