M T W T F S S
Page No.:
Date:
YOUVA

Module-3

* IPC (Inter-process Communication)

1. Independent Process:

- Process cannot affect or cannot be affected by other processes executing in the system.
- Any process that does not share any data with any other process.

2. Cooperating Process:

- Can affect or be affected by other processes.
- Any process that shares data with other process.
- Reasons:

(i) Several users might be interested in the same piece of information. We must provide an environment to allow concurrent access to such information. (Information Sharing)

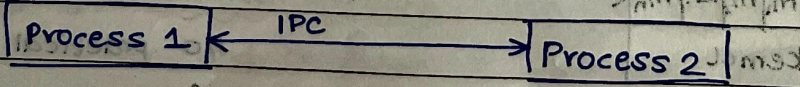
(ii) If we want a task to run faster, we must break it into sub tasks and each subtask will execute in parallel. (Computation Speedup)

(iii) Speedup can be achieved if computer has multiple processing elements like CPUs or I/O channels.

(iv) We must divide the system functions into processes or threads. (Modularity)

(v) Individuals can also work on many tasks at the same time (Convenience)

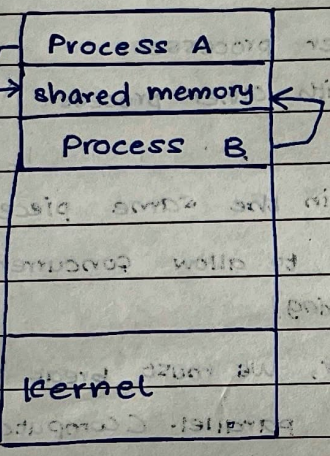
- Cooperating processes need IPC to exchange data and information.



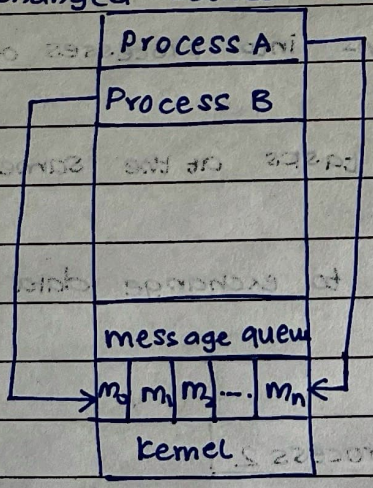
* IPC

- Two models of IPC:
  - Shared Memory
  - Message Passing

A region of memory is shared by cooperating process is established. Processes can read and write data into the shared region.



Communication takes place by means of messages that are exchanged between cooperation processes.



* Bounded Buffer assumes that there is fixed buffer size.

* Unbounded buffer no practical limit on the size of buffer.

Message Passing	Shared Memory
- Message passing is useful when smaller amounts of data are exchanged because no conflicts can be avoided.	- Shared memory allows maximum speed and convenience of communication
- Slower as message passing systems are typically implemented using system calls and thus the more time-consuming task of kernel intervention.	- Faster as in shared memory systems, system calls are required only to establish only shared regions. Once the shared regions are established, all accesses are treated as memory accesses & no assistance from kernel is needed.

→ Shared Memory

* Producer - Consumer Problem (Bounded Buffer Problem):

- Paradigm for cooperating processes.
- Producer process produces information that is consumed by consumer process.

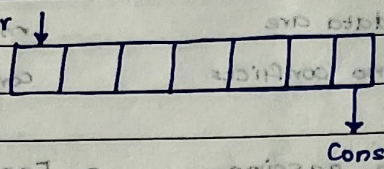
eg: Compiler produces assembly code which is consumed by an assembler, the assembler produces object modules which are consumed by loader.

eg: Server (Web server) produces HTML files and images which are consumed by Web Browser requesting the resource.

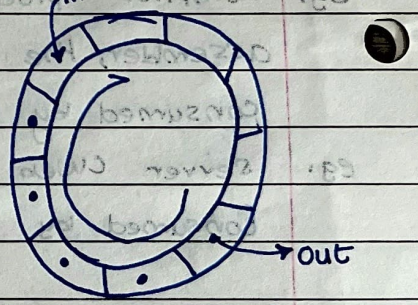
* ~~Solution 1~~: Shared Memory Problem Statement:

- Allow Producer and consumer to run concurrently.
- Available buffer of items that can be filled by producer and emptied by consumer.
- Buffer resides in memory that will be shared by producer & consumer processes.

