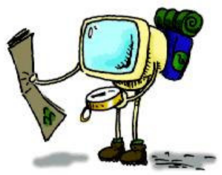
Programming Languages

Tevfik Koşar

Lecture - XXVI
April 27th, 2006

Roadmap

- Shared Memory
- Synchronization
  - Spin Locks
  - Barriers
  - Semaphores
  - Monitors

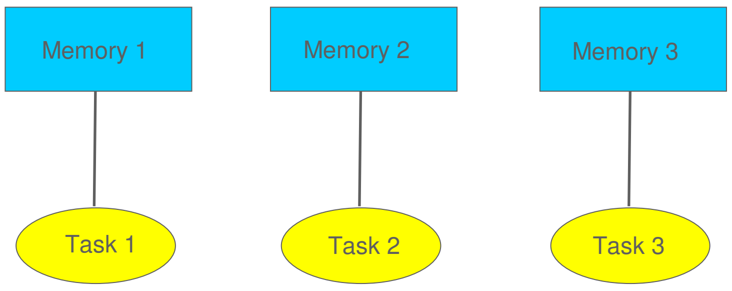


Memory Architectures

- Distributed Memory
- Shared Memory
- Distributed Shared Memory

3

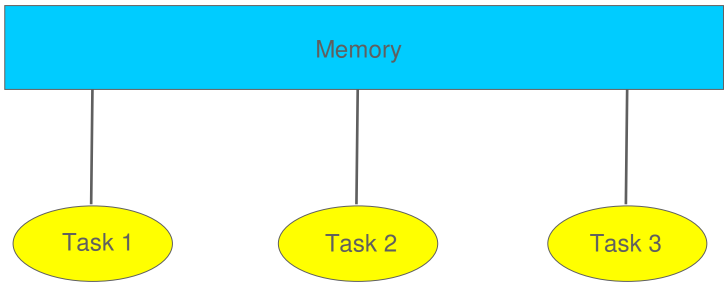
Distributed Memory



- Each process/task has access to its own memory

4

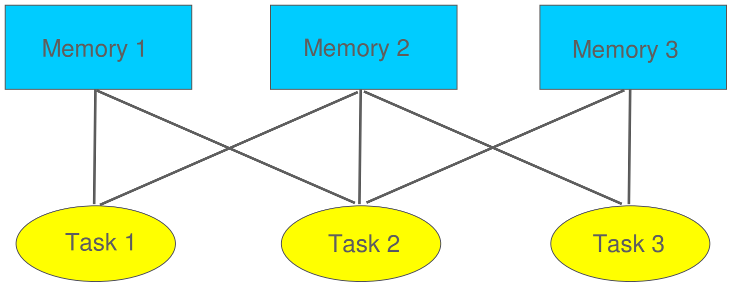
Shared Memory



- Each process/task has access to the same memory

5

Distributed Shared Memory



- Memory is physically distributed but conceptually shared

6

Synchronization

- Major challenge for shared memory and distributed shared memory architectures
- Two forms:
  - Mutual exclusion: ensures that only one process/thread is executing a critical section at any time
    - Never read or write shared data while it's being modified by another process/thread
  - Conditional synchronization: ensures that a process/thread does not proceed until some specific condition holds
    - eg. until a given variable has a given value
- More synchronization → Less parallelism
  - Do not do synchronization unless critical

7

Spin Locks

- Lock: mechanism for enforcing limits on access to a resource/shared data in a concurrent environment
- Spin Lock: A lock where a thread/process waits in a loop ("spins") repeatedly checking until the lock becomes available
- As the thread remains active but isn't performing a useful task, the use of such a lock is called busy waiting.
- Once acquired, spin locks will usually be held until they are released
  - in some implementations they may be automatically released if the thread blocks (aka "goes to sleep")

8

Spin Locks

```
Typedef bool lock;

bool test_and_set (lock *l){                //needs to be an atomic HW
    instruction!!
    if ( *l == false){
            *l = true;
            return true;
    }
    else return false;
}

void acquire_lock (lock *l) {
    while !test_and_set (l)  {} /* wait */;
}

void release_lock (lock *l) {
    *l = false;
}
```

Spin Locks

- Spin locks are efficient if threads/processes are only likely to be blocked for a short period of time
  - they avoid the overhead of operating system process re-scheduling
  - often used within operating system kernels
- Spin locks are wasteful if the lock is held for a long period of time
  - they do not allow other threads/processes to run while a thread/process is waiting for the lock to be released ➔ contention
  - always inefficient if used on systems with only one processor, as no other thread would be able to make progress while the waiting thread/process spun

Barriers

- A point in the control flow where each thread/process must stop until all other processes/threads reach this point
- Require consensus among all threads/processors
- Fundamental to data-parallel computing

11

Barriers

- set a global counter = n
  where n is # of threads/processors
- Each thread/process reaching its barrier
  decrements counter by 1, and stops
  - atomic fetch_and_decrement
  - loop until counter == 0
- If counter becomes 0, all threads continue execution

12

Semaphores

- protected variables
- classical method to restrict access to shared resources
- invented by Edsger Disjkstra and first used in “THE” operating system, 1968

- basically a counter with two associated operations:
  - P (Wait): decrement counter and wait until positive
    - *Probeer (try)*
  - V (Signal): increment counter and wake up a waiting thread
    - *Verhoog (increase)*

13

Semaphores

- The value of a semaphore is the number of units of the resource which are free.
- If there is only one resource, a "binary semaphore" with values 0 or 1 is used.
- The *P* operation busy-waits (or maybe sleeps) until a resource is available, whereupon it immediately claims one.
- *V* is the inverse; it simply makes a resource available again after the process has finished using it.
- *Init* is only used to initialize the semaphore before any requests are made.
- The *P* and *V* operations must be atomic, which means that no process may ever be preempted in the middle of one of those operations to run another operation on the same semaphore.

