Process Concurrency

Ms. Swati Mali

swatimali@somaiya.edu

Assistant Professor

Department of Computer Engineering

K. J. Somaiya College of Engineering

Somaiya Vidyavihar University

3	**Process Concurrency**		10	
	3.1	Concurrency: Principles of Concurrency, InterProcess Communication, Process/Thread Synchronization.		
	3.2	Mutual Exclusion: Requirements, Hardware Support, Operating System Support (Semaphores and Mutex), Programming Language Support (Monitors)		CO3
	3.3	Classical synchronization problems: Readers/Writers Problem, Producer and Consumer problem.		
	3.4	Principles of Deadlock: Conditions and Resource Allocation Graphs, Deadlock Prevention, Deadlock Avoidance: Banker"s Algorithm for Single & Multiple Resources, Deadlock Detection and Recovery. Dining Philosophers Problem		

Why concurrency?

- **Multiprogramming**

- **Multiprocessing**

- **Distributed processing**

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya
TRUST

Processes

- ## Independent processes
  - o Independent processes don't affect or impact any other processes in the system.

- ## Cooperating processes
  - ○ Cooperating processes can affect or be affected by other processes in the system. For example, multiple processes may need to access the same file.
  - o e.g. Producer-consumer

Reading Assignment : https://pages.cs.wisc.edu/~bart/537/lecturenotes/s4.html

Interprocess Communication

- Reasons for cooperating processes:
  - Modularity
    - We may want to construct the system in a modular fashion,
    - dividing the system functions into separate processes or threads,
  - Convenience
    - Even an individual user may work on many tasks at the same time.
    - For instance, a user may be editing, printing, and compiling in parallel.

Concurrency

Concurrency encompasses issues including

- communication among processes,

- sharing of and competing for resources (such as memory, files, and I/O access),

- synchronization of the activities of multiple processes,

- and allocation of processor time to processes.

05-12-2022

K J Somaiya College of Engineering

Concurrency contexts

- **Multiple applications:** processing time is dynamically shared among a number of active applications.

- **Structured applications:** some applications can be effectively programmed as a set of concurrent processes.

- **Operating system structure:** operating systems are themselves often implemented as a set of processes or threads.

Process Synchronization

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution

- Concurrent access to shared data may result in data inconsistency

- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

process synchronization vs process concurrency

- Concurrency can be achieved by using multiple processors, cores, or threads, depending on the level of parallelism that you want to achieve.
- Synchronization is the coordination of the concurrent tasks, to ensure that they do not interfere with each other, or access shared resources in an inconsistent or unsafe way

Concurrency and Aspects of concurrency control

- Mutual exclusion
- Synchronization
- Deadlock
- Starvation

K J Somaiya College of Engineering

Aspects of concurrency control

- Mutual exclusion : the ability of multiple processes (or threads) to share code, resources, or data in such a way that only one process has access to the shared object at a time

- Synchronization : the ability of multiple processes to coordinate their activities by the exchange of information.

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya
TRUST

Some Key Terms Related to Concurrency

- **Atomic operation:**
  - A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.

- **Critical section**
  - A section of code within a process that requires access to shared resources and that must not be executed while an

- **Deadlock**
  - A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something. other process is in a corresponding section of code.

- **Livelock**
  - A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.

- **Mutual exclusion**
  - The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.

- **Race condition**
  - A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.

- **Starvation**
  - A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

- **Data Coherence:** Data coherence refers to the consistency of data or variables stored in shared memory.

Classical problems in concurrency

- Readers/writer's problem

- Producer consumer problem

- Dining philosopher's problem

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya
TRUST

PRINCIPLES OF CONCURRENCY

- Difficulties:
  - The sharing of global resources is fraught with peril. (Race condition etc)
  - It is difficult for the OS to manage the allocation of resources optimally
  - It becomes very difficult to locate a programming error because results are typically not deterministic and reproducible

SOMAIYA
VIDYAVIHAR UNIVERSITY
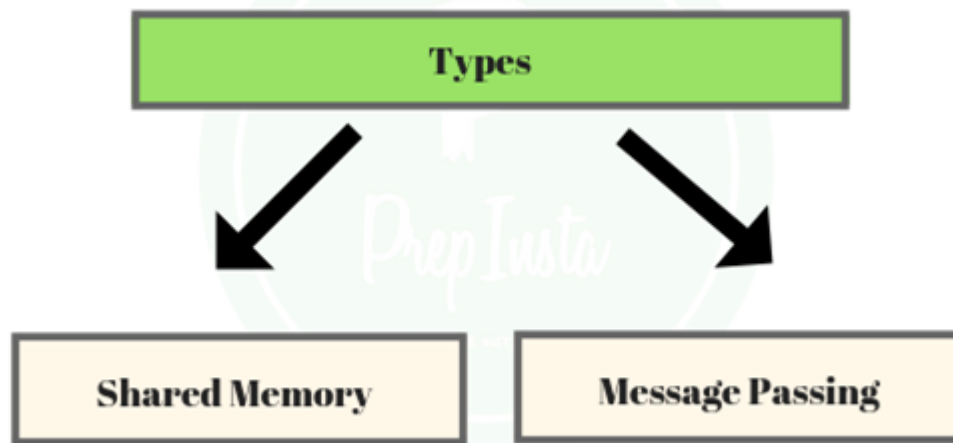K J Somaiya College of Engineering

Somaiya
TRUST

Interprocess Communication

- Cooperating processes need **interprocess communication** (**IPC**) mechanism to exchange data and information

Process P1 ←——— Interprocess Communication ———→ Process P2

K J Somaiya College of Engineering
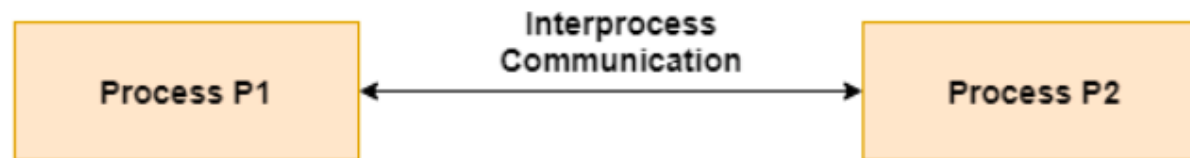
Interprocess Communication

- Two models of IPC
  o **Shared memory**
  o **Message passing**

K J Somaiya College of Engineering

Interprocess Communication

- 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.
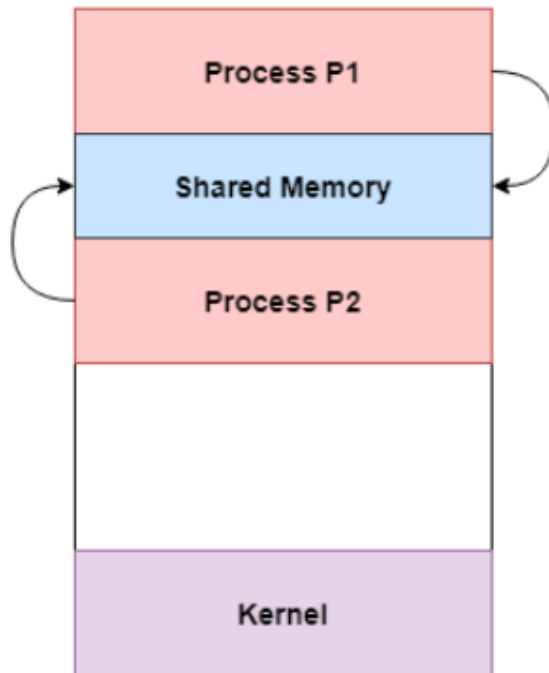
Communications Models

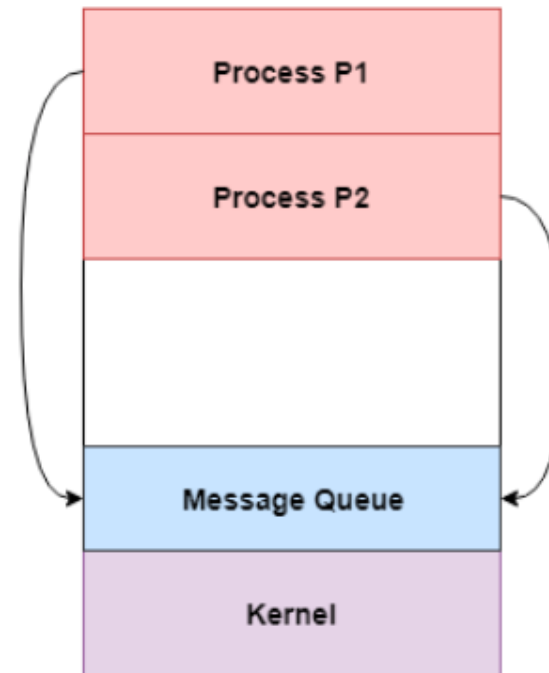**(**a) Message passing.   (b) shared memory.

