- int sem_timedwait(sem_t *sem, const)
  - wait for specific time. If not given lock within that time then come out.
- sem_post(sem_t *sem).
  - Increament the value of count by 1.
  - Non-blocking call.
  - wait() is blocking funct^n. but signal() is non-blocking funct^n.
- Header file for spin lock is pthread.h.

* Recursive lock -

- 'kind' value with 4 options.
- PTHREAD_MUTEX_NORMAL. - normal locking
- PTHREAD_MUTEX_RECURSIVE - Recursive locking
  - If a process locks itself n times recursively then it shall unlock itself n times to unlock completely.
  - A process can only unlock itself.

11/11/19 Producer-Consumer Problem - (Unbounded)

- Producer -

```
while (1) {
wait (mutex);
Produce;
Buffer[in] = item;
in = in+1;
signal (mutex);
signal (full); }
```

- Consumer -

```
while (1) {
wait (full);
wait (mutex);
item = Buffer[out];
out = out+1;
signal (mutex); }
```

- If a producer is producing then no other producer or consumer can enter the critical section.
- similarly if a consumer is consuming then no other producer or consumer can enter the critical section.
- Also there should be a check condition that if there is no item left (empty buffer) then consumer cannot consume anything.

Producer - Consumer (unbounded) -

- consumer same as bounded consumer.
- Producer should check if the buffer is free before producing.
- There is a fixed size of buffer.

Producer -

```
while (1) {
wait (empty);
wait (mutex);
Produce;
Buffer [in] = item;
in = in+1;
signal (mutex);
signal (full); }
```

Consumer -

```
while (1) {
wait (full);
wait (mutex);
item = Buffer (out);
out = out-1;
signal (mutex);
signal (empty); }
```

★ Readers - Writers Problem -

- If any reader is in the critical section then any no. of readers is allowed in the critical section but no writer is allowed.

- If a writer is in CS then neither any reader nor any other writer is allowed in CS.

- No readers and writer cannot execute simultaneously.

- Readers - Only reads the data set
- Writer - Can read and write both.

- Initialize wrt = 1; mutex = 1; readcount = 0.

Readers -

```
do {
    wait(mutex);
    readcount++;
    if(readcount == 1)
        wait(wrt);
    signal(mutex);
    // Reading is performed
    wait(mutex);
    readcount--;
    if(readcount == 0)
        signal(wrt);
    signal(mutex);
} while(TRUE);
```

writer -

```
do {
    wait(wrt);
    // write
    signal(wrt);
} while(TRUE);
```

- Reader can dominate writer i.e. a reader can execute before writer but not vice versa.

* Dining Philosphers Problem :-

- A person can have food if it's left and right chopsticks both are free.

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

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

- Deadlock if context switch happens after first wait and everyone picks it left/right chopstick only.

3/11/19 Monitors :-

- Implemented by using semaphores.
- Only one process will be active within the monitor ~~even if there are multiple processes inside the monitor.~~
- wait() & signal() of monitor is diff. from that of semaphores.

- wait() – A process that invoke the op. is suspended in until another process invokes x.signal().

- signal() – Resumes exactly one suspended process. If no process is suspended then does nothing.

- Entry Queue – For process outside queue monitor that wants to enter

- Condition Variable – To stop the ex. of multiple process inside the monitor simultaneously.

– Suspended processes by x.wait().
– Three process are inside monitor but not ready to execute.

* Conditional Variable Choices –

P – Executing in monitor – Gives x.signal to Q.

Q – In suspended state by x.wait().

– Signal & wait – P waits until Q either leaves the monitor or Q waits for some other condition.

– Signal & continue – Q wait until P gets suspended/terminated.

* Dining

* Monit

– Pick

& Q

one

time

- x.

if (x

}

- If

15/11/19 Dec

* Dining Philosophers Problem –

* Monitor Implementation using Semaphores –

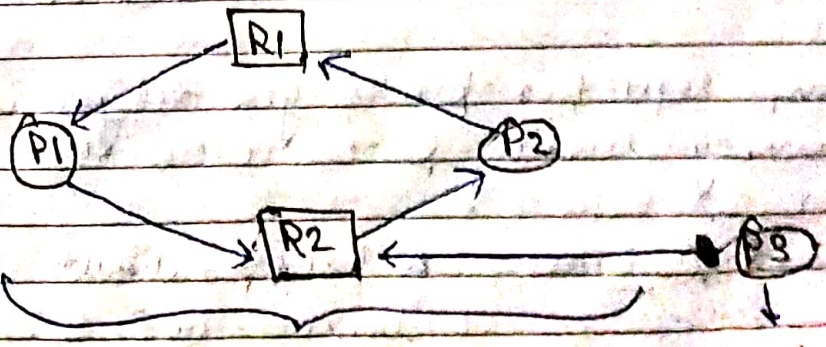
– Pickup() & putdown() function uses wait(mutex) & signal(mutex) so it makes sure that only one process will ex. pickup() or down() at a time.