14

Semaphores

```
P(Semaphore s)
{
    await s > 0, then s := s-1; /* must be atomic once s > 0 is detected */
 }

V(Semaphore s)
{
    s := s+1; /* must be atomic */
}

Init(Semaphore s, Integer v)
{
    s := v;
}
```

15

Dining Philosophers Problem

• Five philosophers spend their time eating and thinking.

• They are sitting in front of a round table with spaghetti served.

•There are five plates at the table and five forks set between the plates.

• Eating the spaghetti requires the use of two forks which the philosophers pick up one at a time.

•possibility of deadlock: if every philosopher holds a left fork and waits perpetually for a right fork (or vice versa) ➔ may lead to starvation

16

Disjkstra's Solution

For each philosopher i:

```
1. Think
2. Wait(room_semaphore) //max value = 4
3. Wait(left_fork_i)
4. Wait(right_fork_i)
5. Eat spaghetti
6. Signal(left_fork_i)
7. Signal(right_fork_i)
8. Signal(room_semaphore)
9. Repeat 1.
```

* Only four philosophers would be able to enter the room simultaneously and no deadlock can occur.

Chandy/Misra Solution (1984)

- For every pair of philosophers contending for a resource, create a fork and give it to the philosopher with the lower ID. Each fork can either be *dirty* or *clean*. Initially, all forks are dirty.
- When a philosopher wants to use a set of resources (*i.e.*, eat), it must obtain the forks from its contending neighbors. For all such forks it does not have, it sends a request message.
- When a philosopher with a fork receives a request message, it keeps the fork if it is clean, but gives it up when it is dirty. If it sends the fork over, it cleans the fork before doing so.
- After a philosopher is done eating, all its forks become dirty. If another philosopher had previously requested one of the forks, it cleans the fork and sends it.

Sleeping Barber Problem

- Based upon a hypothetical barber shop with one barber, one barber chair, and a number of chairs for waiting customers
- When there are no customers, the barber sits in his chair and sleeps
- As soon as a customer arrives, he either awakens the barber or, if the barber is cutting someone else's hair, sits down in one of the vacant chairs
- If all of the chairs are occupied, the newly arrived customer simply leaves

19

Possible Problems

- The barber could end up waiting on a customer and a customer waiting on the barber, resulting in deadlock.
- The customers may not decide to approach the barber in an orderly manner, leading to process starvation as some customers never get the chance for a hair cut even though they have been waiting

20

Solution

- Use three semaphores: one for any waiting customers, one for the barber (to see if he is idle), and a mutex
- When a customer arrives, he attempts to acquire the mutex, and waits until he has succeeded.
- The customer then checks to see if there is an empty chair for him (either one in the waiting room or the barber chair), and if none of these are empty, leaves.
- Otherwise the customer takes a seat – thus reducing the number available (a critical section).
- The customer then signals the barber to awaken through his semaphore, and the mutex is released to allow other customers (or the barber) the ability to acquire it.
- If the barber is not free, the customer then waits. The barber sits in a perpetual waiting loop, being awakened by any waiting customers. Once he is awoken, he signals the waiting customers through their semaphore, allowing them to get their hair cut one at a time.

Implementation:

+ Semaphore Customers
+ Semaphore Barber
+ Semaphore accessSeats (mutex)
+ int NumberOfFreeSeats

The Barber(Thread):

```
while(true) //runs in an infitie loop
{
  Customers.p() //tries to acquire a customer - if none is available he's going to sleep
  accessSeats.p() //at this time he has been awaken -> want to modify the number of
     available seats
  NumberOfFreeSeats++ //one chair gets free
  Barber.v() // the barber is ready to cut
  accessSeats.v() //we don't need the lock on the chairs anymore //here the barber is
     cutting hair
}
```

The Customer(Thread):

```
while (notCut) //as long as the customer is not cut
{
 accessSeats.p() //tries to get access to the chairs
 if (NumberOfFreeSeats>0) { //if there are any free seats
   NumberOfFreeSeats -- //sitting down on a chair
   Customers.v() //notify the barber, who's waiting that there is a
   customer
   accessSeats.v() // don't need to lock the chairs anymore
   Barber.p() // now it's this customers turn, but wait if the barber is
   busy
   notCut = false
 } else // there are no free seats //tough luck
 accessSeats.v() //but don't forget to release the lock on the seats }
```

23

Semaphores

- inadequate in dealing with deadlocks
- do not protect the programmer from the easy mistakes of taking a semaphore that is already held by the same process, and forgetting to release a semaphore that has been taken
- mostly used in low level code, eg. operating systems
- the trend in programming language development, though, is towards more structured forms of synchronization, such as monitors and channels

24