(a)   (b)
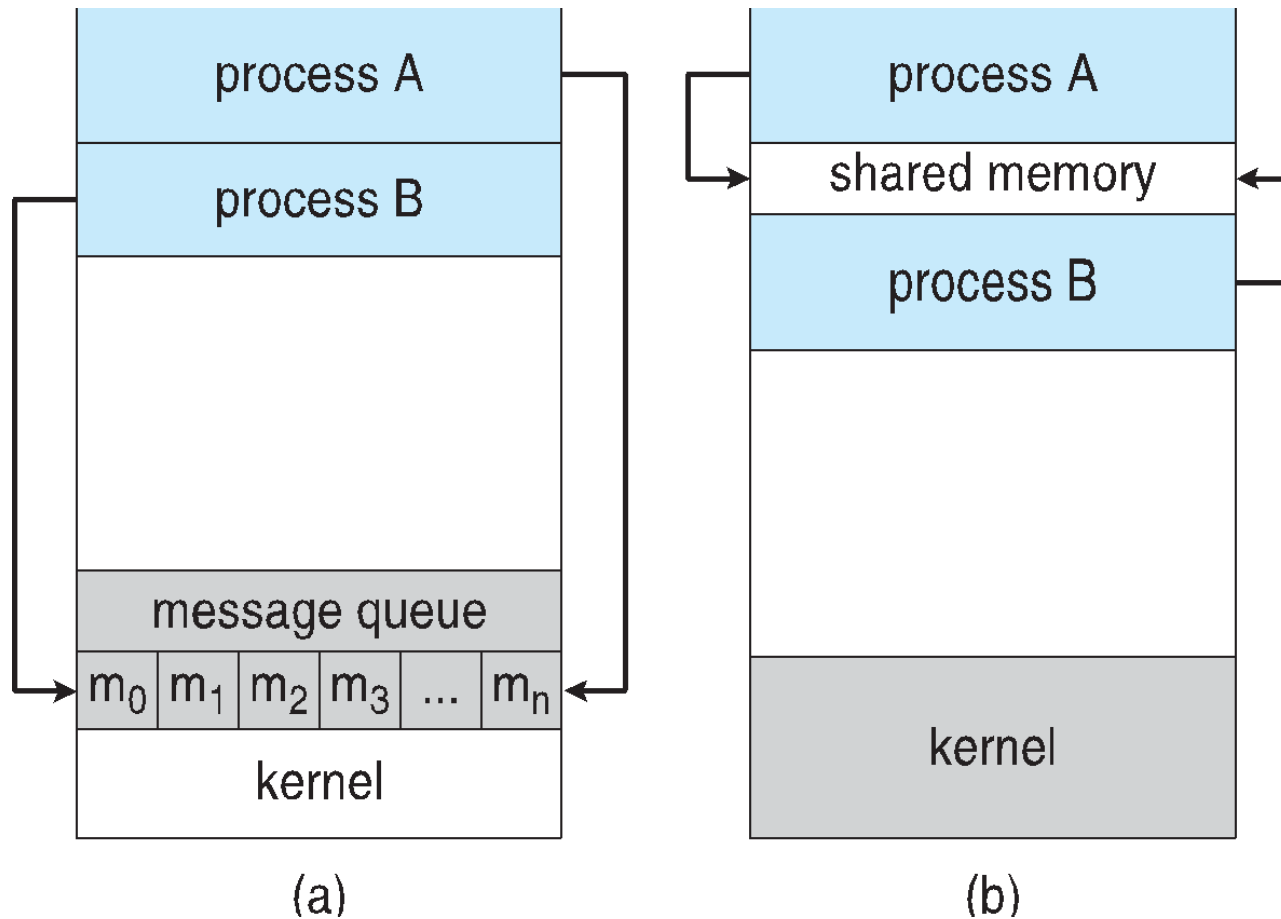
Communications Models

Approaches to Interprocess Communication

Shared Memory

Message Queue

K J Somaiya College of Engineering

Message Passing

Shared memory

- Message passing is useful for
  - **exchanging smaller** amounts of data,
  - because no conflicts need be avoided.
- Message passing is also easier to implement than
  - shared memory for intercomputer communication.

- Shared memory allows
  - **maximum speed** and convenience of communication
- **Shared memory is faster**
  - than message passing.

SOMAIYA

K J Somaiya College of Engineering

Somaiya
TRUST

Message Passing

Shared memory

- **Slower** As message passing systems are typically implemented
    - using system calls and
    - thus require the more time-consuming task of kernel intervention.

- Faster as 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.

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Producer-Consumer Problem

- Paradigm for cooperating processes,

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

K J Somaiya College of Engineering

Producer-Consumer Problem

For example,

1) A compiler may produce assembly code,
   - o which is consumed by an assembler.
   - o The assembler, in turn, can produce object modules,
   - o which are consumed by the loader.

2) Server as a producer and a client as consumer.
   - o For example, a Web server produces (that is, provides) HTML files and images,
   - o which are consumed (that is, read) by the client Web browser requesting the resource.

05-12-2022

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya
T R U S T

Producer-Consumer Problem

- **Solution= Shared memory.**

- To allow producer and consumer processes to run concurrently,

K J Somaiya College of Engineering

Producer-Consumer Problem

- An available 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

- A producer can produce one item
- While the consumer is consuming another item.
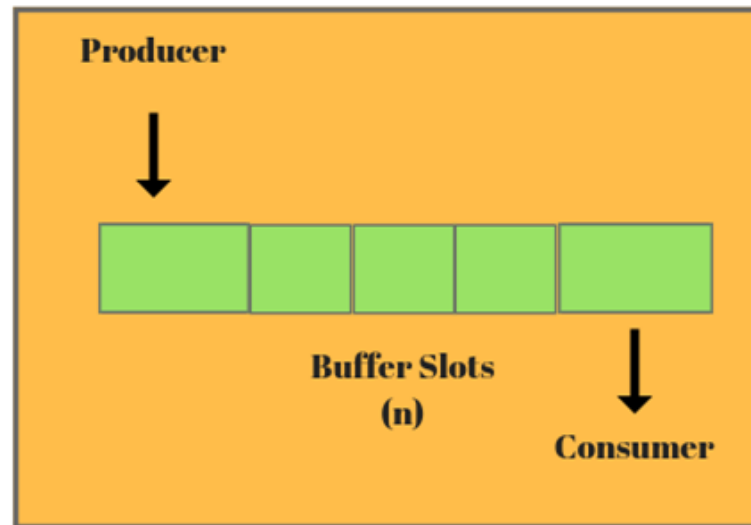
- **The producer and consumer must be synchronized,**
  - so that the consumer does not try to consume an item that has not yet been produced.

SOMAIYA
VIDYAVIHAR UNIVERSITY
Somaiya Vidyavihar
K J Somaiya College of Engineering

Somaiya
TRUST

Producer-Consumer Problem versions

- **Unbounded-buffer**
  - Places no practical limit on the size of the buffer
  - The consumer may have to wait for new items,
  - But the producer can always produce new items.

- **Bounded-buffer**
  - Assumes that there is a fixed buffer size
  - The consumer must wait if the buffer is empty,
 - and the producer must wait if the buffer is full.