- Producer is producing one item, while consumer consumes item



- Producer and consumer processes must be synchronised so that the producer does not produce when buffer is full and consumer doesn't consume when buffer is empty.
- If an unbounded buffer was used here then, consumer would have to wait for the new items to be produced and producer can always produce new items.
- If a bounded buffer was used, then consumer must wait if buffer is empty and producer must wait if buffer is full.
- Algo:
  - Following variables reside in a region of memory shared by producer and consumer process:
  - Shared buffer is implemented as a circular array with two pointers:
    in - points to next free position in buffer
    out - points to first full position in buffer




```
#define BUFFER_SIZE 10
typedef struct {
} item;
item buffer[BUFFER_SIZE];
int in;
int out;
```

- Buffer is empty if in == out
- Buffer is full when ((in+1) % BUFFER_SIZE) == out
- Producer process has a new variable nextProduced in which the new item to be produced is stored.

Consumer consumes

Consumer Process has new local variable nextConsumed that stores the next item to be consumed.

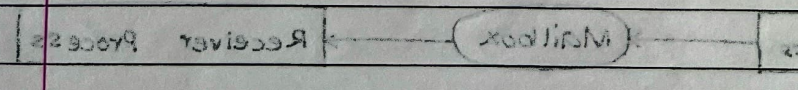
(Solution-2 done later)

* Message Passing

- Message Passing is another way of cooperating processes to communicate with each other.
- Mechanism for processes:
  - (i) to communicate
  - (ii) to synchronise their actions.
- Processes communicate with eachother without resorting to shared variables or shared address space.
- Used by a distributed system (environment where communicating processes reside on different computers. They are connected by a network.

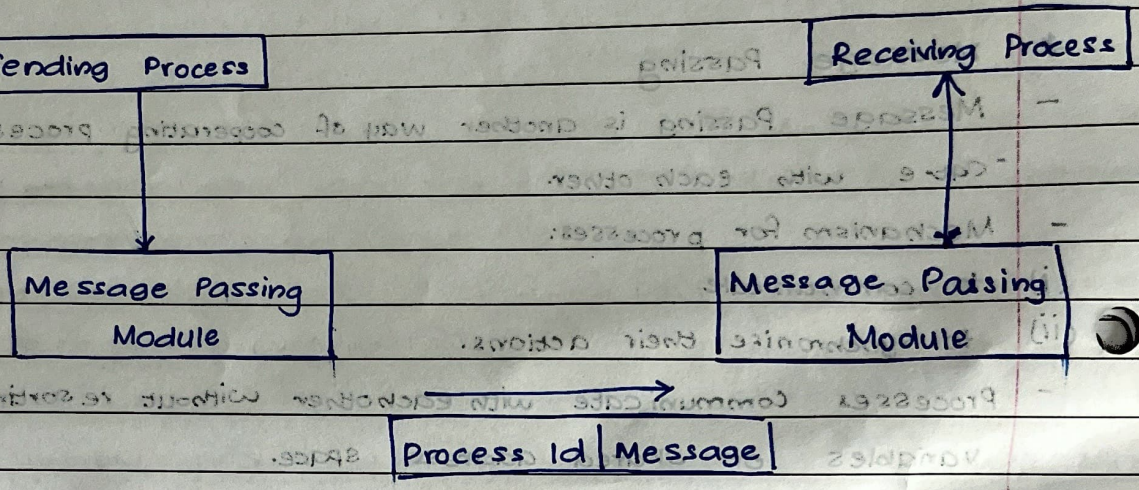
eg: Chat application

- IPC provides two operations: send(message), receive(message)
- Message size is fixed or variable.
- If fixed-size message is sent, implementation is straightforward and programming task is difficult.
- If variable-size message is sent, implementation is complex but programming task is easy.
- If two processes P and Q wish to communicate, a communication link must be made between them and messages can be exchanged using send() or receive() functions.
- Two way of communication between processes:
  - (i) Direct Communication (ii) Indirect Communication.



* Direct Communication:

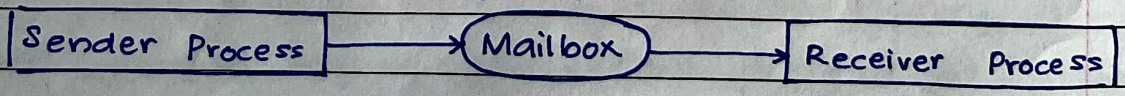
- Processes must name each other explicitly:
  - send (P, message): send a message to P.
  - receive (Q, message) receive a message from Q.



- Processes must know only each other's identity when they wish to communicate.
- A link is associated with only one pair of communicating process.
- Link can be bidirectional or unidirectional.
- Scheme exhibits symmetry, sender and receiver must name each other to communicate.
- Assymetry is employed when the sender names the receipient and receipient is not required to name the sender.

* Indirect Communication:

- Messages are sent and received from mailboxes (ports)
- Each mailbox has a unique id
- Processes communicate only if they share a mailbox.
- Mail box can be viewed as an object in which messages are placed by processes and also removed.



