Wednesday, October 27, 2021 11:09 AM

Synchronization ----------

Process synchronization
Thread synchronization

Data Sharing ----------

10000 + 100*10 = 11000
11000 - 200*10 = 9000

Final balance after the 2 threads have finished working MUST be 9000 rupees

If I run the program many times

I got twice final balance was 8900
Once final balance was 9200
10 times final balance was 9000

Deposit	Shared data	Withdraw
	Balance =~~10000~~ ,~~9800~~ ,10100	
temp = balance } 10000 temp = temp + 100 } 10100 Timer interrupt balance =temp		T = balance } 10000 T = t - 200 } 9800 Balance =t
	Final value = 10,100 }} CORRUPT value	

PROBLEM with DATA SHARING !!! }}}}} RACE CONDITION problem

Race condition MAY occur suddenly !! Not predictable .
It is not due to wrong calculation !!!
It may occur when an interrupt occurs and other thread starts using shared data before the interrupted thread has finished !!!

Solution to problem - RACE CONDITION !!!!

WE will write the CRITICAL SECTION (code that uses shared data) inside IF ELSE !!!

To achieve MUTUAL EXCLUSION of CRITICAL SECTION !!! At a time only one critical section shall execute

Deposit	Shared data	Withdraw
	Balance = 10000	
If semaphore is 0 , wait Else Semaphore -- (1 to 0)		If semaphore is 0 , wait Else Semaphore -- (1 to 0)
temp = balance }} 10000 temp = temp + 100 }} 10100 Timer interrupt balance =temp Semaphore ++ (0 to 1)	Int Semaphore = ~~1~~,~~0~~ ,~~1~~ ,~~0~~ ,1 Balance = ~~10100~~ Balance = 9900	T = balance }} 10100 T = t - 200 }} 9900 Balance =t Semaphore ++ (0 to 1)

Mutual exclusion of critical section to prevent race condition of shared data = process synchronization OR thread synchronization , THREAD SAFETY or PROCESS SAFETY !!!!!!

Semaphore = it is a special integer , that is maintained by the kernel !!!

Semaphore is already thread safe , it is already protected from race condition !!!
There is not interrupt between reading the value of semaphore and changing !!!!!
Semaphores can be accessed ONLY through 2 ----atomic functions (interrupts don't occur during these functions)

Wait () , P()	if sema == 0 , wait else sema ---
Signal () , V()	sema++

Deposit	Shared data	Withdraw
	Balance = 10000	
Wait(sema)		Wait(sema)
temp = balance }} 10000 temp = temp + 100 }} 10100 Timer interrupt balance =temp Signal(sema)	Int Sema = ~~1,0~~ ,~~1~~ ,~~0~~ ,1 Balance = ~~10100~~ Balance = 9900	T = balance }} 10100 T = t - 200 }} 9900 Balance =t Signal(sema)

2 types of semaphores ----------

1. Binary semaphore = MUTEX = used to protect other shared data = value is toggled between 0 to 1
2. Counting Semaphore = used itself as shared data (if u need int shared data that can be only ++,--) = value will range from 0 to n , n to 0

Operating System (Galvin Silberchatz)

Read the process synchronization
Producer Consumer Problem ---Bounded Buffer Problem

Prachi.godbole@gmail.com

Traditional problem used to explain synchronization ---------

Producer thread

Buffer				

Consumer Thread

Producer will add to the buffer with finite size (bounded buffer)
Consumer will remove from the buffer

Shared area = BUFFER //protect it using MUTEX

Full and empty are counting semaphores
If the buffer is full then producer should wait wait(full)
If the buffer is empty then consumer should wait wait(empty)

Starvation = process waits indefinitely for high priority processes to complete
Synchronization = if a process forgets to signal sema then other process may wait indefinitely

Deadlocks ----------
Process wait indefinitely for IO resources !!!

when process foreget to signal(sema)

4 necessary conditions that should occur so that DEADLOCK occurs

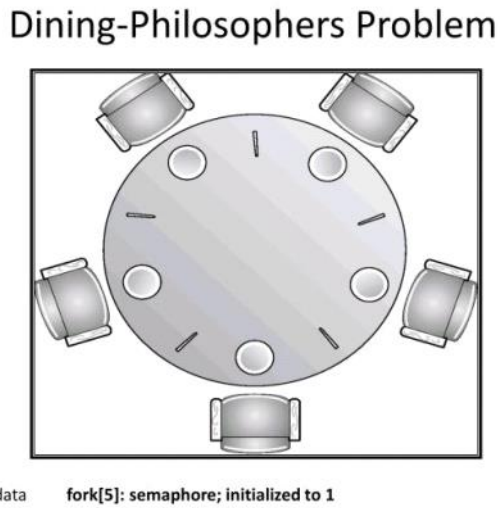
1. **Mutual exclusion**
   io resource may by shared or mutually exclusive

   3 people
   1 fan = can I use the same fan for 3 people **at a time** ?? SHARED resource
   Read only file---- at a time !!!!

   MUTUALLY EXCLUSIVE
   1 pen = can 3 people use the same pen at a time ?? NO …at a time sharing not possible, **one by one** ok!!!
   Printer ---one by one

2, Hold and wait = process holds a resource and waits for another resource
3, No preemption = kernel cannot forcefully take away a resource
4, Circular wait = p1 waits for a resource held by p2 and p2 waits for a resource held by p3 and p3 waits for a resource held by p1

Dining philosophers problem ---------



As all 4 conditions are satisfied the philosopher's will wait indefinitely = DEADLOCK !!!

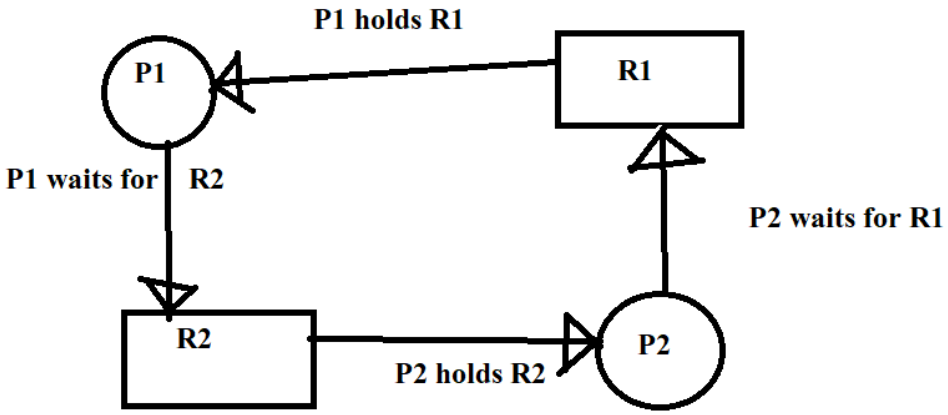
How to solve deadlock
Prevention ----- break any one of the 4 condition
Avoidance ------- allocate resources to the processes using BANKER's ALGORITHM

POST DEADLOCK ---------
Kill a low priority process in the deadlock group , this will release the resources held by the process

---

Linux based OS ---------- IGNORE (if too many processes waiting in deadlock then restart the systems)

Kernel maintains RAG to keep track of the deadlock
RAG = Resource Allocation Graph data structure



CYCLE IN RAG may lead to DEADLOCK

READ THE BOOK -----

Producer consumer problem for synchronization

Banker's algorithm for DEADLOCK avoidance

See u in the lab 2.45 !!!!
