More Synchronization

Readers and Writers and Deadlock

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

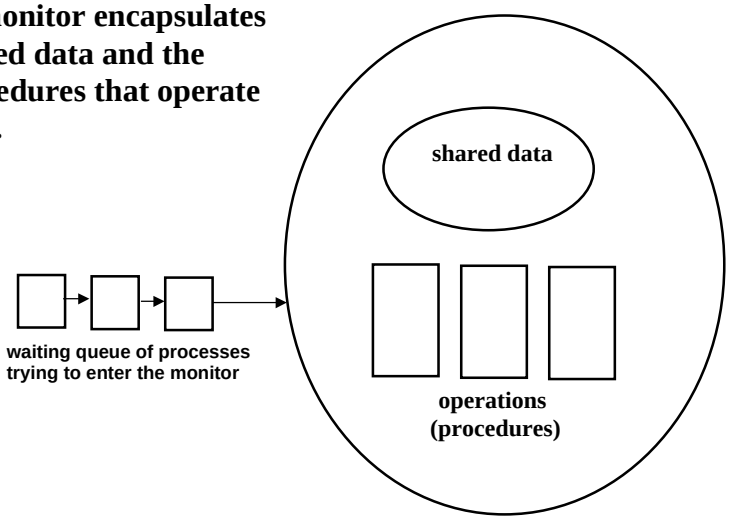
- **Semaphores can be used to solve any of the traditional synchronization problems, but suffer from several problems:**
  1. semaphores are essentially shared global variables
  2. there is no connection between the semaphore and the data being controlled by the semaphore
  3. Use same mechanism for 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. *synchronization* 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

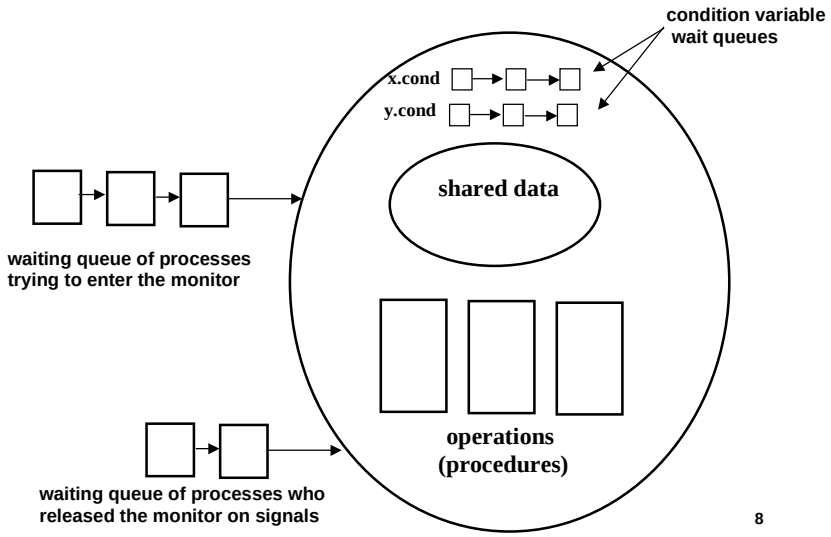
Basic Monitor Structure

```
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);
```

General entry wrapper for all operations.

```
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;
```