- One process can communicate with other processes via a number of different mailboxes.
- Link is established only if processes share a common mailbox
- Each link corresponds to only one mailbox
- Links can be bidirectional or unidirectional.
- send (a, message) - send a message to mailbox 'a'.
- receive (b, message) - send a message to mailbox 'b'.
- Mailbox is owned either by a process or operating system.
- If mailbox is owned by a process, then it becomes a part of the address space of the process.
- The owner can only receieve messages from the mailbox.
- The user can only send messages to the mailbox.
- Since each mailbox has a unique owner, there is no confusion about which process should receieve the message sent to the mailbox.
- If the process that owns the mailbox terminates then the mailbox also disappears and all the processes that send a message to this mailbox are also notified.
- If mailbox is owned by OS, then mailbox has an existence of its own: it is independent and not attached to any process.
- OS then allows the process to:

(i) create a new mailbox

(ii) send and ~~ree~~ receive messages through mailbox.

(iii) delete the mailbox.

- Process beicomes owner of mail box
- Ownership and receiving priveleges can be passed using system calls. This will cause multiple receivers for each mail box.

* Synchronization:

- IPC takes place via send() or receive() primitives.
- Message passing can be:

(i) Blocking send (Synchronous)

- Sending process is blocked until the receiving process or the mailbox receives the message.

(ii) Blocking receive.

- Receiver is blocked until message is available.

(iii) Non-blocking send (Asynchronous)

- Sender process sends message and continues with operation.

(iv) Non-blocking receive (Asynchronous)

- Receiver recieves a valid message or null.

→ Solution-1

→ Producer-Consumer Problem can be solved using synchronisation.

- Producer invokes blocking send() call and waits till message is delievered to receiver or mailbox.
- Consumer invokes blocking receive() it until a message is available.

* Buffering:

- Whether process communication is direct or indirect, messages are exchanged and they reside in a temporary queue.
- Queues are implemented in three ways:

① Zero capacity:

- Max. Queue length = 0
- Link cannot have any message waiting in it.
- The sender must block until the receipient arrives.

② Bounded Capacity:

- Queue has finite length 'n'.
- Almost 'n' messages can reside in it.
- Link's capacity is infinite.
- If queue is not full, message is placed in queue. Either the message is copied or a pointer to the message is kept.
- If queue is full, sender must block until space is available.

③ Unbounded Capacity:

- Queue length is potentially infinite.
- Any no. of messages can wait in it.
- Sender never blocks.
- zero capacity with no buffering

* Producer - Consumer Problem ctd.

* Solution-2

- Modifying the approach for Producer-Consumer Problem by having a counter that keeps track of total number of buffers (full).
- One possibility is to have an integer variable counter, intialized to 0. It is incremented every time we add new item to buffer and decremented when we remove one item from the buffer.
- Earlier Approach :

```
item nextProduced;
while (true) {
        while(((in+1) % BUFFER_SIZE) == out);
        /* do nothing */
        buffer[in] = nextProduced;
        in = (in+1) % BUFFER_SIZE;
        counter++;
```

PRODUCER

- Modified Approach :

PROCESS

```
while (true) {
        while (counter == BUFFER_SIZE);
        /* do nothing */
        buffer [in]= nextProduced;
        in= (in +1) % BUFFER_SIZE ;
        counter ++;
```

- Earlier Approach (Consumer Process)

```
item nextConsumed;
while (true) {
    while (in == out);
    /* do nothing */
    nextConsumed = buffer[out];
    out = (out+1) % BUFFER_SIZE;
```

- Modified Approach:

```
while (true) {
    while (counter == 0);
    /* do nothing */
    nextConsumed = buffer[out];
    out = (out+1) % BUFFER_SIZE;
    counter --;
```

- Producer and Consumer processes work correct separately but may not execute correctly when executed concurrently.
- When several processes access and manipulate data concurrently, outcome depends on order in which access takes place. It is also called Race condition.
- To guard against race condition, only one process must manipulate the variable counter. Thus process should be synchronised.

(Semaphore Solution done later)

* **Critical Section Problem:**

- Each process has a segment of code, called critical section in which the process may be : changing common variables, writing a table file, updating a table and so on.
- When one process is executing in its critical section, no other process is allowed to execute in its critical section. Execution of critical sections is mutually exclusive.
- **Entry Section:** Each code must request permission to enter its critical section. This section of implementing the request is called entry section.
- **Exit Section:** The critical section is followed by an exit section.
- **Remaining Section:** The remaining code is the remaining section.
- Critical Section Solution must satisfy 3 requirements:

(i) **Mutual Exclusion:** If a process P_i is executing in the critical section, no other process can be executing in their critical sections.

(ii) **Progress:** If no process is executing in critical section and some processes wish to enter the critical section, then only those processes can enter which are not executing in remainder section can participate in deciding which will enter the critical section next. This selection cannot be postponed indefinitely.
Only those process interested in entering CS should compete to enter the CS.

(iii) **Bounded Waiting:** There exists a limit / bound on the no. of times that the processes are allowed to enter Critical Sections.
Max after a bound / time limit, after which the process definitely will get a chance to enter critical section.

