Nice Value	Static Pr.	Quan. time
-20	100	800 ms
:	:	:
-1	119	420 ms
0	120	100 ms
1	121	95 ms
:	:	:
19	139	5 ms

- Normal users cannot set -ve nice values - Only superuser or admin can.

+ Dynamic Priority - It should give sup. to I/O bound / more -ve processes over CPU process.

- Dy Pr. = St. pr. - bonus + 5

- Bonus value is based on sleeping time i.e. how much time the process waits in I/O device and not ready queue

Avg. sleep time (ms)	Bonus	Granularity
0 - 100	0	5120
100 - 200	1	2560
200 - 300	2	1280
300 - 400	3	640
400 - 500	4	320
500 - 600	5	160
600 - 700	6	80
700 - 800	7	40
800 - 900	8	20
900 - 1000	9	10
1s and greater	10	10

process has to preempt and give CPU to new process. After the execution of new process, the previous process will continue execution only for the remaining time quantum.

- All the process of 1 priority level finishes execution before executing lower priority (higher no) process.

- Starvation occures in real-time processes also but there is no mechanism to stop that in real-time process.

- sched_yield() - Sys. call to give up CPU by a running process voluntarily.

- The process will execute with next time with the remaining time quantum only.

- Any joining from outside (i.e. new admission or from blocked/suspended state) will join at the back of the queue.

- The process which is already in the CPU (i.e. currently running) will have option to join in front (FIFO) or back (RR).

- At every decision pt. the dynamic priority of currently process changes but quantum time is changed only when the present quantum time is expired.

Page No.: study time
Date: / /

Windows

- Push migration algo. uses past history of the core to determine which is busy and which is free / idle.

Windows CPU Sched.:-

- Priority: 0 – Mem. Management
  1-15 – Variable
  16-31 – Real time.
- One queue for each priority level.
- Higher no. – higher priority.
- High priority to int. process – mouse / window.
- Variable class:- HIGH_PRIORITY_CLASS
  ABOVE_NORMAL
  NORMAL
  BELOW_NORMAL
  LOWEST
  IDLE
- These classes present in both Real time Range and Variable Range.
- From windows 7 – Two levels of thread sch.
  - User level thread scheduling
  - Kernel level thread scheduling.

25/10/19

+ kernel - level scheduling -
- Preemptive scheduling.
- Includes process affinity change.

25/10/19 Synchronization :-

- Independent Process - If a process is not dependent on any other process data wise or temporarily.

- Cooperating Process - If a process is dep. on other process.

- Most the processes are cooperating process.
Two mechanism - i). Shared memory
ii). Message Passing

- If data size is large then shared memory is better because no. of msgs will also be great large.

- If security is concerned then message passing is better.

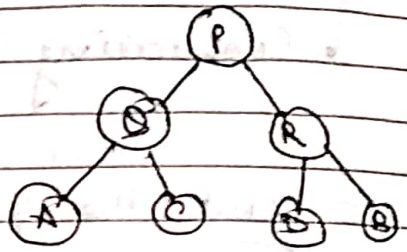
- Msg Passing - A process ~~son~~ sends msg / signal to kernel and then kernel forwards it to other process.

- Secure because kernel handles the msgs.
- If data is large then no. of msgs will also be large so system degradation.

- If SIGCONT reaches the child process before

the SIGSTOP signal then after the child process is blocked by the latter, it cannot continue ever because no other SIGCONT is going to come to it. To avoid this we use WUNTRACED in parent process.
- No race condition in msg passing because no shared space available, data is sent/ recieved b/w the processes.

• Communication link—



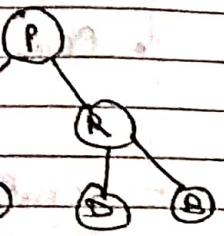
If Process A needs to send a msg/signal to process B then it will go through process P.
- So if P is stopped/blocked then A & B no link is present so communication not possible.

- A & C can communicate via process Q irrespective of state of process P.

• Direct :-
- Exactly one link is associated b/w exactly one pair of process.
- One to one comm. - sender & receiver
• Direct link.

• Mailbox - Multiple person will get mail/ msgs at a common mailbox and the process can take it from there.

the child
e, it cannot
SIGCONT
d the use
ess.
because no
not sent/



en A or
cation not

rocess Q

ue exactly
& receiver

et mail/
mail box
om there.