K J Somaiya College of Engineering

Bounded-Buffer – Shared-Memory Solution

The shared buffer : circular array with two logical pointers: in and out.

    o in points to the next free position in the buffer;

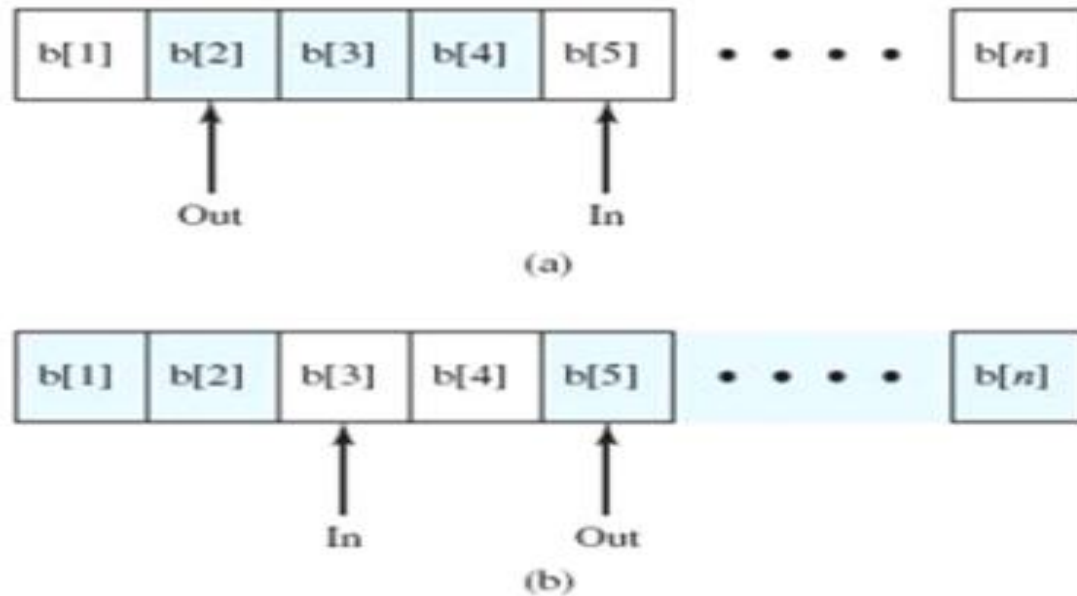   o out points to the first full position in the buffer.

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

item buffer[BUFFER_SIZE];
int in = -1;
int out = -1;
```
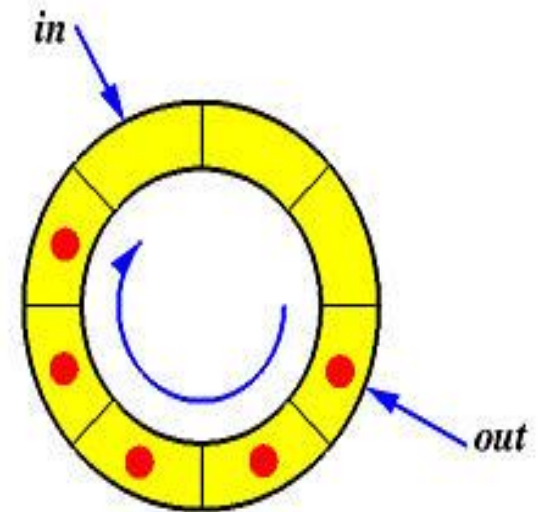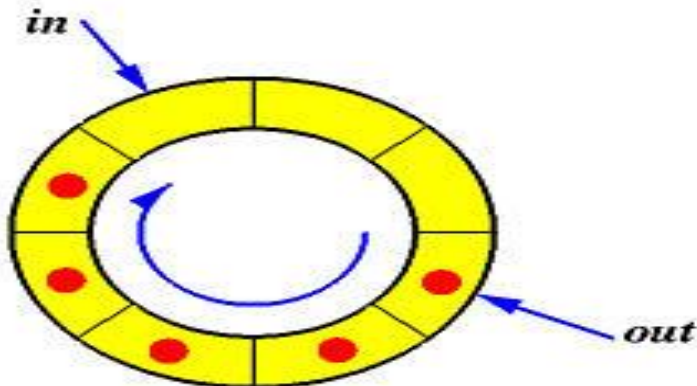
Bounded-Buffer – Shared-Memory Solution

- The buffer is empty when in== out = -1;
- The buffer is full when ((in+ 1)% BUFFER_SIZE) == out.

Bounded-Buffer – Producer

- The producer process has a local variable nextProduced
 - in which the new item to be produced is stored.

```
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;
}
```

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Bounded Buffer – Consumer

- The consumer process has a
  - local variable next Consumed in which the item to be consumed is stored.


```
item next_consumed;

while (true) {
        while (in == out==-1)

                ; /* do nothing */
        next_consumed = buffer[out];

        out = (out + 1) % BUFFER_SIZE;


