COMP90050
Advanced Database Systems

Winter Semester, 2023

Lecturer: Farhana Choudhury (PhD)

Week 3 part 3

Concurrency Problems

Shared counter = 100;

Task1/Trans/Process/Thread	Task2/Trans/Process/Thread
counter = counter +10;	counter = counter +30;

Task1 and Task2 are running concurrently. What are the possible values of counter after the end of Task1 and Task2?

a) counter == 110;

b) counter == 130;

c) counter == 140;

For correct execution we need to impose exclusive access to the shared variable counter by Task1 and Task2.

Concurrency Problems

Shared counter = 100;

Task1/Trans/Process/Thread	Task2/Trans/Process/Thread
counter = counter +10;	counter = counter +30;

Task1 and Task2 run concurrently. What are the possible values of counter after the end of Task1 and Task2?

Note: == means equals.

a) counter == 110	b) counter == 130	c) counter == 140;
Sequence of actions	Sequence of actions	Sequence of actions
T1: Reads counter == 100	T1: Reads counter == 100	T1: Reads counter == 100
T2: Reads counter == 100	T2: Reads counter == 100	T1: Writes counter == 100+10
T2: Writes counter == 100+30	T1: Writes counter == 100+10	T2: Reads counter == 110
T1: Writes counter == 100+10	T2: Writes counter == 100+30	T2: Writes counter == 110+30

Time

Concurrency Control

- To resolve conflicts
- To preserve database consistency

Different ways for concurrency control

– **Dekker's algorithm (using code)** - needs almost no hardware support, but the code is very complicated to implement for more than two transactions/processes

– **OS supported primitives (through interruption call)** - expensive, independent of number of processes, machine independent

– **Spin locks (using atomic lock/unlock instructions)** – most commonly used

Concurrency control: Implementation of exclusive access

Dekker's algorithm

int c1, c2, turn = 1; /* global variable*/

T1	T2
{ some code T1}	{ some code T2}
/* T1 wants exclusive access to the resource and we assume initially c1 = = 0*/	/* T2 wants exclusive access to the resource and we assume initially c2 = = 0 */
c1 = 1; turn = 2;	c2 = 1; turn = 1;
repeat until { c2 == 0 or turn == 1}	repeat until { c1 == 0 or turn == 2}
/* Start of exclusive access to the shared resource **(successfully changed variables)** */	/* Start of exclusive access to the shared resource */
use the resource	use the resource
counter = counter+1;	counter = counter+1;
/* release the resource */	/* release the resource */
c1 = 0;	c2 = 0 ;
{some other code of T1}	{some other code of T2}

Implementation of exclusive access

- Dekker's algorithm
  - needs almost no hardware support although it needs atomic reads and writes to main memory, That is exclusive access of one time cycle of memory access time!
  - **the code is very complicated to implement if more than two transactions/process are involved**
  - harder to understand the algorithm for more than two process
  - takes lot of storage space
  - uses busy waiting
  - efficient if the lock contention (that is frequency of access to the locks) is low

Implementation of exclusive access

- OS supported primitives such as lock and unlock
  - through an interrupt call, the lock request is passed to the OS
  - need no special hardware
  - are very expensive (several hundreds to thousands of instructions need to be executed to save context of the requesting process.)
  - do not use busy waiting and therefore more effective

- All modern processors do support some form of spin locks.