* Critical Section Handling in OS

- There are 2 approaches to handle critical sections in OS depending on if kernel is:

(i) Pre-emptive:

- Preemptive kernels allow a process to be pre-empted while it is running in kernel mode.
- Two kernel mode processes can run simultaneously on different processors.
- Must be carefully designed to ensure kernel data are free from race conditions.

(ii) Non-preemptive:

- Does not allow kernel in a process to be preempted.
- Free from race conditions on kernel data structures.
- Only one process is active in kernel, one at a time.
- Non-preemptive kernel approach will run the process until it:
  - exits kernel mode,
  - blocks or
  - voluntarily yields control of CPU.

- A preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel. It maybe more responsive, since there is less risk at that a kernel-mode process will run for an arbitarily long period before relinquishing the processor to waiting processes.

★ Solutions to Critical Section Problem

1. Software based solutions:

→ Two process Solutions:

- Assume that the load and store machine-language instructions are atomic - that is cannot be interrupted.

(A) Algorithm 1:

```
do {
    while (turn != i) ;        ← entry code
        critical section
    turn = j;                  ← exit code
        remainder section
} while (1);
```

- Structure of Process P_i in algorithm 1
- Entry code acts like a trap, it stops the processes from entering the critical section.
- Turn = Shared, Common / Global Integer initialised to 0 or 1.
- Checking for Mutual exclusion:
- Initialize turn with 0.

P_0, i=0

```
do {
    while (turn != 0) ;
        critical section
    turn = 1;
        remainder section
} while (1);
```

P_1, i=1

```
do {
    while (turn != 1) ;
        Critical section
    turn = 0;
        remainder section
} while (1);
```

- Can P_1 enter CS if P_0 is in CS?
- Lets take turn = 0 for P_0.
- While condition = false, P_0 enters critical section.
- While P_0 is still in CS, turn = 0
- P_1 tries to enter CS. Turn is still 0.

- while condition = true, P_1 gets trapped in an infinite loop.
- P_1 is unable to enter CS.

- Can P1 enter CS while P_0 is in remainder section?
  - For P_0, after critical section is executed, the exit code makes turn = 1
  - It starts executing remainder section.
  - P1 tries to enter CS.
  - while condition = false, P1 enters CS.

- Can P_0 enter CS again after completing RS?
  - After first run, turn = 1
  - For P_0, while condition = true. P_0 gets trapped in an infinite loop.
  - P_0 cannot enter CS.

MUTUAL EXCLUSION SATISFIED!

- if turn == i, P_i executes in CS
- Progress Requirement Check
- After P_0 comes out, P_1 gets to go inside and vice-versa.
- If a process does not want to go in CS, it still has to go due to alteration.
- If P_1 does not want to go and P_0 wants to go in CS, it still cannot go.

PROGRESS REQUIREMENT NOT SATISFIED!

- Bounded Waiting Check:
  - Process 0 can go directly and Process 1 can go immediately after P_0 is done with the critical section.

BOUNDED WAITING SATISFIED!

- Mutual Exclusion is Preserved: Ensures that only one process at a time can be in its critical condition.
- Progress Requirement not satisfied
- Bounded waiting satisfied.

Page No.:
Date:
YOUVA

(B) **Algorithm 2:**

- Two Processes share a Boolean array.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i]= true implies process P_i is ready.
- Boolean array is initialised to be false.
- As it wants to enter CS, it is changed to True.

```
do {
    flag[i]= true;
    while (flag[j]);
        critical section
    flag [i]= false;
        remainder section
} while(1);
```

Structure of process in Algorithm 2

- P_0

```
do {
    flag [0] = true ;
    while (flag [1]);
        critical section
    flag [1]= false;
        remainder section
} while(1);
```

P_1

```
do{
    flag [1]= true;
    while (flag [0]);
        critical section
    flag [1]= false;
        remainder section
}while(1);
```

Boolean Array

[0]	[1]
F	F

- Checking for Mutual Exclusion:
  - If P_0 is executing in CS, can P_1 enter CS?
  - If P_0 wants to enter CS, set flag[0] = T.
  - P_0 checks if P_1 wants to go too.
  - As flag[1] = F, while condition is false, P_0 enters critical section and executes it.

[0]	[1]
T	F

[0]	[1]
T	T

- P1 tries to enter CS.
- Set flag [1]= T.
- While condition = true, P_1 goes in an infinite loop and does not enter CS.

ctd.

- Can P_1 enter CS if P_0 is in Remainder Section?
  - For P_0, after executing CS, flag[0] = false and it enters RS.
  - P_1 tries to enter CS. Set flag[1] = T
  - While condition is false, so P_1 enters CS.

[0]	[1]
F	T

MUTUAL EXCLUSION SATISFIED!