        /* consume the item in next consumed */

}
```

K J Somaiya College of Engineering
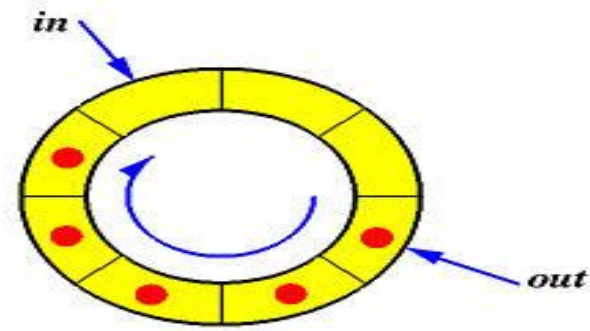
Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate

- The communication is under the control of the users processes not the operating system.

- The code for accessing and manipulating the shared memory be written explicitly by the application programmer.

- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

K J Somaiya College of Engineering

Interprocess Communication – Message Passing

- Another means for cooperating processes to communicate with each other
  - **via a message-passing facility.**

- Mechanism for processes
  - to communicate and
  - to synchronize their actions

- Message system – processes communicate with each other
  - **without resorting to shared variables or shared address space**

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Interprocess Communication – Message Passing

- 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.

- IPC facility provides two operations:
  - o **send**(*message*)
  - o **receive**(*message*)

- The *message* size is either fixed or variable

SOMAIYA
VIDYAVIHAR UNIVERSITY
Somaiya Vidyavihar
K J Somaiya College of Engineering

- If only fixed-sized messages can be sent,
    - the system-level **implementation is straightforward.**
    - the task of programming more difficult.

- Conversely, variable-sized messages require
    - a more **complex system-level implementation,**
    - but the programming task becomes simpler.

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

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

K J Somaiya College of Engineering

- 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?

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya
TRUST

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

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya
TRUST

Indirect Communication

- A mailbox can be viewed abstractly as an
    - object into which messages can be placed by processes and
    - from which messages can be removed.
    - Each mailbox has a unique identification

Fig: Indirect Addressing

K J Somaiya College of Engineering

Indirect Communication-mailboxes

- A process can communicate with some other process **via a number of different mailboxes.**

- Two processes can communicate only if the processes have a shared mailbox,
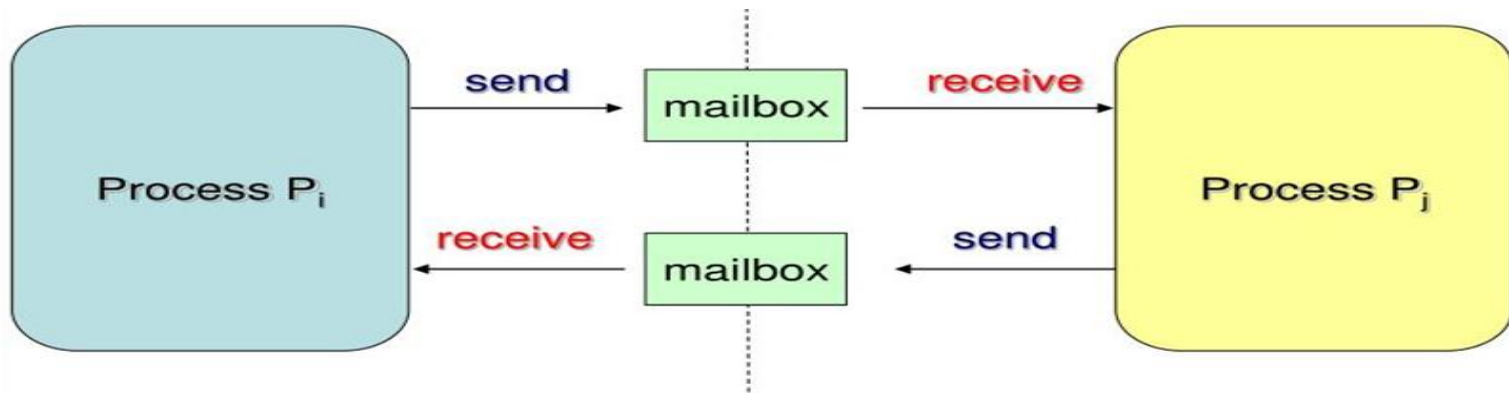
naming (indirect)
- send(m_a, message): m_a identifies mailbox a in the system
- receive(m_b, message): m_b identifies mailbox b in the system

K J Somaiya College of Engineering

Indirect Communication

- Properties of communication link
    - **Link established only if processes share a common mailbox**
    - A link may be associated with many processes
    - **Between Each pair of communicating processes, there may be a number of different links with each link corresponding to one mailbox**
    - Link may be unidirectional or bi-directional

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Indirect Communication

- Primitives are defined as:

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

  receive(*A, message*) – receive a message from mailbox A

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya
TRUST

Synchronization

- IPC takes place via send() and receive () primitives.

- There are different design options for implementing each primitive.

- Message passing may be either
  - Blocking/synchronous or
  - non blocking/ asynchronous

- A mailbox may be owned either
  - by a process or
  - by the operating system.

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

If the mailbox is owned by a process

- Mailbox is part of the address space of the process

- **The owner can only receive messages through this mailbox**

- **The user can only send messages to the mailbox**

- Since each mailbox has a unique owner,
  - there can be no confusion about which process should receive a message sent to this mailbox.

K J Somaiya College of Engineering

- What happens when the process that owns mailbox terminates??

What happens when the process that owns mailbox terminates??

- **When a process that owns a mailbox terminates,**
  - **the mailbox disappears**

- Any process that subsequently sends a message to this mailbox
  - must be notified that the mailbox no longer exists.

K J Somaiya College of Engineering

If the mailbox is owned by OS

- A mailbox that is owned by the operating system has an existence of its own.

- It is independent and is not attached to any particular process.

K J Somaiya College of Engineering

If the mailbox is owned by OS

- If OS owns the mailbox, then:

  o It is independent process, not attached to a process.
  o Then the OS must allow the process to:

  1) create a mailbox.
  2) send and receive messages through mailbox.
  3) delete the mailbox.

K J Somaiya College of Engineering

o Process becomes the owner of new mailbox by default.

o Owner process can only receive messages through this mailbox.

o Ownership and receiving privileges can be passed using system calls to other processes. This will result in multiple receivers for each mailbox.

K J Somaiya College of Engineering

Synchronization

- IPC takes place via send() and receive () primitives.

- There are different design options for implementing each primitive.

- Message passing may be either
  - Blocking/synchronous or
  - non blocking/ asynchronous

Synchronization

- Blocking send
  - The sending process is blocked until the message is received by the receiving process or by the mailbox.

- Non blocking send
  - The sending process sends the message and resumes operation.

- Blocking receive
  - The receiver blocks until a message is available.

- Non blocking receive
  - The receiver retrieves either a valid message or a null.

Synchronization

- When both send() and receive() are blocking,
  o Rendezvous between the sender and the receiver.

- The solution to the producer-consumer problem becomes trivial –
  o use blocking send() and receive()

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Solution =Producer-consumer problem

- Producer invokes
  - the blocking send() call and
  - waits until the message is delivered to either the receiver or the mailbox.

- Consumer invokes
  - receive(), it blocks until a message is available.

Buffering

- Whether communication is direct or indirect,
  - messages exchanged by communicating processes
  - reside in a temporary queue.

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya
TRUST

Buffering

- Queues can be implemented in three ways:
  - Zero capacity
  - Bounded capacity
  - Unbounded capacity.

Buffering

- Zero capacity
  - Maximum Queue Length=0
  - The link cannot have any messages waiting in it.
  - The sender must block until the recipient receives the message.

Buffering

- Bounded capacity
  - Queue has finite length *n;*
  - At most *n* messages can reside in it.
  - The link's capacity is finite.

05-12-2022

K J Somaiya College of Engineering

Buffering

- Bounded capacity
  - If the queue is not full
    - the message is placed in the queue
    - either the message is copied or a pointer to the message is kept
    - the sender can continue execution without waiting.
  - If the link is full,
    - the sender must block until space is available in the queue.

Buffering

- Unbounded capacity.
    - The queue's length is potentially infinite;
    - Any number of messages can wait in it.
    - The sender never blocks.

K J Somaiya College of Engineering

Buffering

- The zero-capacity = message system with no buffering;

- The other cases =systems with automatic buffering.

SOMAIYA

K J Somaiya College of Engineering

solution mechanisms to achieve concurrency

- **Software approach:**
  - o Here no hardware, OS, or programming language level supported is assumed.
  - o Burdens programmers to write the programs (which when executed becomes processes) to achieve concurrency themselves.
  - o Dekker's or Peterson's algorithms are known for the software approach solutions to mutual exclusion.

- **Hardware approach** :
  - o Use of special hardware instructions and control over the hardware such as disabling the interrupts.

- **Support from operating systems and programming languages**:
  - o Here, the operating system offers support to some signals
  - o the programmers can be provided with language constructs or library with which they can easily write the code for the process synchronize with each other.

Requirements for mutual exclusion

- Mutual exclusion must be enforced: Only one process at a time is allowed into its critical section

- A process that halts in its noncritical section must do so without interfering with other processes.

- The solution must ensure no deadlock or starvation.

- When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.

- No assumptions are made about relative process speeds or number of processors.

- A process remains inside its critical section for a finite time only

- Mutual exclusion techniques can be used to resolve conflicts, such as competition for resources, and to synchronize processes so that they can cooperate

Illustration of Mutual Exclusion

/* PROCESS 1 */	/* PROCESS 2 */		/* PROCESS n */

void P1
{
while (true) {
/* preceding code /;
EnterCriticalSection (Ra);
/* Execute Critical Section */;
ExitCriticalSection (Ra);
/* remaining code */;
}
}
	void P2
	{
	while (true) {
	/* preceding code */;
	EnterCriticalSection (Ra);
	/* Execute Critical Section */;
	ExitCriticalSection (Ra);
	/* remaining code */;
	}
	}
		...	void Pn
			{
			while (true) {
			/* preceding code */;
			EnterCriticalSection (Ra);
			/* Execute Critical Section */;
			ExitCriticalSection (Ra);
			/* remaining code */;
			}
			}

Hardware approaches to enforce mutual exclusion

- **Disabling interrupt** and

- **Use of special instructions**.

K J Somaiya College of Engineering

Disabling interrupt

- In a uniprocessor system, concurrent processes are not overlapped but they can only be interleaved.
- Furthermore, a process executes until it calls an OS service or until it is interrupted.
- Thus, it is sufficient to prevent a process from being interrupted to guarantee mutual exclusion, This capability can be provided in the form of primitives defined by the OS kernel for disabling and enabling interrupts.
-

The pseudo code:
while (true) {
/* disable interrupts */;
/* critical section */;
/* enable interrupts */;
/* remainder */;
}
-

- As the processes are not interrupted, the mutual exclusion is guaranteed.
- seems simple, the price paid for the same is very high.
- As the processor is not allowed to interleave the processes, the efficiency and overall system performance degrades significantly.
- Also, this particular solution will not work with multiprocessor architecture.
- When the computer includes more than one processor, it is possible (and typical) for more than one process to be executing at a time. In this case, disabled interrupts do not guarantee mutual exclusion.

Special Machine Instructions:

- In a multiprocessor systems, several processors share main memory.

- Moreover, the interrupt disabling cannot help the mutual exclusion.

- So to give mutually exclusive access to any memory location the processor designers have proposed several machine instructions that carry out some actions atomically, with one instruction fetch cycle.

- Examples: compare&Swap, test&set, Exchange, etc

Compare & Swap Instruction

int compare_and_swap (**int** *word, **int** testvalue, **int** newvalue)

{

int oldvalue;

oldvalue = *word;

if (oldvalue == testvalue) *word = newvalue;

return oldvalue;

}

K J Somaiya College of Engineering

Advantages and disadvantages of hardware approaches to enforce mutual exclusion.

- **Advantages**:
  - The solution works with any number of processes on uniprocessor as well as multiprocessors architectures.
  - It is very simple and therefore easy to verify.
  - It can also support multiple critical sections; and each critical section can be defined by its own variable.

- **Disadvantages:**
  - **Busy waiting is employed.**
  - **Starvation is possible.**
  - **Deadlock is possible.**
  - Consider the following scenario on a single-processor system. Process P1 executes the special instruction (e.g., compare&swap, exchange) and enters its critical section. P1 is then interrupted to give the processor to P2, which has higher priority. If P2 now attempts to use the same resource as P1, it will be denied access because of the mutual exclusion mechanism. Thus it will go into a busy waiting loop. However, P1 will never be dispatched because it is of lower priority than another ready process, P2. Because of the drawbacks of both the software and hardware solutions just outlined, we need to look for other mechanisms

Support from Operating systems and Programming Languages

- Signals

- Semaphores

- monitors

K J Somaiya College of Engineering

Semaphore

- **Semaphore:** Semaphore is an integer variable that can be initialized to some value.

- The semaphore values are used for signaling the processes for communication.

- The semaphores can be operated upon by three operations viz, initialize, increment and decrement.

- All these operations are atomic

- Depending on exact definition of semaphore, the decrement (wait) operation may block a process and the increment (signal) operation may unblock a process.

- Semaphores are supported by operating system to achieve concurrency and synchronization.

Mutex

- Short for **mu**tual **ex**clusion object.

- A mutex is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously.

- A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section.

- When a program is started, a mutex is created with a unique name.

- After this stage, any thread that needs the resource must lock the mutex from other threads while it is using the resource.

- The mutex is set to unlock when the data is no longer needed or the routine is finished.

Monitor

- Monitor is a programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type.

- The processes have only one entry and exit points to the monitor.

- Processes can access the same by invoking one of the procedures contained in monitor, which in turn can access the data and variables local to the monitor.

- To achieve mutual exclusion, only one process can be active inside a monitor.

- Generally, the access procedures are *critical sections*. A monitor may have a queue of processes that are waiting to access it.

Monitor

- A monitor supports synchronization by the use of **condition variables**

- **condition variables** are contained within the monitor and accessible only within the monitor.

- Condition variables are a special data type in monitors, which are operated on by two functions:

- cwait(c): Suspend execution of the calling process on condition c.

- csignal(c): Resume execution of some process blocked after a cwait on the same condition. If there are several such processes, choose one of them; if there is no such process, do nothing.

Semaphores : **Types of semaphore**

- **Binary Semaphore**:
  - o Binary Semaphore is a semaphore that takes on only the values 0 and 1.

- **General or counting Semaphore**:
  - o Any Semaphore that is not restricted to have value only 0 and 1 and can have values >=2 are called general or counting semaphores.

- In other words, the nonbinary semaphore is often referred to as either a counting semaphore or a general semaphore.

- **Strong Semaphore**:
  - o A semaphore whose definition includes FCFS policy when it is unblocked from the blocked queue, is called a Strong Semaphore.

- **Weak Semaphore**:
  - o A semaphore whose definition does not include any policy when it is unblocked from the blocked queue, is called a Weak Semaphore.

semaphore primitives

- Semaphore is an integer variable that can be initialized to some value.

- The semaphore values are used for signaling the processes for communication.

- They can be operated upon by three operations viz,
  - initialize,
  - increment and
  - decrement.

- All these operations are atomic in nature.

- Depending on exact definition of semaphore, the decrement (wait) operation may block a process and the increment (signal) operation may unblock a process.

Definition of Semaphore Primitives

