G22.2250-001
Operating Systems

Lecture 4

Process Synchronization (cont'd)
Classical Synchronization Problems

September 25, 2007

Outline

Announcements
- Lab 1 was due on Friday
  - Please see me after class if you have not handed this in

- Process synchronization primitives
  - (Review) Locks, Semaphores
  - Condition variables
  - Implementation techniques
- Classical synchronization problems
  - Mutual exclusion, sequencing, bounded buffer
  - Readers-writers, dining philosophers
  - A larger example
- Language support for synchronization
  - Conditional critical regions
  - Monitors

[Silberschatz/Galvin/Gagne: Chapters 6.4-6.8]

(Review)
Synchronization Primitives (1): Locks (Mutexes)

- Locks
  - a single boolean variable **L**
    - in one of two states: **AVAILABLE, BUSY**
  - accessed via two *atomic* operations
    - **LOCK** (also known as **Acquire**)
      ```
      while ( L != AVAILABLE ) wait-a-bit
      L = BUSY;
      ```
    - **UNLOCK** (also known as **Release**)
      ```
      L = AVAILABLE;
      wake up a waiting process (if any)
      ```
  - process(es) waiting on a LOCK cannot “lock-out” process doing UNLOCK

- Critical sections using locks
  ```
  LOCK( L )
  CRITICAL SECTION
  UNLOCK( L )
  ```
  - Mutual exclusion? Progress? Bounded waiting?

(Review)
Synchronization Primitives (2): Semaphores

- Semaphores
  - a single integer variable S
  - accessed via two *atomic* operations
    - **WAIT** (sometimes denoted by **P**)
      ```
      while S <= 0 do wait-a-bit;
      S := S-1;
      ```
    - **SIGNAL** (sometimes denoted by **V**)
      ```
      S := S+1;
      wake up a waiting process (if any)
      ```
  - WAITing process(es) cannot “lock out” a SIGNALing process

- Binary semaphores
  - S is restricted to take on only the values 0 and 1
  - WAIT and SIGNAL become similar to LOCK and UNLOCK
  - are *universal* in that counting semaphores can be built out of them

Universality of Binary Semaphores

- Implement operations on a (counting) semaphore **CountSem**
  - use binary semaphores S1 = 1, S2 = 0
  - integer C = initial value of counting semaphore

```
P(CountSem)

P(S1);
C := C-1;
if ( C < 0 ) then
   begin V(S1); P(S2); end
V(S1);
```

```
V(CountSem)

P(S1);
C := C+1;
if ( C <= 0 ) then V(S2);
               else V(S1);
```

  - S1 ensures mutual exclusion for accessing C
  - S2 is used to block processes when C < 0
  - is a race condition possible after V(S1) but before P(S2)?

Synchronization Primitives (3): Condition Variables

- Condition variables
  - an *implicit* process queue
  - three operations that *must be performed within a critical section*
    - WAIT
      associate self with the implicit queue
      suspend self
    - SIGNAL
      wake up exactly one suspended process on queue
      - has no effect if there are no suspended processes
    - BROADCAST
      wake up all suspended processes on queue

- Two types based on what happens to the process doing the SIGNAL
  - Mesa style (Nachos uses Mesa-style condition variables)
    - SIGNAL-ing process continues in the critical section
    - resumed process must re-enter (so, is not guaranteed to be the next one)
  - Hoare style
    - SIGNAL-ing process immediately exits the critical section
    - resumed process now occupies the critical section

Uses of Condition Variables

- Can be used for constructing
  - critical sections, sequencing, …
- Primary use is for waiting on an event to happen
  - after checking that it has not already happened
    - WHY IS THIS IMPORTANT?
- Example: Three processes that need to cycle among themselves

  <print 0>; <print 1>; <print 2>; <print 0>; <print 1>; …
  - One variable: **turn**; three condition variables: $\mathbf{cv_0}$, $\mathbf{cv_1}$, $\mathbf{cv_2}$
  - Process P_i executes (in a critical section)

```
while ( turn != i) WAIT(cv_i)
<do the operation>
turn := (turn + 1) mod 3; SIGNAL(cv_turn)
```