Progress Requirement Check:

- Can P_0 enter CS again after completing RS?
  - If P_1 is not interested in CS and flag[1] = F and P_0 wants to go again, we set flag[0] = T.
  - while condition = false, P_0 enters CS

[0]	[1]
T	F
F	F

Progress Requirement Check:

- If P_1 wants to go in CS again and immediately after RS, it can enter if P_2 is not interested.
- Whichever process interested and other processes are not, can enter the CS.
- If both P_0 and P_1 get trapped in while loop, both cannot enter CS.

[0]	[1]
T	T

Both go in an infinite loop and thus there is no progress here.

- Mutual Exclusion preserved.
- Progress requirement not satisfied.

(c) **Algorithm 3 / Peterson's Solution**

Two processes share two variables:

int turn; and Boolean flag[2]

Variable turn indicates whose turn it is to enter CS and flag array indicates if a process is ready to enter CS.

```
do {
    flag[i] = True;
    turn = j;
    while(flag[j] && turn == j);
        Critical Section
    flag[i] = False;
        remainder Section
} while(true);
```

Structure of Peterson's solution:

P_0

```
do {
    flag[0] = true;
    turn = 1;
    While(flag[1] == T && turn == 1);
        Critical Section
    flag[0] = false;
} while(true);
```

P_1

```
do {
    flag[1] = True;
    turn = 0;
    While(flag[0] == T && turn == 0);
        Critical Section
    flag[1] = False;
} while(1);
```

Boolean Array[2]

[0]	[1]
F	F

- Mutual Exclusion Check:

• If P_0 is in CS, can P_1 enter CS?

turn = 1

[0]	[1]
T	F

- For P_0, set flag as true and turn as 1.
- P_1 is not interested
- While (T && F) → While condition is false.
- P_0 enters the CS.
- P_0 is already in CS and P_1 tries to enter CS.
- Set flag[1] = T and turn = 0
- While (T && T) → while condition is true. P_1 gets trapped in infinite loop. So it cant enter CS.

turn = 0

[0]	[1]
T	T

• Can P_1 enter while P_0 is in remainder section?

- After P_0 is done with execution is in CS, flag is set as False
- P_1 tries to enter CS. flag[1] = T and turn = 0.
- while (T && F) → while condition is false.
- P_1 enters CS

[0]	[1]
F	T

turn = 0

MUTUAL EXCLUSIVE SATISFIED!

- Progress Requirement Check:

If both P_0 and P_1 want to enter CS

- For P_1, if P_1 is interested. Set flag as true and turn = 0
- Context switch occurs.
- As P_0 had set turn as 1
- Now, P_1 tries while(F && T) → while condition is false. P_1 enters CS.
- Now, P_0 is also interested. set flag as true and turn = 1
- while (T && T) → While condition is true. P_0 gets trapped.
- Now context switch. P_0 does not enter CS.

PROGRESS CHECK SATISFIED!

- **Bounded Waiting Check:**

P_i will be critical section after atmost one entry by P_j. Everyone gets a fair chance.

BOUNDED WAITING CHECK SATISFIED!

2. **Hardware based Solutions:**

- Software solutions like Peterson's may not work for modern computer architectures.
- Race conditions are prevented and critical regions can be protected by locks.
- A process must acquire a lock before entering CS; it is released when it exits the CS.

```
do {
    acquire lock
        Critical section
    release lock
        remainder section
} while (1);
```

(A) **Test Memory word and set value: test and set()**

```
boolean test-and-set (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

- Executed automatically. If 2 tests and set instructions are executed simultaneously, they will be executed sequentially in some arbitary order.
- Returns original value of passed parameter and sets new parameter value to True.

- Mutual Exclusion implementation

P_0

```
do {
    while (TestandSet(&lock));
        critical section
    lock = False;
        remainder section
} while(1);
```

- P_0 tries to enter CS. Set lock = true and TBS returns false.
- while (false), P_0 enters CS.
- Now, P_1 tries to enter CS. P_0 is still in CS.
- lock = true and TBS returns true.
- while(true), thus P_1 goes in infinite loop. P_1 does not enter CS.
- After P_0 has come out of CS, lock becomes false and it goes in remainder section.
- Now P_1 tries to enter CS as lock is false.
- while condition is now false and P_1 comes out of loop and enters CS.

(B) Swap Instruction

```
void Swap (boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

See for all in PDF

directly to semaphores now.

- Both test and set, swap do not satisfy bounded wait requirement.

* Semaphores

- Semaphore is a synchronization tool.
- It is an integer S that is accessed apart from initialization only through two standard operations: wait() and signal()
- Wait() operation originally termed 'P' from proberen
- signal() operation originally termed 'V' from verhogen.

```
wait(S) {
    while S<=0 ;
    // no op
    S--;
}
```

```
signal(S) {
    S++;
}
```

- When one process is modifying the semaphore value, no other process can simultaneously modify the semaphore value.

