More Synchronization

Readers and Writers

Readers/Writers

- Basic problem
  - an object is shared among several processes
  - some can read
  - some can write
  - can't allow write/write, or read/write
- Ideally, writers get preference over readers
  - updates to DB involve changes, which is probably where the real action is.
    - consider the airline reservation example

A Simple Solution

```
semaphore_t semaphore; /* control access to state variable */
semaphore_t wrt;       /* control entry to a writer */
int readcount;         /* how many readers */

Write  Process
        P(wrt);     /* any one in there? */
          DO WRITING
        V(wrt);

Read Process
        P(semaphore);
                readcount++;
                if (readcount == 1) P(wrt);        /* if we're first, wait on writers */
        V(semaphore);
                /* Do Reading */
        P(semaphore);
                readcount--;
                if (readcount == 0) V(wrt);        /* no more readers, allow a writer */
        V(semaphore);
```


Discussion

- Readers only in if no writers
- Writers only in if no readers
- But, readers can block out writers.
  - forever
  - independent of when they show up
- We need to have more control over the SCHEDULING policy.
  - no reader can access the DB if there is a writer in it, OR there is a writer waiting.
- This is a more complicated policy to enforce
  - do not rely on the implementation of the underlying synchronization mechanisms

A Better Reader Writer Solution

- OK to read if no writers or waiting writers
- Ok to write if no readers or writers
- Introduce a bunch of "state" variables to talk about the state of the system.
  - AR = # active readers
  - WR = # waiting readers
  - AW = # active writers
  - WW = # waiting writers
- And some semaphores
  - OkToRead --> "Can Read"
  - OkToWrite --> "Can Write"
  - lock --> binary semaphore on the state variables

A Reader/Writer

```
Acquire Read Lock
        P(lock)
        IF ((AW+WW) == 0) {
                /* No writers */
                V(OkToRead);
                AR++;
        } else WR++;
        P(OkToRead);

        /* READ DATA */

Release Read Lock
        P(lock)
        AR--;
        if (AR == 0 && WW > 0) {
                /* awake writers first */
                V(OkToWrite);
                AW++;WW--;
        }
        V(lock)
```

```
AcquireWriteLock
        P(lock);
        if (AW + AR + WW == 0) {
                V(OkToWrite);
                AW++;
        } else WW++;
        V(lock);
        P(OkToWrite);

        /* WRITE DATA */

ReleaseWriteLock
        P(lock);
        AW--;
        if (WW > 0 ) {
                V(OkToWrite);
                AW++; WW--;
        } else while (WR > 0) {
                V(OkToRead);
                AR++; WR--;
        }
        V(lock);
```

Synchronization: Monitors

Synchronization with Semaphores

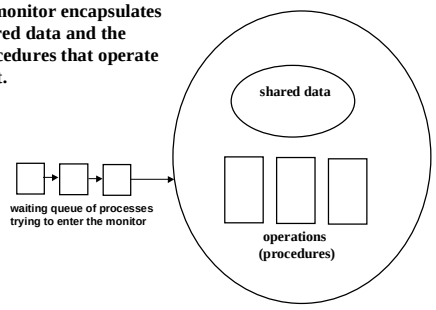
- **Semaphores can be used to solve any of the traditional synchronization problems, but suffer from several problems:**
  1. semaphores are essentially shared global variables
  2. there is no connection between the semaphore and the data being controlled by the semaphore
  3. Use same mechanism for scheduling and synchronization.
  4. they can be accessed from anywhere in the code
  5. there is no control or guarantee of proper usage
- **So, semaphores are sometimes hard to use and prone to bugs.**
- **One solution is to provide programming language support for synchronization.**
  - Why does putting something in the language make it harder to misuse?

Monitors

- **A monitor is a programming language construct that supports controlled access to shared data.**
  - leverage language, compiler, runtime.
  - advantages?
- **A monitor is a module that encapsulates:**
  1. some *shared data* structures
  2. *procedures* that operate on that shared data
  3. s*ynchronization* between concurrent processes that invoke those procedures
- **A monitor *protects* the data from unstructured access.**
- **The monitor guarantees that processes trying to access the data through its procedures interact only in legitimate ways.**

A Monitor

•A monitor encapsulates shared data and the procedures that operate on it.



Monitor Facilities

- **A monitor guarantees mutual exclusion**
  - only one process can be executing within the monitor at any instant
    - semaphore implicitly associated with monitor.
  - if a second process tries to enter a monitor procedure, it blocks until the first has left the monitor
  - More restrictive than semaphores
    - easier to use most of the time
- **Once in the monitor, a process may discover that it cannot continue, and may wish to sleep. Or it may wish to allow a waiting process to continue.**
  - *Condition Variables* provide synchronization within the monitor so that processes can wait or signal others to continue.