- x.signal()
  if (x_count > 0)
  {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
  }

  x.wait()
  x_count++;
  if (next_count > 0)
    signal(next);
  else
    signal(mutex);
  wait(x_sem);
  x_count--;

- If P signals for Q and blocked itself, then when Q gets blocked by some other condition or Q gets ex. completed, it will make P signal P.

5/11/19 Deadlock Problem :–



Deadlock

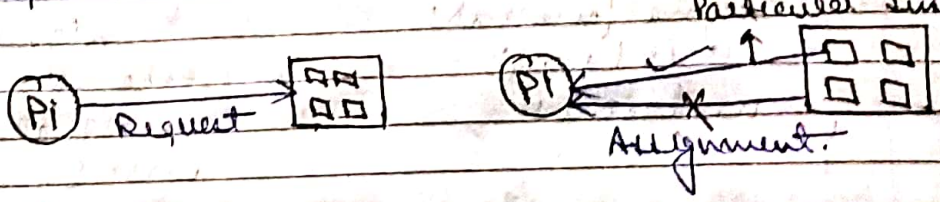
Not a part of deadlock

- Deadlock occurs when some processes are holding few resource and req. for other resource.

- A process which is not holding any resource will not be a part of deadlock.
- ~~A process which is not requesting~~

* Resource Cycle - Request → Use → Release

- Assignment - From resource to process
- Request - From process to resource.



* Deadlock Characterisation - 4 Conditions req.

• Mutual Exclusion - Only Atleast one resource should be non-sharable i.e. at a time only 1 process can use it.

Eg:- File in Read Mode - No mutual Ex. - No deadlock
" " Write " - Mutual Ex - Deadlock.

- Any requesting process gets delayed until the currently running process releases that resources.

• Hold & wait - A process should hold atleast 1 resource and req. any other resource to cause deadlock.

- A process not holding or not requesting can never be a part of deadlock.

- **No Preemption** – Almost all resources we use are non-preemptable by nature.

- If resources are preemptable then a process can release it without completing its work and avoid deadlock.

- **Circular wait** – A circular cycle of request by the processes should be formed.

- All the 4 conditions should be met for deadlock. If any 1 of these condition is not met then deadlock can be avoided.

* Handling Deadlock –

- **Prevention** – Not allow system to enter deadlock state.
  - Too costly to implement.
  - Prevent atleast 1 of the any 4 condition.
- ~~Avoidance~~ **Detection & Recovery** – ~~Avoid~~ Allow the system to enter deadlock and then recover
  - Followed by most OS.

* Deadlock Prevention –

- **Prevent Hold & wait** – If a process holds R1 & req. R2 then it frees R1 and requests for R1 & R2 together.

~~– Low resource utilization – Starvation possible.~~

- Req. all the resources at once & before beginning execution.
- Starvation possible.

- ~~Prevent~~ ~~No P~~
- Prevent no Preemption -

Eg:- Priority Inversion - Priority is also a resource. Give your highest priority to some other lowest priority process and take it's lowest priority.

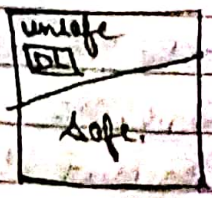
- Prevent circular wait - Take resources in a particular order only.

- If a process has R_2 & he wants R_1 then he has to release R_2, take R_1 first and then R_2 (If R_1 comes first in order than R_2).

- Requires preemption to give up resources.

18/11/19 Deadlock Avoidance :-

- Prevent a process to get into unsafe state even though only a small % of unsafe is deadlock.



- Underperformance / Underutilization.