* Counting semaphore

- The value of a counting semaphore can range over an unrestricted domain.
- The semaphore is initialized to the number of resources available.
- When a process uses a resource, it performs wait() operation on sep semaphore and decreases the count.
- When a process releases a source, it performs signal() operation and increases the count.
- When the count reaches 0, all resources are used. After that if a process wishes to use a resource it will be blocked until the count becomes greater than 0.

* Binary Semaphore

- Value can range only between 0 and 1.
- Also called mutex locks
- Provide mutual exclusion.

★ Busy Waiting:

- No two processes can execute the wait() and signal() on the same semaphore at the same time.
- When a process tries to enter a CS while another process is already is in its CS, it must go in an infinite loop in the entry code.
- Busy waiting wastes CPU cycles that some other process can use productively.
- All Software Solutions suffered from busy waiting.
- When process is in waiting critical section, the semaphore used is spin lock. The process 'spins' while waiting for lock.
- When locks are to be held for a short time, spin locks are useful.
- One thread can 'spin' while other thread is in CS.

(See implementation of semaphore with no busy waiting)

★ Bounded Buffer Problem

(Producer-Consumer soln-3 using semaphores)

- Semaphore mutex — Binary semaphore
- Provides mutual exclusion for access to buffer pool
- Initialized value to 1.
- Semaphore full — Counting semaphore
- Counts no of full buffers
- initialized to 0.
- Semaphore empty — Counting semaphore
- counts no of empty semaphores.
- initialized to n.

- Structure of Producer Process

```
do {
    wait (empty);
    wait (mutex);
    // CS: add item to buffer
    signal (mutex);
    signal (full);
} while (1);
```

- Structure of consumer process.

```
do {
    wait (full);
    wait (mutex);
    // CS: remove item from buffer
    signal (mutex);
    signal (empty);
} while (1);
```

* Reader's Writer's problem

- Readers can only read dataset, do not perform any updates
- Writers can both read and write or update.

- Problem:
  - Allow multiple readers to read at same time, no adverse effect
  - If writer and some other process (reader / writer)

- Requirement:
  - Writer must have exclusive access to shared object.

1. Reader's - Writer's Problem:

- Semaphore rw-mutex ⟶ Binary semaphore
  Initialised to 1
  Common to both reader and writer
  Mutual exclusion for writers
  Also used by first or last reader that enters or exits mutex.
- Semaphore mutex ⟶ Binary semaphore
  Initialised to 1
  Ensure mutual exclusion when read_count is updated.
- integer read-count
  Initialised to 0
  Keeps track of how many processes are currently reading the object.

- Writer Process

```
do {
    wait(rw_mutex);
    // writing is performed
    signal(rw_mutex);
} while(1);
```

- Reader Process

```
do{
    wait(mutex);          // to mutex
    read_count++;         // read_count variable
    if(read_count == 1)
        wait(rw_mutex);   // ensures no writer can enter if there is even one reader
    signal(mutex);
    // reading is performed
    wait(mutex);          // reader leaves
    read_count--;
    if(read_count == 0)
        signal(rw_mutex); // writer can enter
    signal(mutex);        // reader leave
} while(1);
```

2. Writer Reader Problem:
   same code

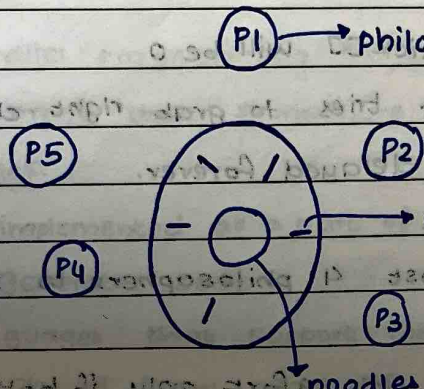
3. Writer writer problem
   Take only writer code

4. Reader reader problem
   Take only reader problem.

* Dining Philosopher's Problem.

- Philosopher's spend their life thinking and eating
- When thinking, they do not interact with neighbors
- When eating / hungry, a philosopher needs both left and right chopstick to eat.
- A hungry philosopher will not eat if both chopsticks are not available.
- A philosopher will pick up 2 chopsticks closest to her to eat.
- 1 Chopstick between every neighbor.
- Picks up chopstick one at a time
- Needs both to eat, release both when done.

- Dining Philosopher is a classic synchronisation 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 way.



- 5 philosophers share a common circular table. Bowl of noodles at the centre of table and 5 single chopsticks.
- Shared dataset:
  Bowl of rice (dataset)
  Semaphore - chopstick [5] initialised to 1. (Binary semaphore)
- Execute wait() operation if philosopher takes a chopstick
- Execute signal() operation if chopstick is released.

