Kumar Garapaty

CS 162 Notes for 2/9/2009

Last time talking about producers/consumers

Announcement: No discussion tomorrow – Nachos 7pm 306 soda

Readers and Writers Problem – standard problem, shared data – various processes reading and writing it – obtain decent results

- A shared database with readers and writers, writers can't overlap readers
- It is safe for any number of readers to access the database simultaneously, but each writer must have exclusive access.
    - Note write – read modify write
    - must use semaphores
    - Checking account
    - Writers are actually readers too

Constraints

- Scheduling
    - Writers can only proceed if there are no active readers or writers
    - Readers can proceed if there are no active or waiting writers – use semaphore oktoread
    - To keep track of who's r/w – state variables
    - AR – active, WR- waiting, AW-active WW – waiting, AW is always 0 or 1

Initialization:

```
OkToRead = 0; OkToWrite = 0; Mutex = 1,
```

System starts empty, variables equal 0

Can argue not the most efficient scheduling, scheduling: writers get preference

Reader Process:

```
        P(Mutex); - lock acces on shared variables
        if ((AW+WW) == 0) - if no writers in system, give permission to read, increments count of
reader
        {
                V(OkToRead);
                AR = AR + 1;
        }
        else WR = WR + 1;
        V(Mutex);
        P(OKToRead);
        --read the necessary data;
        P(mutex); - lock variable
        AR = Ar-1;- decrements active readers
        if (AR ==0 && WW > 0)
        {
                V(OKToWrite);
                AW = AW + 1;
                WW = WW – 1;
```

```
	}
	V(Mutex);
```

Writer Process:

```
	P(mutex); - locks things
	if ((AW + AR + WW)==0) – nothing in the system (do we need WW - no)
	{
		V(OKToWrite); - permission to write
		AW = AW + 1; - increment active write
	}
	else WW = WW + 1; - wait to write
	V(Mutex); - release mutex
	P(OKToWrite) – permission get through this – if not hangs there
	--write the necessary data
	P(Mutex); - locks
	AW = AW -1; - decrement active writer
	if (WW>0)
	{
		V(OKToWrite); - let one of them go
		AW = AW+1;
		WW = WW-1;
	}
	else while (WR > 0) – keep going till all waiting readers are done
	{
		V(OkToRead)
		AR + AR +1;
		WR = WR -1;
	}
	V(Mutex) – can exit
```

Another problem: Dining Philosophers Problem

Assume 5 philosophers (works for N). Out to dinner at Italian restaurant. Seated at circular table, with one fork between each pair of philosophers. Philosophers need 2 forks to eat.

Algorithm for getting forks so that they can eat.

Assume solution must be:

- Symmetric – all philosophers use same algorithm
- Can't number the philosophers as part of the solution – all even get forks (can refer to them in numbers)
- Efficient – more than one philosopher eats – as many to eat as possible
- No central control

Obvious solution

- a. pick up left fork
- b. pick up right fork; wait if necessary
- c. eat

Fails, due to immediate deadlock – each philosopher ends up with one fork, waiting for right fork – not an optimal solution

Second

- a. pick up left fork
- b.if available, pick up right fork, else,
    - (b1) put down left fork
    - b2 wait for right fork
    - b3 pick up right fork
    - b4 pcik up left fork; wait if necessary
- c.eat

Assuming all move in same speed, Fails in opposite order – now each is waiting for left fork.