Implementation of exclusive access

Spin Locks

Executed using atomic machine instructions such as test and set or swap

- need hardware support – should be able to lock bus (communication channel between CPU and memory + any other devices) for two memory cycles (one for reading and one for writing). During this time no other devices' access is allowed to this memory location.
- use busy waiting
- algorithm does not depend on number of processes
- are very efficient for low lock contentions – all DB systems use them

Implementation of Atomic operations: test and set

```
testAndSet(int *lock)
{ /* the following is executed atomically, memory bus can be locked for up
    to two cycles (one for read and for writing*/
        if (*lock == 1){ * lock = 0; return (true)}
            else return (false);
    }
```

<u>Using test and set in spin lock for exclusive access</u>

int lock = 1; % initial value

T1

```
/*acquire lock*/
while (!testAndSet( &lock );
    /*Xlock granted*/
//exclusive access for T1;
counter  = counter+1;
/* release lock*/
lock = 1;
```

T2

```
/*acquire lock*/
while (!testAndSet( &lock );
    /*Xlock granted*/
//exclusive access for T2;
counter  = counter+1;
/* release lock*/
lock = 1;
```

Implementation of Atomic operation: compare and swap

The following code may have lost values

```
temp = counter +1; //unsafe to increment a shared counter

counter = temp;  //this assignment may suffer a lost update
```

a) counter == 110
Sequence of actions
T1: Reads counter == 100
T2: Reads counter == 100
T2: Writes counter == 100+30
T1: Writes counter == 100+10

Using compare and swap in spin lock for exclusive access

Implementation of Atomic operation: compare and swap

The following code may have lost values

```
temp = counter +1; //unsafe to increment a shared counter

counter = temp;  //this assignment may suffer a lost update
```

Instead, we can use the atomic operation of compare and swap instruction

```
boolean cs(int *cell, int *old, int *new)

{/* the following is executed atomically*/

if (*cell == *old)      { *cell = *new; return TRUE;}

else { *old = *cell; return FALSE;}

}
```

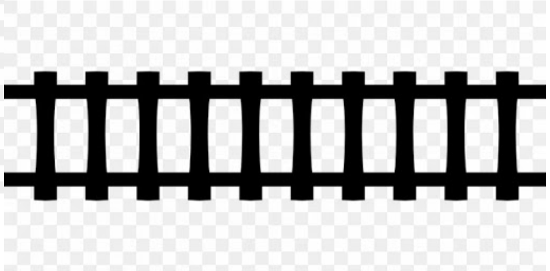
```
   temp = counter;
   do
           new = temp+1;
   while(!cs(&counter,&temp,&new));
```

<u>Using compare and swap in spin lock for exclusive access</u>

Semaphores

Semaphores derive from the corresponding mechanism used for trains: a train may proceed through a section of track only if the semaphore is clear. Once the train passes, the semaphore is set until the train exits that section of track.

Try to Get(track), wait if track not clear

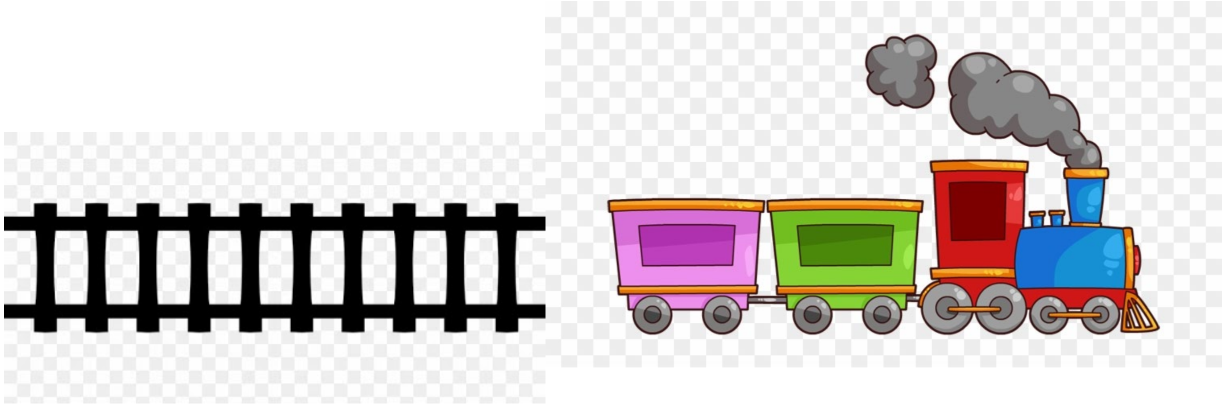


If Get(track) successful, use it (no other train will be able to use it now)

Semaphores

Semaphores derive from the corresponding mechanism used for trains: a train may proceed through a section of track only if the semaphore is clear. Once the train passes, the semaphore is set until the train exits that section of track.