★ Semaphore Solution

```
do {
    wait( chopstick [i]);
    wait ( chopstick [(i+1) % 5 ]);
    // eat
    signal ( chopstick [i]);
    signal ( chopstick [ (i+1) % 5 ]);
    // think
} while (1)
```

Here, philosopher i has picked up both chopsticks and eating function is performed. Once, philosopher i is done eating, chopsticks are put down and it goes back in thinking state.

- The algorithm guarantees that no two neighbors are eating simultaneously.
- But it is still rejected because of deadlock.
- If all 5 philosophers simultaneously decide to eat and each grabs left chopstick
- All elements of chopstick[] will be 0
- When each philosopher tries to grab right chopstick, deadlock is created and it is delayed forever.
- To handle deadlock:

(1) We allow at most 4 philosophers to sit on table simultaneously.

(2) Allow a philosopher to pick a fork only if both are available. That is put the waiting operations in CS.

(3) Use an assymetric solution:
Odd numbered philosophers will first pick up left chopstick and then right chopstick.
Even numbered philosopher will first pick up right chopstick and then left chopstick.

- These solutions still cannot get rid of starvation.

* Monitors:

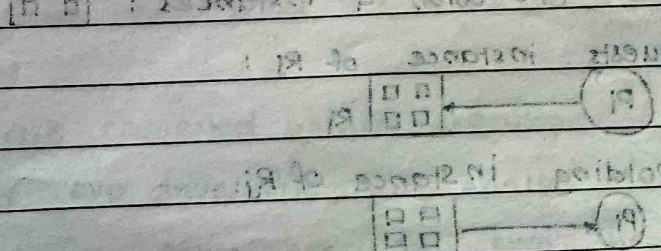
- High level synchronization construct
- Programmer-defined operators
- Declaration of variables
- Bodies of procedures

```
monitor monitor_name
{
// shared variable declarations
Procedure P1(...) {...}
Procedure P2(...) {...}
:
Procedure P3(...) {...}
Initialization code(...) {...}
}
```

- Collection of condition variables and procedures combined.
- Monitor ensures that only one process maybe active at a time.
- Programmer does not need to synchronize the code using constraints.
- x.wait() and x.signal() conditions can be used.
- Advantages:
  (i) Makes parallel programming easier
  (ii) Less error prone than semaphore
- Disadvantages:
  (i) Have to implemented as a part of programming language.
  (ii) Compiler generates code
  (iii) Some languages that support monitors
  (iv) Compiler should know what OS is and other burdens.

* Monitor Solution to Dining Philosopher:
- Philosopher can be in 3 states :
  enum {Thinking, Hungry, Eating} state[5];
- Philosopher will eat only if both chopsticks are available.
  Philosopher i can be in variable state[i] = Eating only if two adjacent neighbors are not eating
- (state[(i+4)%5] != Eating) and (state[(i+1)%5] != Eating)
- We also need to declare - condition self[5];
- ith philosopher can delay herself when she is ~~hn~~ hungry but unable to obtain chopsticks she needs.
- Distribution of chopsticks is maintained using monitor DiningPhilosophe

(See code from ppt)



* Deadlock:

- A process is in deadlock where every process is dependent on the other processes for an event to happen.
- Deadlocks can arise because of 4 conditions:

1. Mutual Exclusion:
   - Only one process can use a resource at a time.
   - If another process requests the resource, the requesting must be delayed until the process is released.
2. Hold and Wait:
   - Process holding one resource wants to acquire one more resource that is held by another process.
3. No preemption:
   - Process cannot be preempted.
   - A resource can be released only by the process after process is has completed its task.
4. Circular wait:
   - There is a set $\{P_0, P_1, P_2, P_3, P_4, \ldots, P_n\}$ waiting processes such that:

     P_0 is waiting for resource held by P_1.

     P_1 is waiting for resource held by P_2.

     ⋮

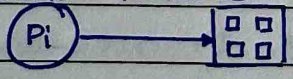
     P_{n-1} is waiting for resource held by P_n

     P_n is waiting for resource held by P_0.

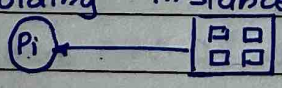
** Resource Allocation graph

- Request edge: $P_i \rightarrow R_j$
- assignment edge: $R_j \rightarrow P_i$
- Resource type with 4 instances:

- P_i requests instance of R_j:



- P_i is holding instance of R_j:



- If graph contains no cycles then no deadlock.
- If graph contains a cycle:
  deadlock exists.
- If only one instance per resource then deadlock.
- If several instances per resource type then deadlock.

* Deadlock Prevention:
- Avoid atleast one of the conditions causing deadlock

1. Mutual Exclusion:
- Mutual Exclusion not required for shareable resources
eg: If there are several processes requesting to access a read-only file they can be granted simultaneous access.
- Mutual exclusion must hold for non-shareable resources.
eg: A printer cannot be shared simultaneously.