Implementing the Synchronization Primitives

- Need support for atomic operations from the underlying hardware
  - applicable only to a small number of instructions
    - else, can implement critical sections this way

Three choices

- Use n-process mutual-exclusion solutions
  - complicated
- ✓ Selectively disable interrupts on uniprocessors
  - so, no unanticipated context switches ➡ atomic execution
  - solution adopted in Nachos (see Lab 2 for details)
- ✓ Rely on special hardware synchronization instructions
- Can implement one primitive in terms of another
  - Nachos Lab 2

Implementation Choices (1): Interrupt Disabling

- Semaphores

  P(S)

```
DISABLE-INTERRUPTS
while S <= 0 do wait-a-bit
            [ENABLE-INTERRUPTS; YIELD CPU; DISABLE-INTERRUPTS]
S := S-1;
ENABLE-INTERRUPTS
```

  V(S)

```
DISABLE-INTERRUPTS
S := S+1;
ENABLE-INTERRUPTS
```

- Drawback
  - a process spins on this loop (busy waiting) till it can enter critical section
  - can waste *substantial* amount of CPU cycles idling
    - Even if `wait-a-bit` is implemented as
      - give up CPU (i.e. put at the end of ready queue)
    - since there are still context switches
  - not a very useful utilization of valuable cycles

Efficient Semaphores

- Implement P and V differently
  - maintain an explicit *wait queue* organized as a scheduler structure

```
type semaphore = record
        value: integer;
        L: list of processes;
        end;

P(S):   S.value := S.value - 1;        V(S):   S.value := S.value + 1;
        if ( S.value < 0 )                     if ( S.value <= 0 )
           then begin                             then begin
                add process to S.L                     remove P from S.L
                block;                                 wakeup(P);
           end;                                   end;
```

  - still need atomicity: can use previously discussed solutions
    - can have spinning but only for a small period of time (~10 instructions)
  - queue enqueue/dequeue must be fair
    - not required by semantics of semaphores

Implementation Choices (2): Hardware Support

- Rationale: Hardware instructions enable simpler/efficient solutions to common synchronization problems
  - disabling interrupts is a brute-force approach
  - does not work on multiprocessors
    - simultaneous disabling of all interrupts is not feasible

- Two common primitives
  - test-and-set
  - swap

Semantics of Hardware Primitives

- Test-and-set
  - given boolean variables X, Y, atomically set X := Y; Y := true

```
boolean Test-and-set( boolean &target ) {
    boolean rv = target;
    target = true;
    return rv;
}
```

- Swap
  - atomically exchange the values of given variables X and Y

```
temp = X; X = Y; Y = temp;
```

  - can emulate test-and-set

```
boolean Test-and-set( boolean &target ) {
  boolean t := true;
  swap (target, t);
  return t;
}
```

Implementing Locks Using Test-and-Set

LOCK: `L : boolean := false`
`while Test-and-set(lock) wait-a-bit`

UNLOCK `lock := false`

- Properties of this implementation
  - Mutual exclusion?
    - first process P_i entering critical section sets lock := true
      - test-and-set (from other processes) evaluates to true after this
    - when P_i exits, lock is set to false, so the next process P_j to execute the instruction will find test-and-set = false and will enter the critical section
  - Progress?
    - trivially true
  - Unbounded waiting
    - possible since depending on the timing of evaluating the test-and-set primitive, other processes can enter the critical section first
    - See Section 6.4 for a solution to this problem

Synchronization Primitives in Real OSes

- Unix: Single CPU OS
  - implement critical sections using interrupt elevation
    - disallow interrupts that can modify the same data
    - (Linux 2.4 and earlier, Section 6.8.3) disable kernel preemption
  - another possibility: interrupts never "force" a context switch
    - they just set flags, or wake up processes
  - primitives
    - **`sleep`** (address);
    - **`wake_up`** (address); -- wakes up all processes sleeping on address
  - typical code

```
ENTRY: while (locked) sleep(bufaddr);
       locked = true;
EXIT:  locked = false; wake_up (bufaddr);
```

Synchronization Primitives in Real OSes (contd.)