```
struct semaphore {
int count;
queueType queue;
};


void semWait(semaphore s)
{
s.count--;
if (s.count < 0) {
/* place this process in s.queue */;
/* block this process */;
}
}


void semSignal(semaphore s)
{
s.count++;
if (s.count <= 0) {
/* remove a process P from s.queue */;
/* place process P on ready list */;
}
}
```

```
/* program mutualexclusion */

const int n = /* number of processes */;

semaphore s = 1;
void P(int i)
{
while (true) {
semWait(s);
/* critical section */;
semSignal(s);
/* remainder */;
}
}

void main()
{
parbegin (P(1), P(2), . . ., P(n));
}
```

At particular time, the value of a counting semaphore is 10, it will become 7 after:
(a) 3 V operations
(b) 3 P operations
(c) 5 V operations and 2 P operations
(d) 2 V operations and 5 P operations

Which of the following option is correct?
(A) Only (b)
(B) Only(d)
(C) Both (b) and (d)
(D) None of these

wait () operation = originally termed P

from the Dutch *proberen, Meaning* "to test" or "to attempt"

signal() operation = originally called V

from *verhogen, Meaning* "to increment

At particular time, the value of a counting semaphore is 10, it will become 7 after:
(a) 3 V operations
(b) 3 P operations
(c) 5 V operations and 2 P operations
(d) 2 V operations and 5 P operations
Which of the following option is correct?
(A) Only (b)
(B) Only(d)
(C) Both (b) and (d)
(D) None of these

Answer: (C)
Explanation: P: Wait operation decrements the value of the counting semaphore by 1.
V: Signal operation increments the value of counting semaphore by 1.
Current value of the counting semaphore = 10
a) after 3 P operations, value of semaphore = 10-3 = 7
d) after 2 v operations, and 5 operations value of semaphore = 10 + 2 – 5 = 7
Hence option (C) is correct.

SOMAIYA
VIDYAVIHAR UNIVERSITY

At a particular time of computation, the value of a counting semaphore is 10. Then 12 P operations and "x" V operations were performed on this semaphore. If the final value of semaphore is 7, x will be:
(A) 8
(B) 9
(C) 10
(D) 11

Answer: (B)
Explanation: Intially the value of a counting semaphore is 10 Now 12 P operation are performed.
Now counting semaphore value = -2
"x" V operations were performed on this semaphore and final value of counting semaphore = 7
i.e $x + (-2) = 7$
$x = 9$
So, option (C) is correct.

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
  - The event in question is execution of signal operation

- Let s and q be two semaphores initialized to 1

$$P_0 \qquad\qquad\qquad P_1$$

```
        wait(S);                        wait(Q);

    wait(Q);                        wait(S);
  ...                      ...
    signal(S);                      signal(Q);
    signal(Q);                      signal(S);
```

Deadlock and Starvation

P_0 P_1

```
wait(S);                    wait(Q);

wait(Q);                    wait(S);

...              ...