- Information needed -
  Max [n][m];
  Allocated [n][m]
  Available [m]

	Max Need	Currently Holding
P_0	10	5
P_1	4	2
P_2	9	2

Total - 3 process, 12 resource available

- Currently available = 12 - (5+2+2) = 3.
- System goes only in safe state. So resource will not allocate any resource to P_0/P_2 because they need more than 3 ~~so~~ - unsafe state. ~~will~~
- Allocate 2 resource to P1 → P1 finishes and release all 4 resource.
- Now available 5 resource → Allocate to P0 → P0 finishes and releases all 10.
- Now given 7 to P2 and P2 executes.

- If we allocate P1 2 resources and P2 1 resource then after P1 executes and releases all 4 resource it will go into ~~not~~ unsafe state so P2 is not given resource even if the resource is free.

* Avoidance Algorithm

• Single Instance - 2 edges -
Resource.
i). Req. (Process → Resource)
ii). Assignment (Resource → Process)
iii). Claim Edge (Proc → Resource)
- represented by dashed line to represent that a process may req. that resource in future.

- used to check if a claimed resource is allocated or then whether whether that we may result in unsafe state in future or not.

- If a process req. a resource and there forms a cycle including claim edge then the resource it may result in deadlock so the resource will not be allocated to the requesting process.

• Multi Instance Resource - Banker's Algorithm.

- Multiple inst. of each resource is available.
- Underutilized resources - inefficient.

		Allocation	Max.	Available
		A B C	A B C	A B C
B	P_0	0 1 0	7 5 3	3 3 2
	P_1	2 0 0	3 2 2	
	P_2	3 0 2	9 0 2	
	P_3	2 1 1	2 2 2	
	P_4	0 0 2	4 3 3	

— Total Available resource = (Σ Allocation) + Available currently.

Need = Max. − Allocated.

	Need		
	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Available

A	B	C
3	3	2

— There must be atleast 1 seq. of ex. so that the system never goes to unsafe state.

— Allocate to P1 — 122
P1 ex. and release —
Allocate to P3 — 011
P2 ex. and release —
Allocate to P0 — 443
P0 ex. and release
Allocate to P4 — 431
P4 ex. and release
Allocate to P2 — 600
P2 ex. and release

Available

	A	B	C
	3	3	2
P1	2	0	0
	5	3	2
P3	2	1	1
	7	4	3
P0	0	1	0
	7	5	3
P4	0	0	2
	7	5	5
P2	3	0	2
	10	5	7

← 10 5 7 Total resource Available.

- If req > need then it is an error as the process is trying to ask for more resource than the max allowed for that process.

19/11/19

* Deadlock Detection :-

- Wait-for Graph - Remove the resources in the resource allocation graph.

- If there exists a cycle then it is a deadlock

*

	Allocation	Req.	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Any process with 0 0 0 allocation or req. will not be a part of deadlock.

- Execute P_0
  Free P_0
  Execute P_2
  Free P_2
  Execute P_3
  Free P_3
  Execute P_4
  Free P_4
  Ex. P_1
  Free P_1

Available

A B C

0 0 0 → T_0

0 1 0 → P_0

0 1 0

3 0 3 → P_2

3 1 3

2 1 1 → P_3

5 2 4

0 0 2 → P_4

5 2 6

2 0 0 → P_1

7 2 6

	Allocation	Req.	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	0 1 0 — P_0
P_2	3 0 3	0 0 1	0 1 0
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Execute P_0
  Free P_0

- Now with the available resources no process can complete so deadlock situation.
- Except P_0 all the processes are part of the deadlock.

- Detection Algo. Complexity = $O(m \times n^2)$.
  Very heavy so this algo. is run only when utilization is less than a certain % which happens if some processes are caught in deadlock state. (Usually 40%).

  m = No. of resource
  n = No. of Processes.

Recovery from Deadlock:-

i). Recover Manually — kill process manually.
- kill a process which is a part of many cycles.
- Or a process which did min work till now
- least or least imp process.

study time
Page No.:
Date: / /

19/11/19

ii). Recover Automatically –

- Process Termination
- Resource Preemption – If a resource is preempted then the process has to go & run the code again from the start.

- Incremental Backup – Backup only the diff. from the last time the same file was backed up.

– While restoring it will take an old complete file and process those backups to modify it to the recent file.

Memory Management

- Memory protection happens due to two reg — base reg and limit reg.

- CPU always generates logical add. — This needs to be converted to physical add.