Release(track) after using (so that others can use it)



Semaphores

Computer semaphores have a get() routine that acquires the semaphore (perhaps waiting until it is free) and a dual give() routine that returns the semaphore to the free state, perhaps signalling (waking up) a waiting process.

Semaphores are very simple locks; indeed, they are used to implement general-purpose locks.

Implementation of Exclusive mode Semaphore

Pointer to a queue of processes

If the semaphore is busy but there are no waiters, the pointer is the address of the process that owns the semaphore.

If some processes are waiting, the semaphore points to a linked list of waiting processes. The process owning the semaphore is at the end of this list.

After usage, the owner process wakes up the oldest process in the queue (first in, first out scheduler)

Implementation of Exclusive mode Semaphore

type long PID

type struct Process{

PID pid; /*process ID*/

PCB * sem_wait; /* waiting process are put in the queue */

} PCB;

PID MyPID (void); /* returns the caller's process ID */

PCB * MyPCB(void)

/* returns pointer to caller's process descriptor */

void wait (void); /* suspends calling process */

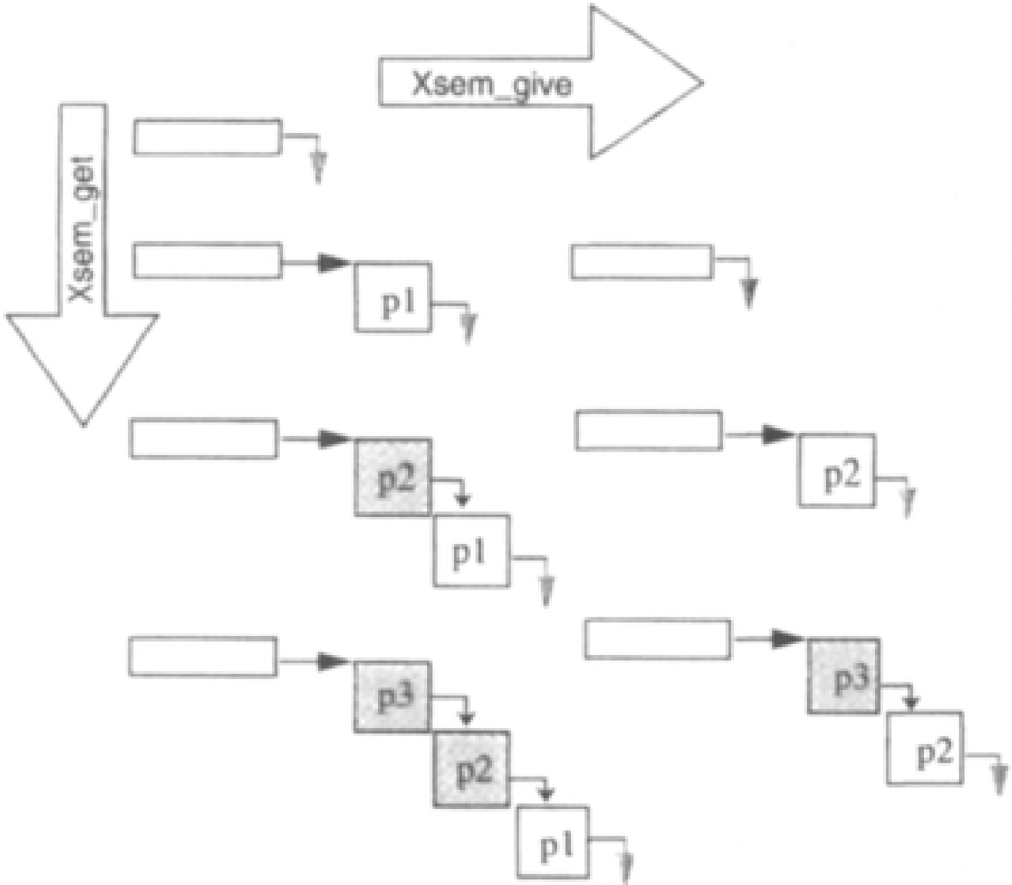
void wakeup(PCB * him) wakes him.

