Process Synchronization and Deadlocks

Process Synchronization and Deadlocks – Slide Plan

- Why do you need synchronization? The Critical-Section Problem – Slides 3-9
- How do you synchronize? Synchronization Hardware, Locks, Mutexes and Semaphores – Slides 10-17
- Synchronization Examples – Slides 18-21
- Barrier synchronization – Slides 22-25
- Synchronization problems - Deadlocks – Slides 26-27

Why do you need Synchronization?

- Critical sections – register coherence, variable (queue) coherence, atomicity preservation, interrupt register coherence
- Solution – semas, variety of locks, interrupt masking
- Semantic (co-ordination) coherence – matrix multiplication, barrier wait
- Sync going awry – deadlocks, sema priority inversion, basic application-level logical solutions, OS-level solutions – sema priority inversion (avoidance), watchdog (recovery)

Why do you need Synchronization?

- Fundamentally, the need for synchronization of multiple threads/processes (we will use ‘threads’ here on because that is simpler) is because serial semantics are expected even in multithreaded scenarios, i.e. we expect the same results from concurrent execution that we would have got in completely serialized execution scenarios

- Basically, what we are expecting is serial execution results in a shorter time

Why do you need Synchronization?

- Let us look at a basic concurrent/multithread execution scenario:

Expected results after a run each of T1 and T2, assuming Counter is initially 0: Counter =2

Why do you need Synchronization?(contd.)

- If you look at this closer, you will notice that a load of the variable ‘counter’ from memory (value 0 in this case) will need to be done into a register before an increment is done.
- Assuming that T1 got swapped out at this point by say, a higher priority T2, the register would be backed up on T1’s stack and restored at some point when T1 get swapped back in.
- Meanwhile in this T1 swap-out time window, assume T2 that just got swapped in, also went in and loaded ‘0’ (which was what the value of the variable would still be because T1’s incrementing has not been commited yet to memory). The ‘0’ is incremented to 1 and commited to memory.

Why do you need Synchronization? (contd.)

- When T1 is eventually scheduled back in, it will also increment 0 (in its post-context switch restored register) to 1 and push it back to memory.
- Thus, we have missed a write to ‘counter’ and haven’t got the serial semantics that we required.
- This is the case because, the threads didn’t get mutually exclusive increment access to ‘counter’, the register providing a space for the lack of mutual exclusion to exploit.
- In other words, the register contents across the context switch of T1 have effectively become stale, you could call it a lack of register coherence. A seemingly indivisible, atomic increment at assembly instruction level, proving not to be so.

Why do you need Synchronization? (contd.)

- Let's look at another loss of atomicity, this time at the high-level language level, in an enqueuing/dequeueing scenario.
- Enqueueing: Prev = Tail; Tail = new (Node); Tail → next = Prev
- Need for mutual exclusion is because if a second process did the same as above for a different node, depending on which enqueueing (Tail = new (Node)) happened last, that would overwrite the first, if Prev=Tail and Tail = new (Node) happened in an overlapped manner.
- For dequeue: Head = Head.next involves a load, a change and a store, which needs to be mutually exclusive, else you might miss a dequeue.
- This is a lack of atomicity as three high-level language steps have lost their sequential coherence.

Why do you need Synchronization? (contd.)

- Let’s look at a case of nested interrupts.
- There is a stage in early interrupt/exception processing where some special purpose registers that support interrupts and interrupt returns need to be saved on the interrupt/exception stack. If interrupts are not masked in this time window, a nested interrupt can cause return from interrupts to be incoherent.
- Therefore, interrupts are masked by processor architecture in this window, and need to be re-enabled by interrupt handler code to allow for nested interrupts after this window.

How do you Synchronize?

- There are a few processor instructions that implement atomicity at the hardware level (at the lowest register-bus level of coherence, atomicity is after all a hardware issue).
- Operating system-level solutions like a variety of locks, semaphores, mutexes, condition variables are built on top of these processor-level primitives, to safeguard coherence and help obtain expected serial results.
- And, in some extreme cases like the interrupt handling case, masking of interrupts is done to achieve coherence. And, in the multiprocessor context, masking of interrupts is replaced by spinlocks.