Condition Variables

- A place to wait. Sometimes called a "rendezvous point"
- The actual logic is provided by the program, not by the condition variable

```
BOOLEAN NoteEnoughMilk, MilkInTransit;
CONDITION MilkCondition
IF (NotEnoughMilk AND MilkInTransit) THEN
    Condition.Wait(MilkCondition);
```

- Three operations on condition variables
  - Condition.Wait(c)
    - release monitor lock, wait for someone to signal condition
  - Condition.Signal(c)
    - wakeup 1 waiting thread
  - Condition.Broadcast(c)
    - wakeup all waiting threads

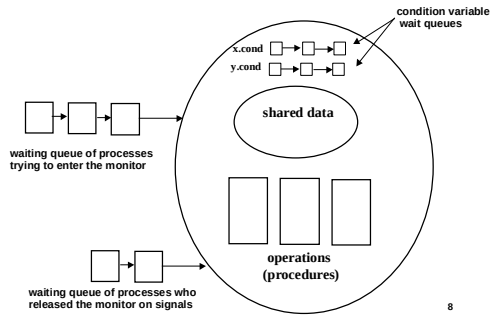
Basic Monitor Structure

```
resource:  monitor
begin
      busy: boolean;  free:  condition;
      procedure acquire;
      begin
            if busy then free.wait;
            busy = true;
      end

      procedure release;
      begin
            busy=false;
            free.signal;
      end

      busy=false              ; initialize busy
end
```


Basic Ideas

- **the monitor is controlled by a lock; only 1 process can enter the monitor at a time; others are queued**
- **condition variables provide a way to wait; when a process blocks on a condition variable, it givesup the lock.**
- **a process *signals* when a resource or condition has become available; this causes a waiting process to resume *immediately*. The lock is automatically passed to the waiter; the original process blocks.**

Monitors Have Several Associated Queues



Bounded Buffer Monitor Example

```
bounded buffer:  monitor
      begin buffer: array 0..N-1 of portion;
            lastpointer:  0..N-1;
            count: 0..N;
            nonempty, nonfull:  condition;
      procedure append(x: portion)
      begin if count = N then CONDITION.Wait(nonfull);
            buffer[lastpointer] := x;
            lastpointer:=(lastpointer+1) MOD N;
            count:=count+1;
            CONDITION.Signal(nonempty);
      end;
      procedure remove(result x:portion)
      begin if count = 0 then CONDITION.Wait(nonempty);
            x:=buffer[(lastpointer-count) MOD N];
            count:=count-1;
            CONDITION.Signal(nonfull.);
      end;
      count:=0;  lastpointer:=0;
end bounded buffer;
```


Monitors and Semaphores

- **Monitors and Semaphores can be implemented in terms of each other. E.g., to implement monitors with semaphores, we need:**
  - mutex : a sema to control entry to the monitor (init to 1)
  - next : a sema to suspend a process when it signals another (init to 0)
  - next-count : integer # of processes waiting due to signals
  - x-sem : a sema to suspend a process on a wait (init to 0) [one semaphore for each condition]
  - x-count: integer # of proc. waiting due to waiting on condition [one for each condition]

Monitors implemented with Semaphores

```
P(mutex);
 < body of operation>
if next-count > 0 then V(next) else V(mutex);

x.wait:
      x-count:=x-count+1;
      if next-count>0 then V(next) else  V (mutex);
      P(x-sem);
      x-count:=xcount-1

x.signal
      if x-count>0
      then begin
            next-count:=next_count+1;
            V(x-sem);
            P(next);
            next-count:=next-count-1;
            end;
```

General entry wrapper for all operations.

Example of Wait/Signal

```
alarmclock: monitor
begin now: integer
      wakeup: condition;

proc wake (n:int);    wake me in n clock ticks
begin alarmsetting: int;
      alarmsetting := now + n;
      while now < alarmsetting do CONDITION.Wait(alarmsetting);
      CONDITION.Signal(wakeup);
end;

proc tick;   clock tick called by external clock
begin
      now = now + 1;
      CONDITION.Signal(wakeup);
end;

end alarmclock;
```


Two kinds of Monitors

- HOARE Monitors
  - SIGNAL(c)
    - Run waiter immediately. Signaller blocks right now.
    - Condition is guaranteed to hold when blocker runs.
    - But, signaller must RESTORE MONITOR INVARIANTS before signalling.
- MESA Monitors
  - SIGNAL(c)
    - waiter is made ready, but the signaller continues.
    - Condition is not necessarily true when the waiter runs again.
    - Signaller must not restore invariant until it leaves the MONITOR
      - either with a WAIT or an explicit return/
    - WAKEUP is only a HINT that something must have changed.
      - must recheck conditional case.

Examples

- HOARE

```
if (NotReady)
      Condition.Wait(C);
```

- MESA

```
while (NotReady)
      Condition.Wait(C);
```

- MESA monitors easier to use
  - more efficient
    - fewer switches
    - directly supports broadcast.
- Hoare monitors leave less to “chance.”

In Summary...

- MONITORS
  - Use different mechanism for scheduling and mutual exclusion
  - are higher level than semaphores
- Protect CODE and not data
  - consider the difference
- May require some higher level language support...
- Mutexes are an alternative...
  - hybrid of monitors and semaphores

Mutex Example

```
mutex_t mu
condition_t co;
boolean ready;
…
foo()
{
        mutex_lock(mu)
                if (!ready)
                        condition_wait(co, mu);
                }
                …
                …
                ready = TRUE;
                condition_signal(mu);
        mutex_unlock(mu);
}
```