- Blocking Send – Sender blocked until the msg. is received.
  Blocking Receive – Receiver blocked until a msg is available.
  Eg - SIGSTOP – receiver waits until SIGCONT is received.
- Non-Blocking Send – Sender sends the msg and continue.
  Eg – SIGCONT.
  Non-Blocking Receive – Receiver receives a signal or a NULL to continue.
  Eg:- Exit status of a child process.
- Synch. Problems –
  – If a msg is lost or process fails before ending, receive process is permanently blocked in case of blocking receive – deadlock.
- TCP – Msg acknowledge given by receiver to the sender.
  Eg:- File transfer transfer blw diff. system.
  – If acknowledgement is not received then the sender will send the msg/packet again.
- UDP – No msg acknowledge is given by receiver to sender.
  Eg:- Video call.

- No point in sending a packet of data again at a later time because that data is not req. later.

- TCP is reliable.
  UDP is faster for smaller msg but no reliability.

- In case of non-blocking receive if the process executes receive before msg is sent, the msg gets lost.
  Eg- Child Parent
  SIGSTOP ←
  SIGCONT
  SIGSTOP
  SIGCONT →

- If the child misses it's signal then both child and parent will get forever blocked permanently at their resp. SIGSTOP signals — Deadlock.

★ Producer - Consumer Problem — (Buffering).

* Zero Cap. — Whenever the producer produces, consumer has to consume.
  — Strict Scheduling — P1→C1→P1→C1 . . . .

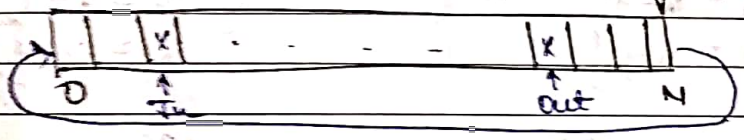
* Bounded Cap — Fixed cap. of N.
  — Producer checks if the buffer full.
  — Consumer has to check if the buffer is not empty.

- Infinite Buffer – Infinite length buffer.
  - Consumer has to check if buffer is not full.
  - Producer need not check if the buffer is full.

- In a bounded buffer, once the prod./cons. reaches the end of buffer they will have to come back at the start and continue from there (Circular production/consumption).
- This does not happen in unbounded buffer.

~~B ≡ in → 0, out → 0~~

- If in == out – Buffer empty.



- The place where 'in' is pointing is empty i.e the producer has to fill the object there next.
- The place where 'out' is pointing is not empty i.e the consumer has to take the obj from that place.

- If Since in place is empty so it is difficult to check for buffer full in this mechanism.

- So maintain a counter.
  - If count == 0 – Buffer Empty
  - If count == Buffer_size – Buffer Full.

- Race condition may occur because prod. and consumer are running in diff threads so they will try to increase and decrease the counter resp.

5/11/19 - If there is only one producer and consumer then IN & OUT variable can be a local variable.

- If more than one producer/consumer then IN/OUT must be a global variable. - Race condition

* for(i=0; i<100; i++) { x = x+1; x = x-1; }

for(i=0; i<100; i++) { x = x+1; x = x-1; }

- $x_{max} = 199$; $x_{min} = -199$.

* for(i=0; i<100; i++) { x = x+1; x = x-1; }

for(i=0; i<100; i++) { x = x-1; x = x+1; }

- $x_{max} = 200$; $x_{min} = -200$.

- To prevent race condition make the operation atomic - either the operation execute completely or do not execute at all.

* Critical Section - That section/set of code in a program which acts as atomic operation.

- If context switch happens in blw the exe.

of critical section then a next time when the process executes again it will have to execute that section from the start again.

- Or if there is a shared variable/file then a process using it can lock it while it is executing so that no other process can use that and race condition is avoided.
- After the process executes it unlocks the variable/file for other processes.

+ do {
  Entry section — only 1 process at a time will be able to go through this line.
  Critical section
  Exit section
  Remainder section
} while (1).

- (flag == 0) — check
  Critical code
  make flag = 1;

(flag == 1) — Check
Critical code
make flag = 0;

- So only 1 process will be able to execute that at a time.

• Min. 3 properties of Critical section Problem -

1). Mutual Ex. — If any process Pi is ex. at its critical section then no other process should ex. it's critical section.
- Others should wait at entry section.

study time
Page No.:
Date : / /

ii). Progress - If no process is ex. critical section and some process want to ex. it's critical section then that process should not be delayed indefinitely.

iii). Bounded waiting - If many process wants to exe. critical section then one process should not go again and again to ~~avoid~~ ~~starvation of other processes~~. also give other processes a chance.

6/11/19.

- These conditions are there for the same critical section of a prog. i.e. multiple process trying to execute the same piece of code.

→ Eg :- • Barber's chair
- Multiple faculty and student at the same time.

- File Reading is not mutually exclusive - ~~Any~~ Multiple process can exist simultaneously - ~~is~~ not a critical section problem.

