Tuesday, April 5, 2022 3:31 PM

Signal

trap -l

https://man7.org/linux/man-pages/man7/signal.7.html

Belady's Anomaly ---- Page Replacement algorithms

Question - If the number of frames increase then will the page faults and page replacements reduce

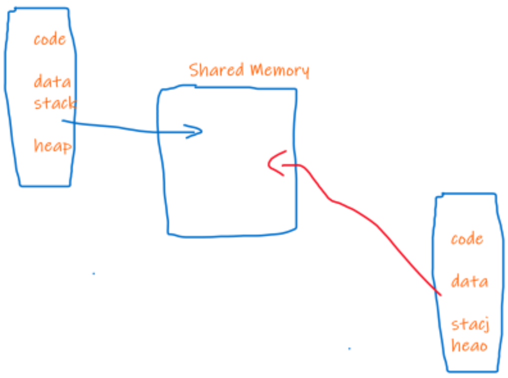
F1	
F2	
F3	
F4	
F5	

FIFO Page replacement ---- page faults may increase with increase in page frames!!
LRU /MRU } OPT ------ page faults decrease with increase in page frames!!!

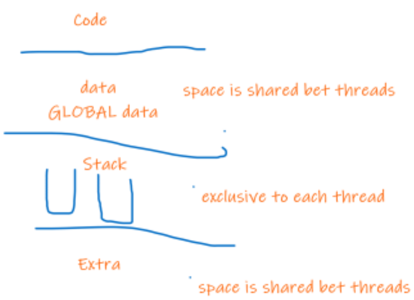
Signals
Deadlocks
Semaphores-----------------------

---

1. TWO Processes SHARE DATA ------ SHARED MEMORY IPC



1. TWO threads share memory ----



PROBLEM due to DATA SHARING between processes or data sharing between threads-------

RACE CONDITION

EXPECTED FINAL BALANCE = 10000 +3000=13000 13000 -900 = **12100**

Thread1		Thread2

Void deposit (amt) 3000
{
Temp = balance; 10,000
Temp = temp + amt; 13,000
Timer interrupt
 	
Balance =temp
}
	Balance=10000
	
	Balance =9100
	
	Balance =13000
		Void withdraw(amt) 900
		{
		Temp = balance; 10,000
		Temp = temp - amt; 9100
		Balance =temp
		}
		over

RACE CONDITION is dangerous --- because you can never know when it occurs !!!!

Solution to race condition is CALLED as mutual exclusion -----------
CRITICAL SECTION execution should be protected by a lock

CRITICAL SECTION = code that is using shared data
When thread1 executes a critical section thread 2 should now execute its own critical section !!!!!!

Thread1 **Thread 2 should wait till thread 1 finishes!!!**
{
//CS
.....
...
}

MUTUAL EXCLUSION ---- CS is exclusively controlled by the thread who first takes it
Then when the first thread is DONE using the CS it releases it
THEN another thread will exclusively use the CS
MUTUAL EXCLUSION ----------**MUTEX ---- FLAG**

Thread1		Thread2

Void deposit (amt) 3000
{
 	
FLAG is open so enter CS
CS begin
FLAG =CLOSE
 	
Temp = balance; 10,000
Temp = temp + amt; 13,000
Timer interrupt
 	
Balance =temp
 	
CS end
FLAG = OPEN
 	
}
	Balance=10000
	
	FLAG=OPEN
	FLAG - CLOSE
	
	Balance = 13000
	
	FLAG =OPEN
	FLAG =CLOSE
	
	Balance =12100
	FLAG =OPEN
		Void withdraw(amt) 900
		{
		
		IS the FLAG OPEN??
		FLAG is OPEN, enter CS
		CS begin
		FLAG=CLOSE
		Temp = balance; 13000
		Temp = temp - amt; 12100
		Balance =temp
		
		CS end
		FLAG =OPEN
		}
		over

RACE CODITION PROBLEM is solved by using **MUTEX** ----- process **synchronization** or thread **synchronization**

WHAT if the MUTEX has a race condition !!!!!

HENCE MUTEX is a special data type that does not have race condition !!!!
This MUTEX is called as SEMAPHORE !!!

Semaphore is an int type --- such that when this is checked and modified interrupt does not OCCUR !!!!

Semaphore is modified using TWO special ATOMIC functions WAIT (P) and SIGNAL (V)

wait(sema)

{
If SEMA is 0 (CLOSE)
WAIT
Else if SEMA is 1 (OPEN)
SEMA = 0 (CLOSE and return)

}

SIGNAL (sema)
{
Sema ++ , OPEN

}

Semaphores are of TWO Types ---
1. MUTEX is a BINARY Semaphore ---------OPEN and CLOSE (toggle between 0 and 1)
2. COUNTING SEMAPHORE --- its values change from 0 to n , n to 0

TOO much synchronization is BAD !!!!!

When the code uses mutex code is called as THREAD SAFE code , THREAD SAFETY can be achieved by MUTEX .