signal(S);                          signal(Q);

signal(Q);                          signal(S);
```
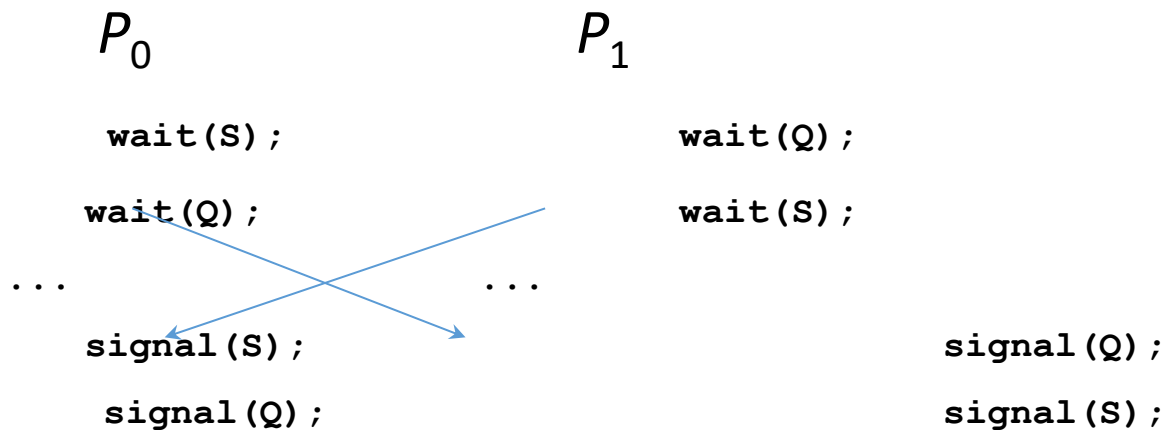
- **P0 executes wait(S) then P1 executes wait(Q)**

- **When P0 executes wait(Q), it must wait until P1 executes signal(Q)**

- **When P1 executes wait(S), it must wait until P0 executes signal (S)**

Deadlock and Starvation

- Deadlock-Every process in the set is waiting for an event that can be caused only by another process in the set

- Starvation – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended

- Priority Inversion – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via priority- inheritance protocol

SOMAIYA
VIDYAVIHAR UNIVERSITY

K J Somaiya College of Engineering

Dining-Philosophers Problem
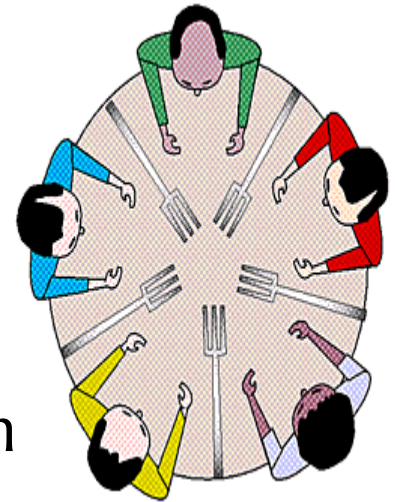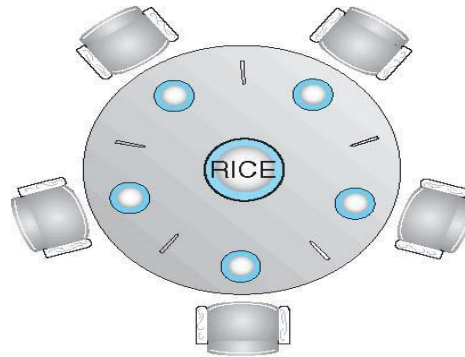
- Philosophers spend their lives alternating thin and eating

- When thinking
  - Don't interact with their neighbors

- When hungry-
  - A philosopher needs both their right and left chopstick to eat.
  - A hungry philosopher may only eat if there are both chopsticks available

Dining-Philosophers Problem
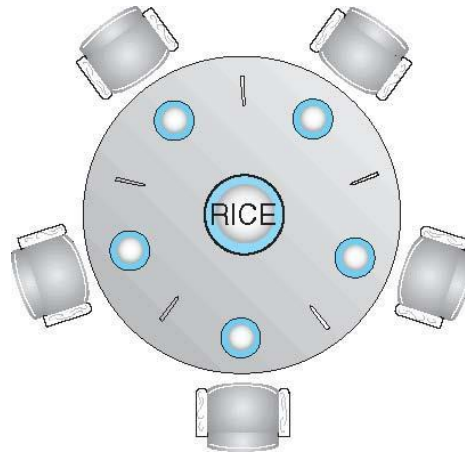
- Occasionally try to pick up 2 chopsticks that are closest to her to eat from bowl
  - Chopsticks that are between her and her left and right neighbor
  - Pick up only one at a time
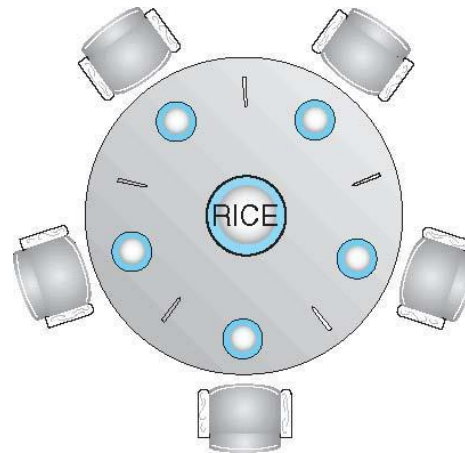 - Need both to eat, then release both when done
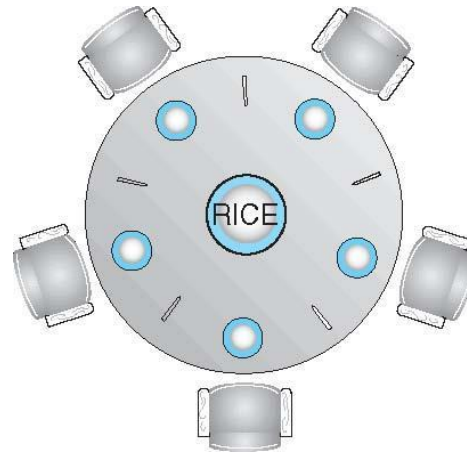
Dining-Philosophers Problem
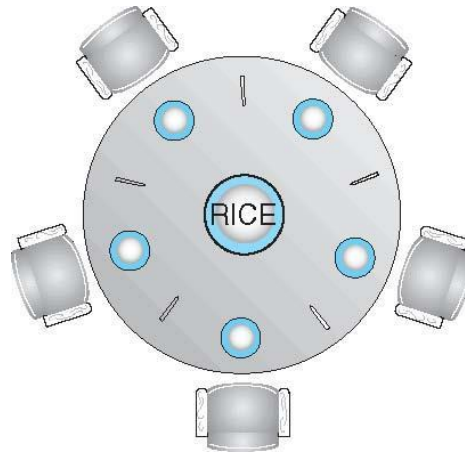
- Classic Synchronization problem

- Example of large class of concurrency control problems

- Represents the need to allocate several resources among several processes
  - in a deadlock free and starvation free manner

Dining-Philosophers Problem
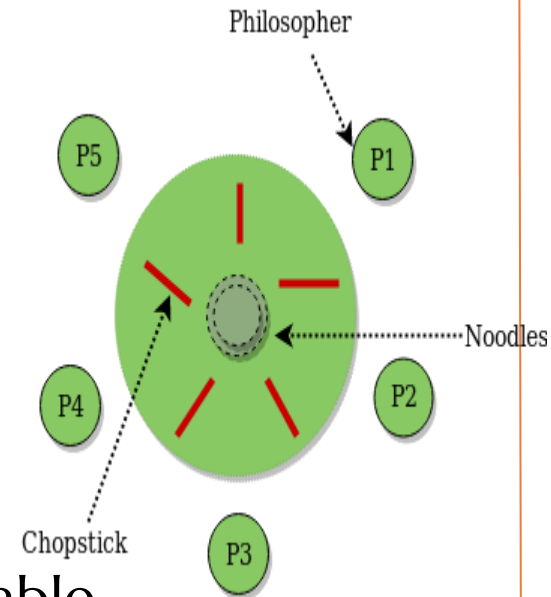
- 5 Philosophers share a common circular table
  - Surrounded by 5 chairs=each belonging to one philosopher
  - Center of table ->Bowl of rice
  - Five single chopsticks
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

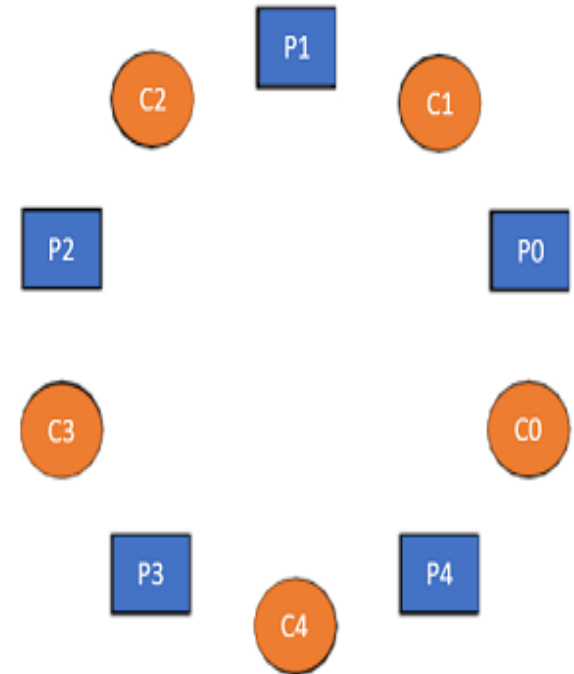# Semaphore Solution for Dining-Philosophers Problem

- Grab the chopstick –By Executing wait operation on the semaphore

- Release the chopstick-By executing the signal operation on the appropriate semaphore

- Semaphore chopstick [5] initialized to 1

Semaphore Solution for Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
     wait (chopStick[ (i + 1) % 5] );

                    //  eat

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

                    //  think

} while (TRUE);
```

Semaphore Solution for Dining-Philosophers Problem Algorithm

What is the problem with this algorithm?