- Solaris 2: multi-CPU OS
  - for brief accesses only
    - adaptive mutexes
    - starts off as a standard spinlock semaphore
      - if lock is held by running thread, continues to spin
        - » valid only on a multi-CPU system
      - otherwise blocks
  - for long-held locks
    - (process queue) semaphores
    - condition variables
      - wait and signal
    - reader-writer locks
      - for frequent mostly read-only accesses
  - turnstiles
    - the queue structure on which threads block when waiting for a lock
    - associated with threads rather than lock objects
      - Each thread can block on at most one object, so more efficient

Outline

Announcements
- Lab 1 was due on Friday
  - Please see me after class if you have not handed this in

- Process synchronization primitives
  - (Review) Locks, Semaphores
  - Condition variables
  - Implementation techniques
- Classical synchronization problems
  - Mutual exclusion, sequencing, bounded buffer
  - Readers-writers, dining philosophers
  - A larger example
- Language support for synchronization
  - Conditional critical regions
  - Monitors

[Silberschatz/Galvin/Gagne: Chapters 6.4-6.8]

Classical Synchronization Problems

- Commonly encountered problems in operating systems
  - used to test any proposal for a new synchronization primitive

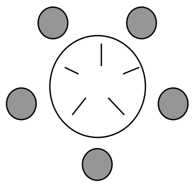
1. Mutual exclusion
   - only one process executes a piece of code (critical section) at any time
   - OS examples: access to shared resources
     - e.g., a printer
2. Sequencing
   - a process waits for another process to finish executing some code
   - OS examples: waiting for an event
     - e.g., recv suspends until there is some data to read on the network

Classical Synchronization Problems (cont'd)

3. Bounded-buffer (also referred to as the Producer-Consumer problem)
   - a pool of n buffers
   - *producer* process(es) put items into the pool
   - *consumer* process(es) take items out of the pool
   - issues: mutual exclusion, empty pool, and full pool
   - OS examples: buffering for pipes, file caches, etc.
4. Readers-Writers
   - multiple processes access a shared data object X
     - any number of *readers* can access X at the same time
     - no *writer* can access it at the same time as a *reader* or another *writer*
   - mutual exclusion is too constraining: WHY?
   - variations:
     - reader-priority: a reader must not wait for a writer
     - writer-priority: a writer must not wait for a reader
   - OS examples: file locks

Classical Synchronization Problems (contd.)

5. Dining Philosophers
   - 5 philosophers
   - 5 chopsticks placed between them
     - to eat requires two chopsticks
   - philosophers alternate between thinking and eating
   - issues: deadlock, starvation, fairness
   - OS examples: simultaneous use of multiple resources
     - e.g., disk bandwidth and storage



Mutual Exclusion and Sequencing Using Semaphores

- Mutual exclusion: Semaphore initialized to 1

```
P(S);
CRITICAL SECTION
V(S);
```

- Sequencing: Semaphore initialized to 0

```
process 1          process 2
                      B();
                      V(S);
  P(S);
  A();
```

Bounded-buffer Using Semaphores

- Three semaphores
  - **mutex**: provide mutual exclusion between processes (initial value = 1)
  - **empty**: count the number of empty slots (initial value = N)
  - **full**: count the number of full slots (initial value = 0)

```
Producer(s):

 repeat
    // produce item in nextp
    P( empty );
    P( mutex );
    // add nextp to buffer
    V( mutex );
    V( full );
 until false;
```

```
Consumer(s):

 repeat
    P( full );
    P( mutex );
    // remove item to nextc
    V( mutex );
    V( empty );
    // consume item in nextc
 until false;
```


Readers-Writers Using Semaphores

To allow multiple readers, synchronize only the first/last reader with writers

```
Reader(s)

P(x);
  rcount := rcount + 1;
  if (rcount == 1) then P(wsem);
V(x);

READ

P(x);
  rcount := rcount - 1;
  if (rcount == 0) then V(wsem);
V(x);
```

```
Writer(s)

P(wsem);
WRITE
V(wsem);
```

stream of readers can starve writers

can release either waiting readers or writers

Readers-Writers Using Semaphores: Writer-Priority

Have a writer block out subsequent readers (same as readers block out writers)