Deadlocks ---

DEADLOCK -------------- PROBLEM ---- process waits infinitely/ indefinite time for a RESOURCE

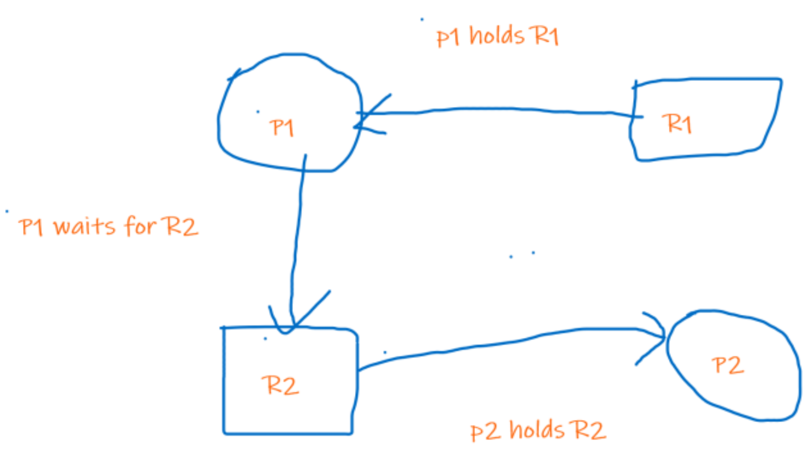
STARVATION -------process waits for indefinite time because it has low priority

DEADLOCK occurs because of 4 NECESSARY CONDITIONS

1. Mutual Exclusion = The resource is mutually exclusive !!!
   File opened for WRITING , Printer = examples of mutually exclusive resource
   File opened for Reading = example of shared resource
2. Hold and wait = hold a resource and wait for another resource
3. No Preemption = kernel is non preemptive , it does not take away resource forcefully
4. Circular Wait = P1 waits for P2 , P2 waits for P3 , P3 waits for P1

RAG ----------NON DEADLOCK SITUATION

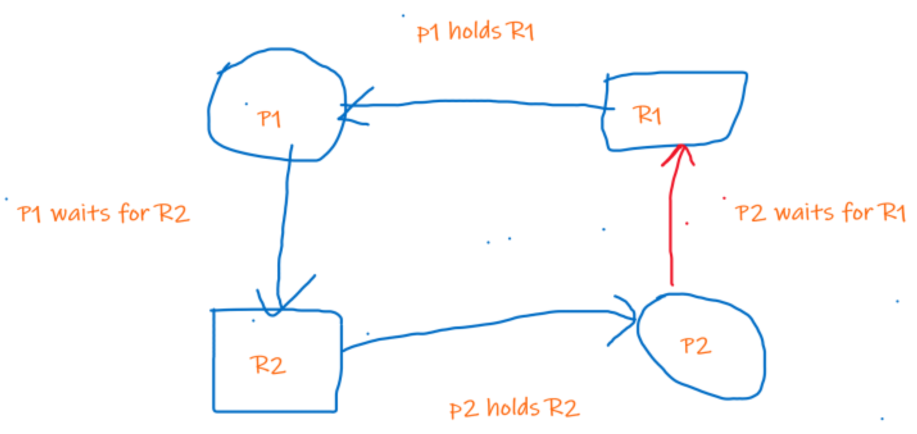
RAG = Resource Allocation Graph



P2 will execute and finish and release R2
then
P1 will use R2 and it will run and finish

CYCLE in RAG ----------

RAG = Resource Allocation Graph



CYCLE in RAG --- deadlock might occur

Solution to DEADLOCK

DON'T let DEADLOCK OCCUR ---------------

Deadlock Prevention = Break any one of the 4 necessary conditions

Deadlock Avoidance = Banker's algorithm = when Kernel allocates resources it first verifies if system will be in SAFE STATE or UNSAFE state after allocating the resource .

SAFE STATE then allocate else REJECT the request !!!!

SOLVE DEADLOCK AFTER it OCCUR s ------------------- Kill the low priority process in the CYCLE , release those resources and allocate to other process!!!!

IGNORE----------------LINUX based OS

SIGNALs -------------------------- interrupt !!!!

Signal is an integer

2 = SIGINT

SIGKILL , SIGSTOP = NON MASKABLE SIGNAL !!!

SIGTERM 15 === this signal is sent when we fire the kill pid , WE CAN MASK IT with our own SIGHANDLER
SIGINT 2 === this signal is sent when we do ctrl -c , We can MASK it
SIGKILL 9 ===========this is fired when kill -9 pid , WE CANNOT MASK it

We use signal() system call to MAP signal number with signal handler !!!!

Kill(pid,signo) --- system call can be used to invoke kill from a C program !!!

CLASSICAL SYNCHRONIZATION PROBLEM --------------------------

PRODUCER CONSUMER PROBLEM ------------ Bounded Buffer problem

One thread is a producer ----- it produces an item
|
| item
|

BOUNDED -BUFFER (finite size buffer)
|
|
|item

Another thread is consumer ----- consume the item

ACCESS to BOUNDED BUFFER is protected ---synchronized by using MUTEX semaphore

COMMUNICATION between producer and consumer is done using TWO COUNTING SEMAPHORES
FULL , EMPTY

If the buffer is FULL -------------Producer should wait
If buffer is EMPTY ------------ Consumer should wait

If producer adds an item in the empty buffer --------------- consumer should be signalled /notified
If consumer consumes an item from a full buffer --------- producer should be signalled /notified

---

mutex = 1 , full = 0 , empty = 10

Producer	Shared data	Consumer
	Arr [10] = bounded buffer	

While(1)
{
 	
sem_wait(empty) ----- decr empty
Produce item
Sem_wait(mutex)
Add item to buffer
		While(1)
		{
		
		sem_wait(full) ---decr full
		
		sem_wait(mutex)

Sem_post(mutex) sem_post(full) ----incr full }		Remove item from buffer sem_post(mutex) Consume item sem_post(empty) ---incr empty }

CLASSICAL DEADLOCK PROBLEM -- DINING PHILOSOPHER's PROBLEM