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);
}
```

Deadlock

Consider the alternative....

DeadLock

- A collection of processes are each waiting on a resource held by another.

P(s) P(q)	P(q) P(s)

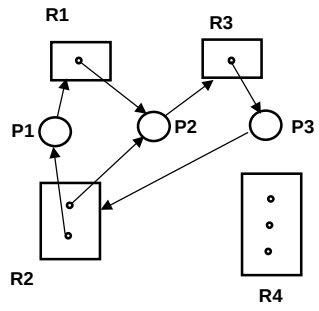
Deadlock. You're Stuck.

- Deadlock is a problem that can exist when a group of processes compete for access to fixed resources.
- Def: deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set.
- Example: two processes share 2 resources that they must *request* (before using) and *release* (after using). Request either gives access or causes the proc. to block until the resource is available.

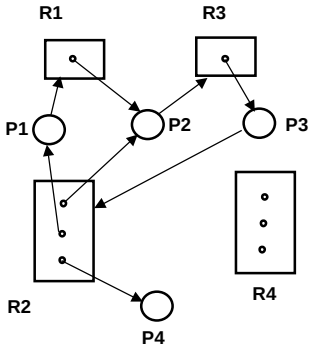
Resource Allocation Graph

- Deadlock can be described through a *resource allocation graph*.
- The RAG consists of a set of vertices $P=\{P_1,P_2,..,P_n\}$ of processes and $R=\{R_1,R_2,..,R_m\}$ of resources.
- A directed edge from a process to a resource
  - P_i->R_j, implies that P_i has requested R_j.
- A directed edge from a resource to a process
  - R_j->P_i, implies that R_j has been allocated by P_i.
- If the graph has no cycles, deadlock cannot exist. If the graph has a cycle, deadlock may exist.

Resource Allocation Graph Example



There are two cycles here: P1-R1-P2-R3-P3-R2-P1 and P2-R3-P3-R2-P2, and there is *deadlock*.



Same cycles, but no deadlock.

Four Conditions for Deadlock

- Deadlock can exist if and only if 4 conditions hold simultaneously:
  - 1. mutual exclusion: at least one process must be held in a non-sharable mode.
  - 2. hold and wait: there must be a process holding one resource and waiting for another
  - 3. no preemption: resources cannot be preempted
  - 4. circular wait: there must exist a set of processes [p1, p2, ..., pn] such that p1 is waiting for p2, p2 for p3, and so on...

Dealing with Deadlock

- Prevention
  - deny one of the four conditions
- Avoidance
  - Use information from requestors to “be careful” about allocation
  - Rationing
- Detection and Recovery
  - Wait until too late and violate one of the four conditions
- Stop
  - PANIC()
- Ignore it and keep going
  - BAD

Possible Approaches

- **Deadlock Prevention: ensure that at least 1 of the necessary conditions cannot exist.**
  - mutual exclusion: make resources sharable (isn’t really possible for some resources)
  - hold and wait: guarantee that a process cannot hold a resource when it requests another, or, make processes request all needed resources at once, or, make it release all resources before requesting a new set.
    - pre-determine everything you need.
  - circular wait: impose an ordering (numbering) on the resources and request them in order
    - example, order semaphores “alphabetically”
      - P(s);P(t) is ok, but P(t);P(s) is not.
      - ensures that everyone acquires semaphores in the same order.

Why does ordering work?

Assume we impose an ordering on resource acquisition such that a process can not acquire R_i where i>j unless $C(R_i) > C(R_j)$.

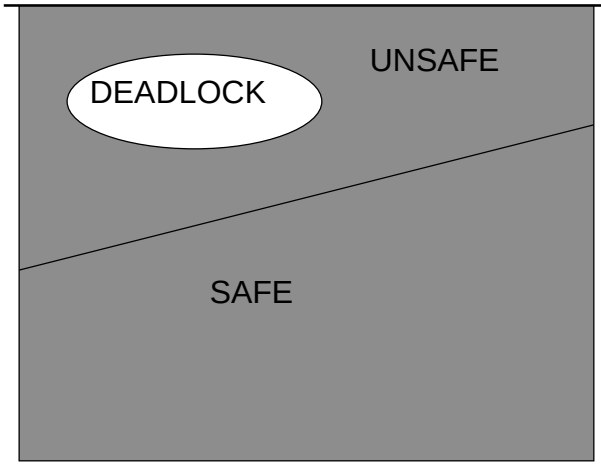
Now, assume that circular wait exists among a set of processes {P0,P1,..Pn}

Process P_i waits for R_i which is held by $R_{i+1.}$
For example, P0 waits for R0, which is held by P1.
Then, P_n waits for R_n which is held by P0.

In general, P_{i+1} holds R_i while waiting for $R_{i+1.}$
==> $C(R_i) < C(R_{i+1})$ for all i.
==> C(R0) < C(R1) < C(R2) ... < C(Rn) < C(R0)
==> C(R0) < C(R0) which is bogus.

So, a circular wait can not exist.

Deadlock Avoidance



Consider the space of all process allocation states.
# of available and allocated resources, with max request for each process known.

Some of those are safe.
no chance for deadlock.
the next request is satisfiable.
Others are unsafe.
something could happen that leads to deadlock.

Possible Approaches

- Deadlock Avoidance
  - general idea: provide info in advance about what resources will be needed by processes toguarantee that deadlock will not exist.
- E.g., define a sequence of procs <P1,P2,..Pn> as safe if for each Pi, the resources that Pi can still request can be satisified by the currently available resources plus the resources held by all Pj, j<i.