Solution
N+1 philosophers
semaphore mutex (init 1)
    used for mutual exclusion
array H(0:N), init 'not hungry'
    values 'not hungry, hungry, eating'
semaphore array prisem(0:N), init (0) ("Private semaphore") - one for each philosopher, hangs up on if philosophers pick up forks
procedure test(me): - test myself or test neighbors – looks both ways and I am hungry then set state to eating
    if H((left) != eating and H(me) = hungry
    and H(right)- != eating do
    begin
    H(me) = eating
    V(prisem(me))
    end

cycle begin
    think(philosopher me) – do when they are not eating
    P(mutex) – hungry, locks state
    H(me):= hungry – take forks
    test(me) – looks to see if forks available, sets state to eating, permission to pick up forks
    V(mutex)
    P(prisem(me)) – forks are available, pick them up, only go to it when forks are availble
    eat
    P(mutex) – locks shared state
    H(me):=not hungry – sets state hungry – put down forks
    test(left) – test others
    test(right)
    V(mutex)- unlock state
end

mutex – none of them can change state
Solution is free of deadlock, but permits unbounded delay
It does not prevent starvation – neighbors are hungry and you are hungry sometimes – forks wont be put down both times
Private semaphore – used to control the progress of each process, and a common semaphore is used to allow for unambiguous inspection and modification of common state variables.

Threads – 4.1 – 4.4 - text

Thread – lightweight process, is a type of process

- Thread has its own pc, register set values, and stack
- thread shares with 1 or more threads its code, data, and OS resources such as open files with other threads – normal heavy process – has only 1
- Task consists of the set of threads sharing code, data, etc, a task with one thread is an ordinary (heavy weight) process)
- Switching between threads is much lower overhead than switching between separate processes. - only need to reload pc and registers. Only need to reload user registers, not change entire PCB (e.g. acc info, etc). D
    - In some cases, thread switching can be done by code in user-level library so no OS call is required. This is much lower overhead
        - Note that if thread switching is done by user, then OS doesn't know. Therefore, if one thread is blocked by OS all are blocked – process creates its own threads – user state. Also OS will allocate time per task, even though it may have many threads.
    - A thread can create child threads of its own.
    - Note that since memory is shared, there is low overhead sharing, but not protection – one thread writes something all threads see it, one thread messesup – crashes
- Why use threads
    - On uniprocessor, may provide more convenient model for programming normal sequential program (Does not inherently provide higher efficiency).
    - On shared memory Multiprocessor, may provide parallelism, since diff. Threads can run on different processors in parallel.
        - Note that OS must do scheduling for multiprocessors
- Lower overhead (task switching, memory sharing) than separate parallel (heavyweight processes.

- *Process Syncronization with Condition Variables*
- Processor or thread can cooperate using wait and signal along with condition variables. -
- The operation x.wait means that the process invoking it waits until some other process invokes x.signal. - do a wait on x then somebody releases u when someone does x.signal
- The x.signal operation resumes exactly one suspended process. If no process is suspended, then x.signal has no effect. - similar to v operation – v increment, signal is lost.
- x.signal and x.wait are used to control synchronization with Monitors, which is a special type of critical region. Only one process can be executing in a monitor at a time - idea monitor is a chunk of code and only one can process in a monitor, a mutex, execute , leave

Read paper on monitors

- There is one binary semaphore associated with each monitor, mut exclusion is implicit: P on entry to any routine, V on exit.
- Monitors are a higher-level concept than P and V. They are easier and safer to use.
- Monitors are a synchornization mechanism combining threee features:
    - Shared data
    - Operations on the data
    - Synchronization,scheduling

They are especially convenient for synchonization involving lots of state.

- Monitors need more facilities than just mutual exclusion, Need some way to wait
    - Busy-wait inside monitor?
    - Put process to sleep inside monitor?

- *Condition variables:* things to wait on – makes sense when you go through it
    - Wait(condition): release monitor lock, put process to sleep. When process is allowed to wake up again, re-acquire monitor lock immediately
    - Signal (condition): wake up (FIFO) , o/w do nothing
    - Broadcast (condition): wake up all

Several variations on wait/signal mechanism.
On signal, signaller keeps monitor lock
Once on wait queue, check again and prepared to sleep again.

Four procedures: checkRead, checkWrite, doneRead, doneWrite, conditions OKToRead, OKToWrite – This is all part of one monitor.