Locks, Mutexes and Semaphores

Locks, Mutexes and Semaphores are based on an atomic processor instruction or an in-effect atomic pair of instructions that have the following semantics:

- Test-and-set-lock – Atomically, test a memory location for a value (say, 1) and set it to 1. The instruction "returns" True by setting a flag/register, if the old value before the set, was 0. If it was already 1, then it would return False.

  - ✓ On x86 processors, similar functionality is provided by the cmpxchg with lock prefix instruction pair (this is more compare-and-swap, but broad idea remains the same even if with a slightly different implementation).

  - ✓ On RISC processors like the PowerPC, it is the ll(load linked), sc (store conditional) pair that provide the same functionality

Basic building blocks of Mutual Exclusion

With cmpxchg:

Load eax, address /*old value*/

Mov eax, ebx

Increment ebx

Lock, Cmpxchg (ebx, address)

/*if address still contains old eax,

move current ebx there*/

If cmpxchg returns success, go forward, else loop back to load eax point & try again

Basic building blocks of Mutual Exclusion (Contd.)

With ll-sc:

P1: Loop: Load-linked (reg, addr);
reg=reg+1;
If Store-conditional (addr, reg) = False, Loop.
Else (Exit loop)

P2: Loop: Load-linked (reg, addr);
reg=reg+1;
If Store-conditional (addr, reg) = False, Loop.
Else (Exit loop)

Locks

There are a variety of locks that build on the basic atomic primitives described in the previous slides, here's a basic lock of the busy-waiting variety using ll-sc:

```
void lock(atomic32_p lock_p
        {
                int32 old_val;
                repeat {
                        old_val = lwarx(lock_p);
                } until(old_val == 0 && stwcx (lock_p, 1));
                isync();
                return;
        }
```

Locks (contd.)

```
void unlock(atomic32_p lock_p)
        {
                int32 old_val;
                sync();
                lock_p->value = 0;
        return;
        }
```

Note: The isync, sync is in order to ensure sequential consistency in case of processors that have more relaxed consistency models

Locks, Mutexes and Semaphores(contd.)

- The implementation of the basic lock on the previous slides involved a busy-waiting loop to get the lock.
- Although useful as proof-of-concept, that's too time consuming to be of interest in practical OS-level implementations.
- Therefore, in practical implementations, instead of looping/busy-waiting in attempts to acquire locks, mutexes, or semaphores, you block the task and as part of the releases, you unblock a task that is waiting(for instance, at the head of the wait queue) on the lock, mutex or semaphore.

Locks, Mutexes and Semaphores(contd.)

- The distinctions between locks, mutexes and semaphores are at the OS-level – at the very lowest level, they all use the basic mutual exclusion implementing processor instructions.

- One crucial difference between mutexes and semaphores at the OS-level is that a mutex can only be released by its owner, whereas a semaphore can be released by threads that don't own it. Hence, semaphores can be used purely as a signalling mechanism say, from kernel level to a user-level thread.

Posix Mutexes - example

```
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>

pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* trythis(void *arg)
{
   pthread_mutex_lock(&lock);
   unsigned long i = 0;
   counter += 1;
   printf("\n Job %d has started\n", counter);

   for(i=0; i<(0xFFFFFFFF);i++);
   printf("\n Job %d has finished\n", counter);
   pthread_mutex_unlock(&lock);
   return NULL;
}
```

Posix Mutexes – example (contd.)

```
int main(void)
{
   int i = 0;
   int error;

   if (pthread_mutex_init(&lock, NULL) != 0)
   {
      printf("\n mutex init has failed\n");
      return 1;
   }

   while(i < 2)
   {
      error = pthread_create(&(tid[i]), NULL, &trythis, NULL);
      if (error != 0)
         printf("\nThread can't be created :[%s]", strerror(error));
      i++;
   }
   pthread_join(tid[0], NULL);
   pthread_join(tid[1], NULL);
   pthread_mutex_destroy(&lock);
   return 0;
}
```

Q: What if mutexes weren't used?