Implementation of exclusive lock semaphore operations

```
type struct Process{
        PID pid;
        PCB * sem_wait;
}PCB;
typedef PCB *Xsemaphore
Void initialise( Xsemaphore *sem)
{*sem = NULL; return;}
```

```
void lock(Xsemaphore *sem){
  PCB * new = myPCB();
  PCB * old   = NULL;
 /*put the process in the semWait queue*/
  do{
     new ->semWait = old;
  } while(! cs(sem, &old, &new));
/*if queue not null then must wait for a
wakeup from the semaphore owner*/
  if (old != NULL) wait() % OS call
  return;
}
```

```
void unlock(Xsemphore *sem){
  PCB * new = NULL;
  PCB * old   = myPCB();
   if(cs(sem, &old, &new)) return;
    while(old->semWait !=myPCB()){
          old = old->semWait
    }
    old->semWait = NULL;
    /*wake up the oldest process (first in,
    first out scheduler)*/
    wakeup(old); % OS Call
}
```



The common case: semaphore is free. Process p1 gets and frees the semaphore.

The common wait case: p2 waits for p1. Process p1 wakes up process p2 when p1 releases the semaphore.

An unusual case, a long wait queue has formed. In this case process p1 scans to the end of the queue and wakes up process p2. Such long queues may need the convoy wakeup logic.

Convoy avoiding semaphore

The previous implementation may result a long list of waiting processes – called convoy

To avoid convoys, a process may simply free the semaphore (set the queue to null) and then wake up every process in the list after usage.

In that case, each of those processes will have to re-execute the routine for acquiring semaphore.

Convoy avoiding semaphore

```
void lock(Xsemaphore *sem){
   PCB * new = myPCB();
   PCB * old   = NULL;
   do{
       new ->semWait = old;
    }while( ! cs(sem, &old, &new));
   if(old != NULL) {
              wait();  lock(sem);
    }
   return;
}
```

```
void unlock(Xsemphore *sem){
     PCB * new = NULL;
     PCB * old   = myPCB()
     while(!cs(sem, &old, &new));
     while(old->semWait !=
     myPCB()){
              new = old->semWait;
              old->sem_wait = NULL;
              wakeup(old);
              old = new;
       }
       return;
}
```

A quick summary

Concurrency problems – why we need concurrency control

Implementation of exclusive access –

- Dekker’s algo
- OS primitives
- **Spin locks -** Atomic operations to get and release locks