Semaphore Solution for Dining-Philosophers Problem Algorithm

What is the problem with this algorithm?

- Algorithm Guarantees that No two neighbors are eating simultaneously

Still Must be rejected

- Why?

Semaphore Solution for Dining-Philosophers Problem Algorithm

Why?

- Possibility of deadlock

- If all 5 philosophers are hungry simultaneously and each grabs left chopstick

- All elements of chopsticks will be =0

- When each philosopher tries to grab her right chopstick, delayed forever
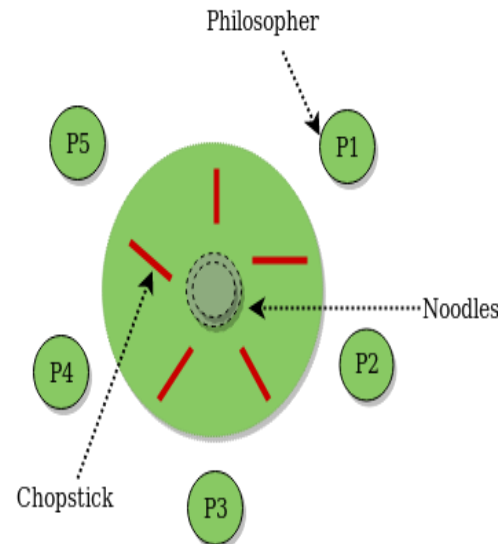
Semaphore Solution for Dining-Philosophers Problem Algorithm

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section)
  - Use an asymmetric solution --
    - Odd-numbered philosopher picks up first the left chopstick and then the right chopstick.
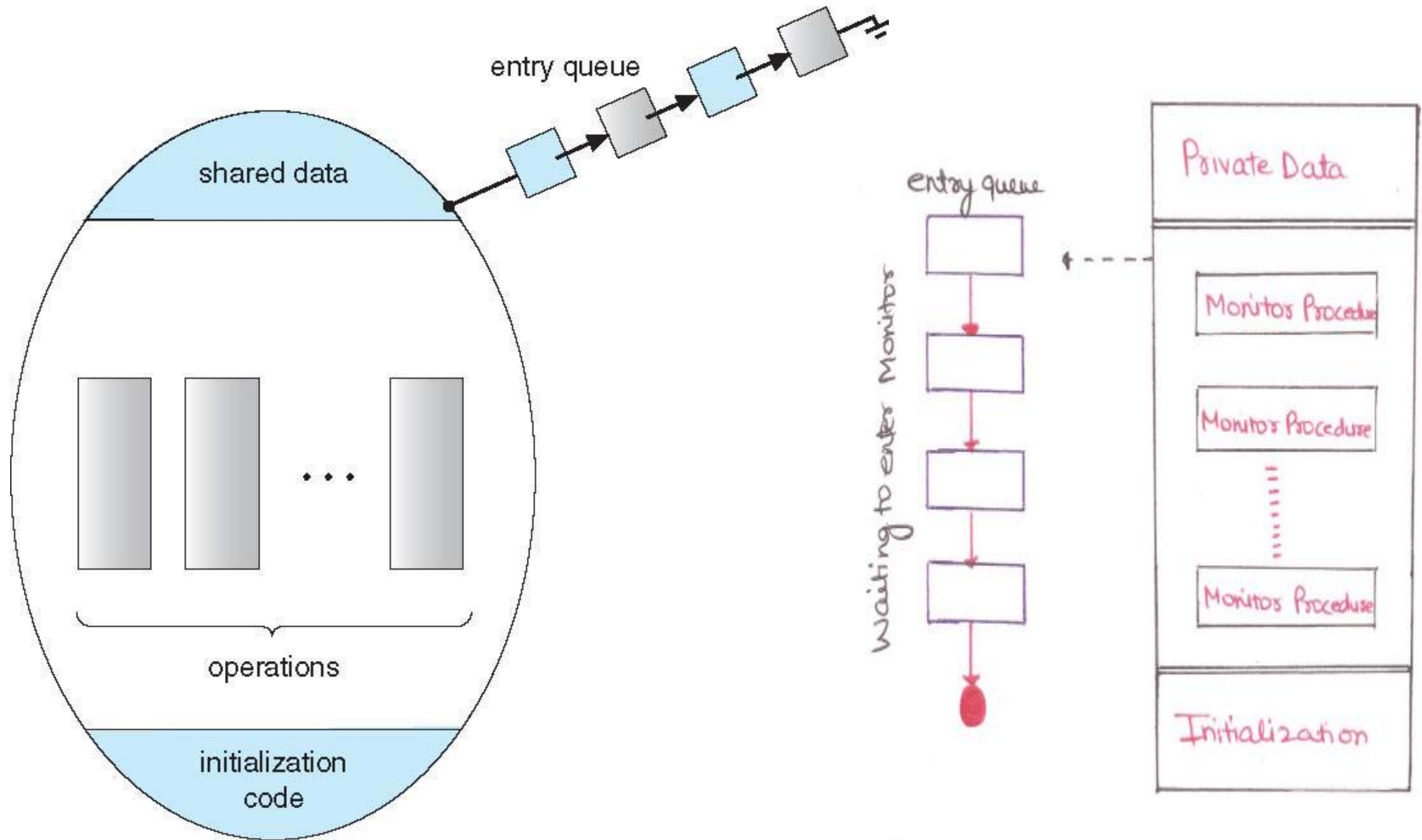   - Even-numbered philosopher picks up first the right chopstick and then the left chopstick

Monitors

Monitors

- A high-level synchronization construct

- *Set of programmer defined operators*

- *Declaration of variables*
  - *whose value define the state of an instance of the type*

- *Bodies of procedures*
  - *that implement operations on the type*

Syntax of Monitors

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }


  procedure Pn (…) {……}


     Initialization code (…) { … }
  }
}
```

Monitors

- *Abstract data type,* internal variables only accessible by code within the procedure

- Procedure defined within the monitor can access only those variables
  - declared locally within the monitor and
  - its formal parameters.

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }


  procedure Pn (…) {……}


      Initialization code (…) { … }
}
}
```

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Monitors

- It is the collection of condition variables and procedures combined together in a special kind of module or a package.

- The processes running outside the monitor can't access the internal variable of the monitor but can call procedures of the monitor.

- Only one process at a time can execute code inside monitors.