Posix Semaphores - example

```
// C program to demonstrate working of Semaphores
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t mutex;

void* thread(void* arg)
{
   //wait
   sem_wait(&mutex);
   printf("\nEntered..\n");

   //critical section
   sleep(4);

   //signal
   printf("\nJust Exiting...\n");
   sem_post(&mutex);
}


int main()
{
   sem_init(&mutex, 0, 1);
   pthread_t t1,t2;
   pthread_create(&t1,NULL,thread,NULL);
   sleep(2);
   pthread_create(&t2,NULL,thread,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   sem_destroy(&mutex);
   return 0;
}
```

Posix Semaphores – example (contd.)

```
int main()
{
   sem_init(&mutex, 0, 1);
   pthread_t t1,t2;
   pthread_create(&t1,NULL,thread,NULL);
   sleep(2);
   pthread_create(&t2,NULL,thread,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   sem_destroy(&mutex);
   return 0;
}
```

Use gcc sema.c -lpthread –lrt to compile

Questions: What is the expected output? What if there was no semaphore used?

Barrier Synchronization

A Simple Centralized Barrier

- **Shared counter maintains number of processes that have arrived**
  - **increment when arrive (lock), check until reaches numprocs**
  - **Problem?**

```
struct bar_type {int counter; struct lock_type lock;
                              int flag = 0;} bar_name;
BARRIER (bar_name, p) {
    LOCK(bar_name.lock);
    if (bar_name.counter == 0)
        bar_name.flag = 0;              /* reset flag if first to reach*/
    mycount = bar_name.counter++;       /* mycount is private */
    UNLOCK(bar_name.lock);
    if (mycount == p) {                 /* last to arrive */
        bar_name.counter = 0;           /* reset for next barrier */
        bar_name.flag = 1;              /* release waiters */
    }
    else while (bar_name.flag == 0) {};     /* busy wait for release */
}
```

Barrier Synchronization (contd.)

- Increment of counter needs to be atomic
- Increment of counter and check needs to be atomic – why?
- What is the problem in the above? Set of the flag (which is the exit criterion for the busy wait) is done just before the exit, but since the reset of the flag is done on entry into the barrier a second time (a re-entry for the same process), if processes other than the last get swapped out in the middle of the exit criterion check loop and come back, there is a chance that they will get to see the newly reset value of flag. Thus, ensuring they never exit the barrier, resulting in a deadlock.

Barrier Synchronization (contd.)

- This can be fixed using a local sense variable that demarcates the first iteration of the barrier from the second, by introducing a local sense of which iteration you are in. The global view of this local sense is consistent except for the time where the new (second) entrant to the barrier resets it and the others are still in the first barrier. Once everyone enters the barrier a second time, there is consistency once more.

- Any option other than thread-local sense variable? Yes, a separate counter, but that makes things clumsy, as you need a semaphore to protect this, etc.

Barrier with Sense Reversal

```
bar_name.flag = 1;
thread_local local_sense  = 1;
BARRIER (bar_name, p) {

        local_sense = !(local_sense); /* toggle private sense variable */
        LOCK(bar_name.lock);
        bar_name.counter++;
        if (bar_name.counter == p)
                {UNLOCK(bar_name.lock);
                bar_name.counter = 0;
                bar_name.flag = local_sense; }/* release waiters*/
        else
                { UNLOCK(bar_name.lock);
                while (bar_name.flag != local_sense) {}; }
        }
```

Problems when Synchronization is done in a wrong way

The most common problems faced when using lock-based synchronization methods are:

- Starvation or a skewed rate of progress among the tasks that are being synchronized due to a lack of guaranteed fairness in the lock acquire primitive
- Deadlocks due to either wrong order followed in acquire and release of locks in case of nested locks or issues like priority inversion
- Lack of proper classification of synchronization scenarios – there might be cases where for instance, you need to identify a scenario as a frequent reader, infrequent writer case and are doing over-serialization of readers.

Solutions to problems of poor Synchronization

- Solutions range from problem avoidance by using wait or lock free data structures to solve synchronization problems or problem detection and recovery by OS mechanisms like watchdog tasks

- Using multiple reader-single writer type locks like read-copy-update (RCU) primitives that are used in the Linux kernel

Thank You!