Semaphores - get lock, release lock, maintain queue of processes

Avoiding long queues in semaphores

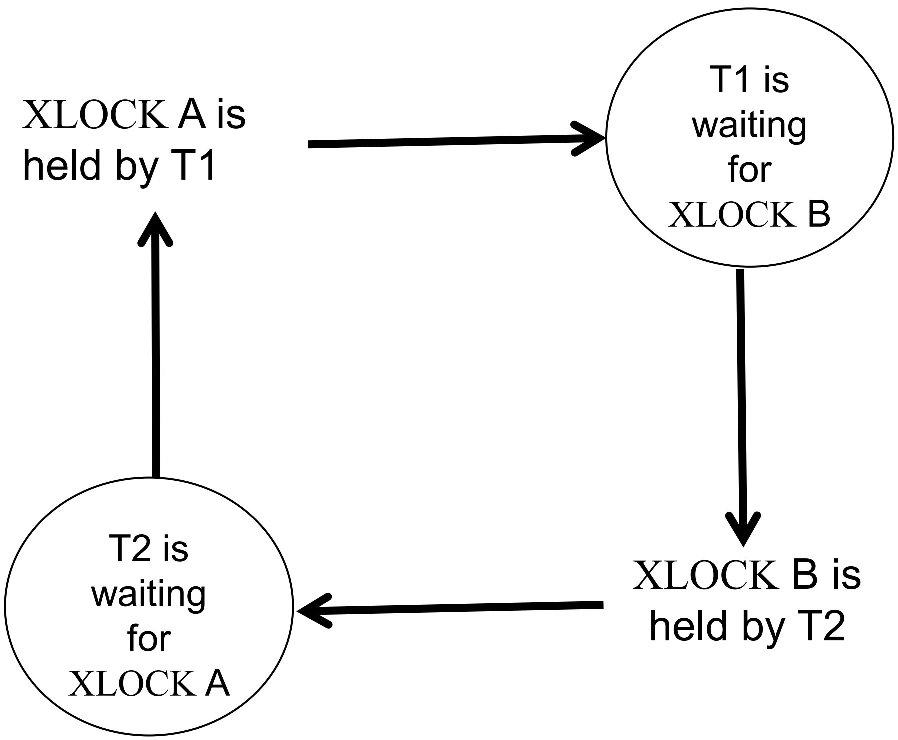
Deadlocks

In a deadlock, each process in the deadlock is waiting for another member to release the resources it wants.

A very simple Deadlock

T1	T2
Begin	Begin
XLOCK(A)	XLOCK(B)
Write to A	Write to B
XLOCK(B)	XLOCK(A)
Write B	Write A
Unlock (A)	Unlock (A)
Unlock (B)	Unlock (B)
end	end

Resource Dependency Graph



Deadlocks

Solutions:

- Have enough resources so that no waiting occurs – not practical
- Do not allow a process to wait, simply rollback after a certain time. This can create live locks which are worse than deadlocks.
- Linearly order the resources and request of resources should follow this order, i.e., a transaction after requesting i^{th} resource can request j^{th} resource if $j > i$ .This type of allocation guarantees no cyclic dependencies among the transactions.

Deadlocks

Pa: Holds resources at level ***i*** and request resource at level ***j*** which are held by Pb. $j > i$

Pq: Holds resources at level ***g*** and request resource at level ***l*** which are held by Pd. $l > g$

Pb: Holds resources at level ***j*** and request resource at level ***k*** which are held by Pc. $k > j$

Pc: Holds resources at level ***k*** and request resource at level ***l*** which are held by Pd $l > k$

Pd: Holds resources at level ***l*** and and is currently running.

$l > k > j > i$ ***and*** $l > g$.

We cannot have loops. The dependency graph can be a tree or a linear chain and hence cannot have cycles.

Deadlock avoidance/mitigation

- Pre-declare all necessary resources and allocate in a single request.
- Periodically check the resource dependency graph for cycles. If a cycle exists - rollback (i.e., terminate) one or more transaction to eliminate cycles (deadlocks). The chosen transactions should be cheap (e.g., they have not consumed too many resources).
- Allow waiting for a maximum time on a lock then force Rollback. Many successful systems (IBM, Tandem) have chosen this approach.

- Many distributed database systems maintain only local dependency graphs and use time outs for global deadlocks.

Deadlocks ...

Deadlocks are rare, however, they do occur and the database has to deal with them when they occur

What is the probability of a deadlock occurrence? - tutorial

Summary

Concurrency problems – why we need concurrency control

Implementation of exclusive access – Dekker’s algo, OS primitives, spin locks

Semaphores - get lock, release lock, maintain queue of processes

Atomic operations to get and release locks

Semaphore queue, avoiding long queues

Deadlocks