2. Hold and Wait:
- Must ensure that whenever a process makes a request for a resource no other process is holding it.
- Method 1:
  - Process must request and be allocated all the resources before execution begins.
- Method 2:
  - Allow process to request resources only when there is none allocated to it.
  - Process can request additional resources, but it must release all the resources that are currently allocated.

eg: Consider a process that copies data from DVD drive to a file on disk, sorts the file and prints the result to a printer.

Method 1:
- All resources are requested before execution.
- Process requests DVD drive, file and printer initially.
- Printer is held for entire execution even though it was needed at the end.

Method 2:

- Process requests only DVD and file initially.
- It copies the data from DVD to file and sorts it and immediately releases both of them.
- Process again requests for file and printer
- After copying data to file and sorting it, it sends it to the printer and releases two resources again.
- For first method, resource utilization is low.
- For 2nd method, starvation is possible.

3. No Preemption:

- Method 1:
  - If a process is holding some resources and requests another resource that cannot be allocated immediately then, all the resources being held currently must be released
  - Process will restart only when its old resources and new ones are available on request.
- Method 2:
  - If a process requests some resource and if they are not available, we check if it is allocated to other process that is itself waiting for some other resource.
  - If so, we preempt the desired resource from the waiting process and allocate it to requesting process.
  - If resources are neither available, nor being held, the requesting process must wait.
  - While waiting some of its resource get preempted only if they are requested.

4. Circular wait:

- We impose a proper bordering of resource allocation.
- P1 and P2 both need R1 and R2. Locking both resources should be in a order like first lock R1 and then R2. Whichever process gets locks R1 first locks R2, and other process will wait for locks to release.

* Deadlock Avoidance

* Banker's Algorithm:

- A resource allocation and deadlock avoidance algorithm.
- It is named so to check whether loan can be sanctioned to a person or no.
- Suppose there are n account holders and total sum is S.
- If a money person applies for a ~~lone~~ loan, the bank will first subtract loan money from total amount. If the remaining S is greater than S then the loan is sanctioned.
- It is done because if all account holders came to withdraw money it could do it easily.
- Bank would never allocate money if it cannot satisfy the needs of all its customers.
- Bank always wants to be in safe state.

eg:

Process	Allocation				Max				Need				Available			
	a	b	c	d	a	b	c	d	a	b	c	d	a	b	c	d
P_0	0	0	1	2	0	0	1	2	0	0	0	0	1	5	2	0
P_1	1	0	0	0	1	7	5	0	0	7	5	0				
P_2	1	3	5	4	2	3	5	6	1	0	0	2				
P_3	0	6	3	2	0	6	5	2	0	0	2	0				
P_4	0	0	1	4	0	6	5	6	0	6	4	2				

- Need = Max - Allocation

- **Step 1:** P_0

  Work = Allocation = [1, 5, 2, 0]

  Need $\leq$ Work, so safe

  Sequence = $\langle P_0 \rangle$

  Work = Work + Allocation

  = [1, 5, 3, 2]

T	F	F	F	F

- **Step 2:** For P_1

T	F	F	F	F

  Work = [1, 5, 3, 2]

  Need $\not\leq$ Work, so unsafe

  P_1 must wait.

- **Step 3:** For P_2 — P_2 is safe

  Work = [1, 5, 3, 2]

  Need $\leq$ Work i.e

T	F	T	F	F

  Work = Work + Allocation

  = [2, 8, 8, 6]

  Sequence: $\langle P_0, P_2 \rangle$

For P_3

Work = [2, 8, 8, 6]

Need ≤ Work, P_3 is safe

Sequence = <P_0, P_2, P_3>

Work = Work + Allocation

= [2, 14, 11, 8]

Finish = | True | False | True | True | False |

For P_4

Work = [2, 14, 11, 8]

Need ≤ Work, P_4 is safe

Sequence = <P_0, P_2, P_3, P_4>

Work = Work + Allocation

= [2, 14, 12, 12]

| T | F | T | T | T |

For P_1

~~Wor~~ Need ≤ Work, P_i is safe

Sequence = <P_0, P_2, P_3, P_4, P_1>

Work = Work + Allocation

= [3, 4, 12, 12]

| T | T | T | T | T |

* Deadlock Detection:

- Allows system to enter deadlock state
- Single Instance of Each Resource Type: Wait-for-graph

Remove all resource nodes and collapse appropriate edges with approaches. If a circle still exists in graph then deadlock present.

eg: see pdf

- Multiple Instance: Banker's Algorithm.

* Deadlock Recovery:

1. Process Termination:

Abort one or more processes to break the circular wait.

- Abort all processes
- Abort one process at a time till DL is eliminated.

2. Resource Preemption:

Selectively preempt some resources until deadlock cycle is broken.

3 issues:

Selecting a victim: which resources and processes must be preempted, determine order & cost

Rollback: return to safe state and restart for that state.

Starvation