Theron Ji
Wednesday 2/11

Monitors

6.7 in text & Hoare article

Readers and writers problem redone with monitors:

- Monitored procedures: checkRead, checkWrite, doneRead, doneWrite
- Conditions: OKToRead, OKToWrite
- AW = active writers, WW = waiting writers, AR = active readers, WR = waiting readers

checkRead():

```
if ((AW+WW))>0)                 // If there are people writing
        WR = WR + 1             // Put yourself on waiting list
        Wait(OKToRead)          // Wait until it's OK
        WR = WR -1              // Remove yourself from waiting list
AR=AR+1                         // Put yourself on active reading list
Read                            // Read!
```

doneRead():

```
AR=AR-1                         // Remove yourself from active reading list
if (AR==0 & WW>0) signal(OKToWrite)   // If there are no more readers, let writers go
```

checkWrite():

```
while((AW+AR)>0)                // If there is someone current writing/reading
        WW = WW+1               // Add yourself to waiting list
        Wait(OKToWrite)         // Wait until it's OK to write
        WW = WW-1               // Remove yourself from waiting list
AW = AW + 1                     // Add yourself to active list
WRITE                           // Write!
```

doneWrite():

```
AW = AW – 1                     // Remove yourself from active list
if (WW>0) signal(OKToWrite)     // If there are more writers, let them go first
else broadcast(OKToRead)        // Else let all the readers go
```

- Very similar to P and V with semaphores

Producers and Consumers Problem w/ Monitors (from Hoare):

```
bounded buffer: monitor

            begin buffer: array 0..N-1 of portion

                last pointer:0..N-1;

                count:0..N;
```

```
        nonempty, nonfull: condition;

    procedure append(x; portion);

        begin if count==N then nonfull.wait;

        buffer[lastpointer] =x;

        last pointer = (lastpointer + 1) mod N

        count = count+1;

        nonempty.signal

        end append;

    procedure remove (result x; portion);

        begin if count==0 then nonempty.wait;

        x=buffer[(lastpointer - count) mod N];

        count=count-1;

        nonfull.signal;

        end remove

    count=0;

    lastpointer=0;

    end bounded buffer
```

Disk Head Scheduler:

- Like an elevator scheduler
- Sorted by levels

Terminology:

1. procedure request – called before issuing request to move head to disk
2. procedure release – call after cylinder is finished
3. headpos – current location of head
4. busy – whether disk is busy
5. sweep – direction of head movement, up or down

Variables:

1. *diskhead*: Monitor
2. *headpos*: Cylinder
3. *direction*: up/down
4. *busy*: Boolean
5. *upsweep/downsweep*: condition

```
procedure request(dest: cylinder);
   begin if busy then

   [if {((headpos < dest) or [headpos == dest & direction==up])
   then upsweep.wait(dest)
   else downsweep.wait(dest)}];
   else [busy=true; headpos=dest;]
   end request;
procedure release;
   begin busy=false;
   if direction==up then
   if {upsweep.queue then upsweep.signal
   else {direction=down; downsweep.signal}}
   else if downsweep.queue then downsweep.signal
   else {direction=up; upsweep.signal}
   end release;

headpos=0;
direction=up;
busy=false;
end diskhead
```

- Monitors is a style of programming where synchronization doesn't get mixed with other code; separate from other monitors

Unix implementation (optional information):
- Has generalized semaphores
- Each semaphore has queue of processes suspended on it
- The *semop* sys call takes a list of semaphore ops and does them one at a time
- If semop is positive, semaphore is incremented and all processes awaken
- If semop is zero, and semaphore value is 0, then continue, else block it
- If semop is negative and less than the semaphore value, they are added
- Lastly, if semop is negative and greater than the semaphore value, its suspended

Unix also uses signals, which are software interrupts processes send each other
- About 20 defined signals (interrupt, quit, illegal instruction, etc.)

Semaphore Implementation

6.5 in text

- No hardware implementation of P & V because too complicated, hard, and long
- One solution: disable interrupts
    - Simulates atomic, because dispatcher "can't" take control
    - Not completely true, because can't disable some interrupts or any traps

- Almost all processors have an atomic read-modify-write instruction
- E.g. Atomic increment value in memory, and then load and decrement value in memory
    - Operations are to increment to value in memory, load incremented value
    - Decrement value in memory
- **1st method:**
- Busy waiting loop:
    - *Init: A=0*
    - *Loop: increment A in memory, load A*
    - *If A != 1, then decrement decrement A in memory, go to loop*
    - **Critical Section**
    - *Decrement memory location A*
- Doesn't work! Due to indefinite postponement
    - For N processes (N > 2), it oscillates between 2 to N
- **2nd method: Swap**
    - Operation is: swap(local(i).lock) – interchanges values of two variables (special atomic operation w/ 2 loads and 1 store)
- Busy waiting loop:
    - *Init: lock = false*
    - *Local(i) = true*
    - *Repeat swap(local(i).lock) until local(i)==false*
    - *Critical section here*
    - *local(i) == true*
    - *lock = false*
- Not guaranteed to work in a certain amount of time; factor of randomness
- **3rd method: Test and set**
    - Set value to true, but return old value; use ordinary write to set back to false; lock is locked if its true
    - Tset(local(i), lock): local(i) = lock; lock = true
- Busy waiting loop:
    - *Init lock = false*
    - *Repeat(Tset(local(i), lock) until local(i) == false*
    - *Critical section*
    - *Lock = false*
- Works!

Will use **test and set** to implement semaphore:

P(S)

Disable interrupts
Local(i):=T
Repeat(Tset(local(i), S.lock)) until local(i) == false
If S>0, then S:=S-1, S.Lock = enable interrupts
Return
Add process to S.Q
S.Lock = false

Enable interrupts
Call dispatcher

V(S)

Disable interrupts
Local(i) = T
Repeat (Tset(local(i),S.Lock) until local(i)==false
If (S.Q is empty) S = S + 1
Else remove process from S.Q, wake it up
S.Lock = false
Enable interrupts

Why enable interrupts?

- If a process is in the middle of P or V, it can prevent you from unlocking it for a bit, and efficiency goes down
- Time issue

Technically can do with solution of "too much milk" problem instead of atomic operations, but why so ~~serious~~ complicated?