- Compile Time Binding — During compilation the physical add is specified of the code.

— If the program needs to run at other system then it has to be recompiled on that system.

— If the code has to stored in some other mem. location in the same system then also it has to be recompiled.

Eg:- • Starting of OS.
• Vector Tables.

- Load Time Binding — At load time the starting location is generated.

— At compilation logical add. until the starting add is generated for the code.

— To change the starting add. the prog. needs to be reloaded.

Eg:- • CS, IP in MMU.
↓ ↳ logical add of each inst. must be start time.
Starting add.

• Execution Time Binding – ★ Physical add. is generated during the run time of the prog.

– Prog. can be loaded at any place in the memory.

– Needs additional hardware support – Page table and segmentation table.

– Followed by us now.

20/4/19 ★ Protection :-

– ~~Two All logical~~

– Logical add. for all the process starts at 0 and should be less than limit reg. ~~value~~ value of that process.

– Logical add. is then added to the base value for that process to generate physical add.

★ Dynamic loading – In a prog. If a function is there, then it's code and data is not loaded in the memory until and unless it is called for the first time.

- All static linking libraries are also dynamically loaded i.e. it because the functions inside the libraries are implemented as modules.
- The libraries are only linked while making the executable file.
- Dynamic linking libraries are linked to the when prog. during ex. only when it is called for the first time.
- Dynamic linking again uses dynamic loading to load diff. modules in the memory.
- Header file includes all the function declaration but no implementation. The contents of header file is copied to the main program.
- Linker Implementation of functions is present in another file (libc.so). So linking means linking to this file.
- Dynamic linking takes less time during ex. than static linking because of stub add. replacing.
- Stubs are created during compilation.

* Swapping :- The complete process is not loaded in the main memory.

- Some blocks are present in main memory and other blocks in swap (secondary) memory.

- The blocks that are req. are swapped in from swap memory to main memory.

- Swap out can be that program own unused block or can be some other program's unused block.

- In compile time binding or load time binding since the physical add. is generated and fixed before exe. of prog. so while swapping in the ~~code~~ program back the back should be loaded at that block of memory only even if other blocks are unused in memory.

- In Exe. time binding, page tables and segment tables are used which are updated acc. if accordingly.

* Contiguous Allocation -

* ① Multiple Partition - Each process has one partition in the memory.

• External Fragmentation - Unused blocks of memory

process is not
the main memory

main memory
ap (secondary)

are swapped in
main memory.

gram own unused
e program is unused

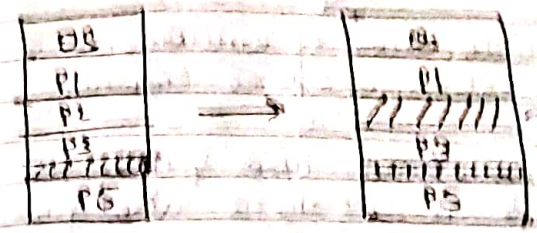
load time binding
is generated and
g. so while
ogram fails, those
that block
ese blocks are

takes and reg-
n are updated

has one parti-
memory.

ks of memory

are created with a portion of them when process are executed between two processes.



→ not big enough to load another prog. An unused region (maybe upto 50 %).

- Process cannot be rearranged in the main memory because it will create dangling pointers.
- Internal Fragmentation - When the block size is fixed in main memory.
  - If prog is smaller than
  - Then, a block is allocated to any process.
  - So if the prog is smaller than the block size it results in internal frag. i.e. some part of that block remains unused.
  - Internal frag. is caused when the blocks are not of fixed size.
- Paging - memory is divided into multiple pages of fixed size (usually 4K).
  - Hence Internal frag. may be caused.

- Segmentation – No fixed size of blocks of memory.
  - Dynamic allocation of memory.
  - Int fragmentation may be reduced.
  - So both segmentation and paging are used. Each segment is allowed two types of fixed size to prevent ex. fragmentation.
- Partition – Fixed size.
  - One process can occupy only 1 partition.
- Dynamic Partition – No fixed size of partition
  - Resize on store.
  - Run fit also to find hole to allocate to a process.

i). First Fit – Start searching from the start of memory to find the first hole that will accomodate the process.

ii). Next Fit – Start searching from the pt from where the last processure executed.

iii). Best Fit – Search for the complete memory to find a hole which will result in least wasted space if allocated to a process.