- Alg 1 - No progress if any 1 process dies. (Turn Algo)
- Alg. 2 - If both flag[0] & flag[1] becomes (Flag Algo) true due to ~~and~~ context switching then both process won't be able to ex. critical section. - Deadlock - Fails progress.

ee. critical
ne. process want
then that
yed indefinitely.

process wants
itical section
not go again
vation of other
rocesses to a

the same critical
ple process
piece of code.

ident at the

exclusive —
simultaneously
problem.

e algo. (Turn Algo.)
[7] because
x-t switching
it be used
— Deadlock

• Algo. 3 — Peterson's Algorithm

— Uses both algo. 1 & 2 to overcome each other problem of progress.

6/11/19. — Two shared variables — turn & flag.

— Deadlock will not happen — either of any one process can always enter critical section.

— Bounded waiting due to turn variable — Both process will get a chance one after the other.

— This algo. is good only for 2 process, so for more than two processes this is not used.

— To enter the critical section, each process has to check multiple condition which would take very long time and in some other process may change the variable during that time ∴ Race condition during condition checking.

• Algo 4 - Bakery Algorithm

— For more than 2 processes.

— Each process gets a token no. — The no. to one with smallest token no. gets a chance to ee. critical section.

— If multiple process has same token no. then the one with smallest pid will enter critical section.

- Token no. is a ~~1~~ integer array in shared memory.
- If a process does not want to ~~en~~ critical section then it's no. is 0. Only
- So only those process will get a no. which want to enter critical section
- while giving token no. to a process, race condition may arise so multiple process may get the same token no.
- After a process has executed the critical section once it will make it's token no. 0.

• number[j,j] < number[i,i]
- First ~~of~~ compare the token no, then then check the pid.

- Token no. is given as First come First Serve algo. No no starvation.

★ H/W Synchronisation -

- OS controls main memory but not the cache - Cache is transparent.
- This is because OS is slower than cache.
- Disabling the interrupt will prevent time slice expiry to maintain synchronisation
- only for uniprocessor micro.

2/11/19

• swap / test and set — HW implemented inst. for synchronisation.

- All inst. in an inst. set architecture is atomic. (Add, sub, load, etc.)

- So disabling the int. before critical section will prevent preemption.
- Enable the int after critical section.

- If a process makes a system call, then it may be preempted by the OS even if int. is disabled for it.

- Not efficient for multicore-multiprocessor system.

* Inst. cycle — Inst fetch → Inst decode → Execute → load mem → write back → Exception handling.

- Special atomic H/w inst. are used for synchronisation instead of int. disable.

* Memory Barriers :—

• MESI — Modified Exclusive Shared Invalid
- Used for synchronisation b/w multiple cores.

- Modified — If data is modified in L1 cache of core 1 but still not updated in the shared L2 cache.

the most recent data in the L1.

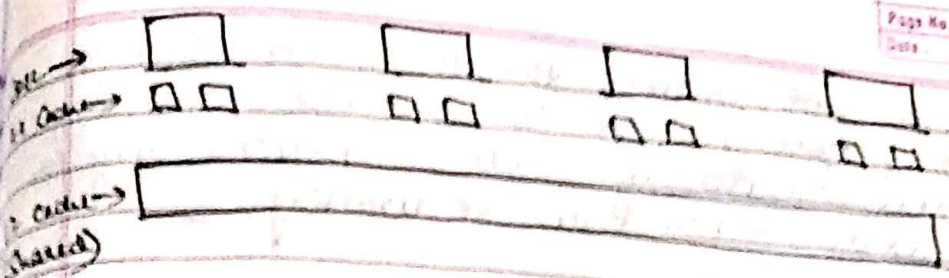
Exclusive - The most recent data is in L1 cache and only one core has that data in L1.

Shared - The most recent data is present in L1 cache of multiple cores.

Invalid - The data is present in multiple cores. So if one core modifies the data in it's L1 cache then the data of other cores becomes invalid.

- Strongly ordered - If one core modifies the data in its cache then it also tells the other cores to modify all their data.

- Loosely / weakly ordered - If one core modifies the data in it's own cache then it makes the data in other core as invalid.
  - When any core eg. the modified data it can request from the first core.
  - Loosely is better than strongly because in strongly, for each update a core needs to send a signal to each core but for weakly a core needs to invade send invalid signal only once and the no. of signals is reduced.



- In bus if n devices are connected then at a time only two devices will be active (sender and receiver) and other n-2 will have to wait ~~till the previous~~

Test & set Instruction

- Works only with shared memory - Tightly coupled system.
- Not for loosely coupled system.
- lock is a shared memory so only one process can access it at a time.
- Test and set is an atomic inst in the inst. set architecture so it won't be preempted in b/w.
- lock will be true whenever a process is in Critical section so others will be in while loop.
- When the process in Critical section finishes it will make lock = false so that any other process can ec. next.

Page No.: study time
Date: / /

- Does not satisfy the third condition because any process can be selected to execute next randomly, so a process may be starve - No Bounded waiting.