Monitors

- Monitor ensures that
  - Only one process may be active within the monitor at a time

- Programmer does not need to
  - code the synchronization constraint explicitly

- But not powerful enough to model some synchronization schemes

- Need to define additional condition construct

Condition Variables

- **Prgmr can define one or more condition variables**

$$\text{condition } x, y;$$

- Only Two operations are allowed on a condition varia~~ble~~
  - **x.wait()**
  - **x.signal()**

Monitor Demo //Name of Monitor
{
variables;
condition variables;

procedure p1 {....}
prodecure p2 {....}

}

Syntax of Monitor
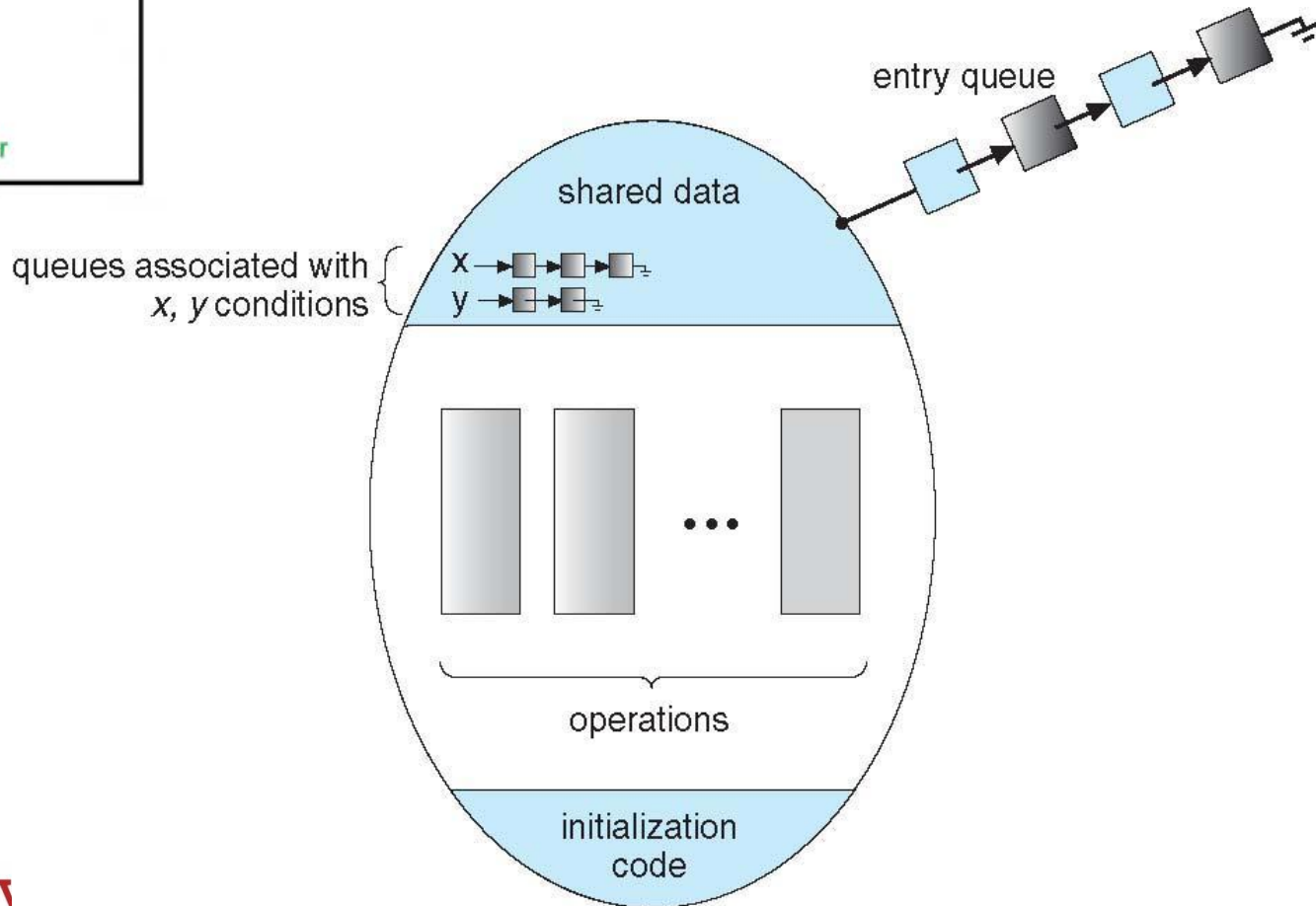
Condition Variables

- **`x.wait()`** –
  - Process performing wait operation on any condition variable are suspended.
    - suspended until **`x.signal()`**
  - The suspended processes are placed in <u>block queue of that condition variable.</u>

- **`x.signal()`** –
  - When a process performs signal operation on condition variable, one of the blocked processes is given chance.
  - resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`process is suspended`**, then it has no effect on the variable
    - State of x, As if the operation was never executed
    - In contrast to semaphore, state of semaphore always gets affected

```
Monitor Demo //Name of Monitor
{
variables;
condition variables;

procedure p1 {....}
prodecure p2 {....}


}
        Syntax of Monitor
```


entry queue

shared data

queues associated with
x, y conditions

x →
y →

operations

initialization
code

Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`,
  - Suspended process Q associated with condition x is invoked
  - what should happen next?

- Both Q and P cannot execute in parallel.
  - If Q is resumed, then P must wait
  - Otherwise both P and Q will be active simultaneously within the monitor

Condition Variables Choices

- Options include
    - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
    - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition

Condition Variables Choices

- Since P was already executing in the monitor,
  - Option 2 seems more reasonable
  - However, If P continues, the logical condition for which Q was waiting may no longer hold by the time Q is resumed

- Both have pros and cons – language implementer can decide

- Monitors implemented in Concurrent Pascal compromise
  - P executing signal immediately leaves the monitor, Q is resumed

- Implemented in other languages including Mesa, C#, Java

Monitors

- **Advantages of Monitor:**

  - Make parallel programming easier and
  - less error prone than using techniques such as semaphore.

- **Disadvantages of Monitor:**

  - Monitors have to be implemented as part of the programming language .
  - The compiler must generate code for them.
  - This gives the compiler the additional burden of having to know what operating system facilities are available to control access to critical sections in concurrent processes.
  - Some languages that do support monitors are Java,C#,Visual Basic,Ada and concurrent Euclid.

Monitor Solution to Dining Philosophers

- Deadlock free solution to Dining Philosophers Problem

- To distinguish amongst the 3 states in which the philosopher may be-
  - **enum { THINKING; HUNGRY, EATING) state [5] ;**

- Philosopher i can set the variable state [i] = EATING only
  - **if her two neighbors are not eating**
  - **(state [(i +4) % 5] ! = EATING) and (state [(i +1) % 5] != EATING)**

Monitor Solution to Dining Philosophers

- Also need to declare
  - condition self[5];

  - in which ith philosopher **can delay herself when she is hungry but is unable to obtain the chopsticks she needs.**

- Distribution of chopsticks is controlled by the monitor DiningPhilosophers

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
  enum { THINKING; HUNGRY, EATING) state [5] ;
  condition self [5];

  void pickup (int                                    Pickup chopsticks
        state[i] = HUNGRY;
        test(i);
        if (state[i] != 
              self[i].wait;
}
```

- If unable to eat, wait to be signaled
- Philosopher can delay herself when she is hungry but is unable to obtain the chopsticks she needs.

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{

    void putdown (int i)                              Put down chopsticks
        state[i] = THINKING;
                    // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
}
```

if right neighbor R=(i+1)%5 is hungry and
  both of R's neighbors are not eating,
  set R's state to eating and wake up neighbour
R by signaling

```
void test (int i) {
        if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
                state[i] = EATING ;
            self[i].signal () ;
            }
    }


    initialization_code() {
        for (int i = 0; i < 5; i++)
        state[i] = THINKING;
        }
```

- If her two neighbors are not eating and she is hungry
  - I.E. if my left and right neighbors are not eating
    - Set her state as eating

- signal() has no effect during Pickup()
- but is important to wake up waiting hungry philosophers during Putdown()

Solution to Dining Philosophers

- Each philosopher *i* invokes the operations `pickup()` and `putdown()` in the following sequence:

```
DiningPhilosophers.pickup(i);
```

EAT

```
DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Solution to Dining Philosophers

- Each philosopher, before starting to eat, must invoke the operation pickup().

- This act may result in the suspension of the philosopher process.

- After the successful completion of the operation, the philosopher may eat.

- After eating, philosopher invokes putdown() and start to think

Solution to Dining Philosophers

- Execution of Pickup(), Putdown() and test() are all mutually exclusive, i.e. only one at a time can be executing

- No 2 neighbors are eating simultaneously

- So No deadlock, but starvation is possible

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Question
?