  - this avoids circularities.
  - when a process requests a resource, the system grants or forces it to wait, depending on whether this would be an unsafe state.

Example:

Processes p0, p1, and p2 compete for 12 tape drives

	max need	current usage	could ask for
p0	10	5	5
p1	4	2	2
p2	9	2	7

3 drives remain

current state is safe because a safe sequence exists: <p1,p0,p2>

p1 can complete with current resources
p0 can complete with current+p1
p2 can complete with current +p1+p0

if p2 requests 1 drive, then it must wait, because that state would be unsafe. (p2 could request its full allocation)

The Banker's Algorithm

Banker's algorithm decides whether to grant a resource request. Define data structures:

n: integer	**# of processes**
m: integer	**# of resources**
available[1..m]	**avail[i] is # of avail resources of type i**
max[1..n,1..m]	**max demand of each Pi for each Ri**
allocation[1..n,1..m]	**current allocation of resource Rj to Pi**
need[1..n,1..m]	**max # of resource Rj that Pi may still request**

let request[i] be a vector of the # of isntances of resource Rj that Process Pi wants

The Basic Algorithm

1. If request[i] > need[i] then error (asked for too much)
2. If request[i] > available[i] then wait (can't supply it now)
3. Resources are available to satisfy the request:
   Let's *assume* that we satisfy the request. Then we would have:
   available = available - request[i]
   allocation[i] = allocatoin[i] + request[i]
   need[i] = need[i] - request[i]
   Now, check if this would leave us in a safe state;
   if yes, grant the request, if no, then leave the state as is and cause the process to wait.

Safety Check

1. new vars: work[1..m] = available ; to accumulate resources
finish[1..n] = false (for all i) ; none finished yet
2. find an i s.t. finish[i]= false and need[i] <= work
(find a proc that can complete its request now)
if no such i exists, go to step 4 (we're done)
3. Found an i:
finish[i] = true ; done with this process
work = work + allocation[i] (assume this process were to finish, add its allocation back to the available list)
go to step 2
4. If finish[i] = true for all i, the system is safe.

Deadlock Detection

- If there is neither deadlock prevention nor avoidance, then deadlocks may occur.
- In this case, we must have:
  - an algorithm that determines whether a deadlock has occurred
  - an algorithm to recover from the deadlock
- This is doable, but it's costly.
- Basic idea is to reduce the resource allocation graph by processes whose request may be granted
  - remove arrows to that process
  - remove arrows from that process
- If no processes exist at the end, we have no deadlock.
- If there exist processes, then we have deadlock.

Deadlock Detection Algorithm

available[1..m] ; # of available resources

allocation[1..n,1..m] ; # of resource of each Ri allocated to Pj

request[1..n,1..m] ; # of resources of each Ri requested by Pj

1. work=available
   for all i < n, if allocation[i] not 0
      then finish[i]= false else finish[i]=true
2. find an index i such that:
      finish[i]=false;
      request[i] <= work
      if no such i exists, go to 4.
3. work=work+allocation[i]
    finish[i] = true, go to 2
4. if finish[i] = false for some i, then system is deadlocked with Pi in deadlock

Deadlock

- Deadlock detection algorithm is expensive. How often we invoke it depends on:
  - how often or likely is deadlock
  - how many processes are likely to be affected when deadlock occurs

Deadlock Recovery

- Once a deadlock is detected, there are 2 choices:
  - 1. abort all deadlocked processes (which will cost in the repeated computations necessary)
  - 2. abort 1 process at a time until cycle is eliminated (which requires re-running the detection algorithm after each abort)
- Or, could do process preemption: release resources until system can continue. Issues:
  - selecting the victim (could be clever based on R's allocated.
  - rollback (must rollback the victim to a previous state)
  - starvation (must not always pick same victim)