* Swap Inst:-

- Swap is also an atomic inst.
- Shared variable lock and local variable key for every process.
- Both mechanism are similar - In Test & Set return value is used but in swap local key variable is used.

• Spin lock - while one process is executing in Critical section the other processes will be executing while loop infinitely.

- Now if the other process has higher priority then the process in critical section then CPU will always be with the other process as spin lock will not end. - Deadlock (only in unicore systems).
- Spin lock is bad for unicore CPU but not bad in multiple core CPU.

* Bounded Waiting with Testand set -

- Use a shared data structure waiting[n].

9/11/19

- After a process has completed compleed execution of Critical section it will allow the next process to enter the Critical section so Bounded waiting is satisfied.

11/11/19 Semaphores :-

- Uses Test and Set and Swap or for synchronisation. - Software sol^n for the n/w inst.
- Special variable called semaphore, used for signaling. - Stores int. value.
- Int variable 'var' - Only two op. possible - wait() - Decreases 'var' by 1 } Atomic signal() - Increases 'var' by 1. } inst.
- Only 1 wait() or signal() can execute at a time.

P_0	P_1	P_2
wait(S)	wait(S)	wait(S)
CS	CS	CS
signal(S)	signal(S)	signal(S).

- S is initialised with 1. So when any 1 process execute wait, it will enter CS and make S=0. So no other process can ex. CS. When that process finishes ex. It will make S=1 and any other process can ex. wait and enter CS.

- If more than 1 process needs to be in C.S. then S can be initialized with a value greater than 1.

- Since wait and signal are atomic no any process can ex. wait/sig at a time. So no race condition.

- Critical section is not atomic so any other preemption can happen in the middle of critical section but some other process can not enter critical section till the first process finishes.

- **Counting Semaphore** = The above algo. is called counting semaphore because it keeps no. of process that can enter crit. sect^n.

- **Binary Semaphore** = Its value can be only 0 or 1 - uses only 1 bit and not int. value.
  - So only 1 process can enter crit. sect^n at a time
  - Also called mutex lock.

- Most of linux kernel does not support counting semaphore
- Counting semaphore can be implemented using binary semaphore.

- Busy Waiting / Spin lock — If a process is in crit. sect then a other process is waiting in CPU for the first process to complete.
  - If a higher priority process comes then it will keep on executing while loop in CPU so starvation / deadlock. — only in unicore processor
  - Helpful in multicore processor because bcz the process which is waiting is already available in some other core so it saves the context switching time.
  - For shorter lock time it may be useful.
  - so all both Counting and Binary semaphore and spin lock are available in std. posix library.
  - Since semaphor is in a shared memory so this algo. works in ~~tightly coupled~~ ~~tightly coupled~~ ~~coupled but not in~~ tightly distributed systems but not in loosely distributed systems.
  - wait() and signal() should be atomic so it becomes a Critical section problem — Test and Set and swap is used in these functions.
  - Instead of Test and Set or swap, disabling int. can also be used but not recommended.

• Blocking Implementation — Not spin lock.

— If a process is in busy waiting in CPU then block it. Change its state from running to blocked or waiting state so that other process can come in CPU.

— When the process completes CS and executes signal() then unblock that prev process.

— ~~This is not~~
— The blocked/waiting queue is in FIFO so bounded waiting is satisfied.

— It also has busy waiting part but for very short time until the process gets blocked.

* Deadlock & Starvation —

— Deadlock may occur if more than 1 semaphore is used and whether there is context switch b/w or after ex. wait().

— To come out of deadlock, priority inversion can be done i.e. priority of both the processes is inverted so any 1 process can execute either ex. signal(s) at the end.

* Counting Semaphore using Binary Semaphore: —

— In signal() — The signal (S1) for wait(S1) of the 1st the first line is present at the last statement of wait() operation.

Scanned by CamScanner

Page No.: study time
Date: / /

* Header File — semaphore.h
• int sem_init(

• pthread = 0 — Semaphore shared b/w threads.
  pthread = 1 — " " " process.
• value — Initial value of semaphore for no. of processes allowed in CS.

* Named semaphore —

O_CREAT — Create a new semaphore
O_EXCL — Exclusive

— Returns a name identifier.

• int sem_wait(sem_t *sem).
— Fr. wait() of.
— Decreament value by 1. (semaphore count).
— Block the process if any other process is already in CS.
— Decrament only if value > 0. so count can never become -ve. so no. of waiting process cannot be tracked.

• int sem_trywait(sem_t *sem)
— Same as sem_wait except if decrement cannot be performed then it will return an error and not get blocked.

- 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 lock
    - If a process locks itself n times recursively then it shall unlock itself n times to unlock completely.
    - A process can only unlock itself.