Reader

readers can queue up preventing a waiting writer from setting rsem

```
P(rsem);
P(x);
  rcount := rcount + 1;
  if (rcount == 1) then P(wsem);
V(x);
V(rsem);

READ


P(x);
  rcount := rcount - 1;
  if (rcount == 0) then V(wsem);
V(x);
```

Writer

```
P(y);
  wcount := wcount + 1;
  if (wcount == 1) then P(rsem);
V(y);

P(wsem);
WRITE
V(wsem);

P(y);
  wcount := wcount - 1;
  if (wcount == 0) then V(rsem);
V(y);
```


Readers-Writers Using Semaphores: Writer-Priority (2)

Reader

```
P(z);
P(rsem);
P(x);
  rcount := rcount + 1;
  if (rcount == 1) then P(wsem);
V(x);
V(rsem);
V(z);
READ


P(x);
  rcount := rcount - 1;
  if (rcount == 0) then V(wsem);
V(x);
```

Writer

```
P(y);
  wcount := wcount + 1;
  if (wcount == 1) then P(rsem);
V(y);

P(wsem);
WRITE
V(wsem);

P(y);
  wcount := wcount - 1;
  if (wcount == 0) then V(rsem);
V(y);
```

Dining Philosophers Using Semaphores

```
Philosopher_i

P( chopstick[i] );
P( chopstick[i+1 mod 5] );

EAT

V( chopstick[i] );
V( chopstick[i+1 mod 5] );

THINK
```

```
Philosopher(j=i+1 mod 5)

P( chopstick[j] );
P( chopstick[j+1 mod 5] );

EAT

V( chopstick[j] );
V( chopstick[j+1 mod 5] );

THINK
```

- Deadlock

  a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set

  - details in Lectures 5 and 6.

Dining Philosophers Using Semaphores - 2

```
Philosopher(even i)

P( chopstick[i] );
P( chopstick[i+1 mod 5] );

EAT

V( chopstick[i] );
V( chopstick[i+1 mod 5] );

THINK
```

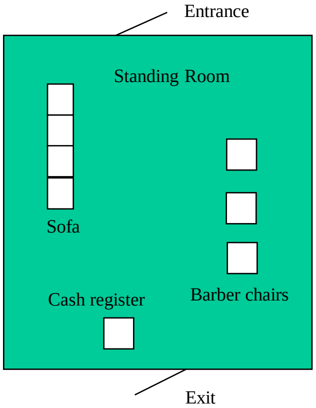
```
Philosopher(odd i)

P( chopstick[i+1] mod 5 );
P( chopstick[i] );

EAT

V( chopstick[i+1] mod 5 );
V( chopstick[i] );

THINK
```

- Alternate solutions
  - allow at most 4 philosophers to sit simultaneously at the table
  - allow a philosopher to pick up chopsticks only if both are available
- All of these solutions suffer from the possibility of starvation!

A Larger Example: A Barbershop Problem

- Example taken from
  Operating Systems: Internals and Design Principles, 3rd Edition
  William Stallings, Prentice Hall, 1998
- The problem: Orchestrating activities in a barbershop
  - 3 chairs, 3 barbers, 1 cash register,
    waiting area: 4 customers on a sofa, plus additional standing room
  - Fire codes limit total number of customers to 20 at a time
  - A customer
    - Will not enter the shop if it is filled to capacity
    - Takes a seat on the sofa, or stands if sofa is filled
    - When a barber is free, the customer waiting longest on sofa is served
      The customer standing the longest takes up seat on the sofa
    - When a customer's haircut is finished, any barber can accept payment but because of the single cash register, only one payment is accepted at a time
    - Barbers divide their time between cutting hair, accepting payment, and sleeping

A Barbershop Problem (cont'd)



- Shop and sofa capacity
  - **`max_capacity`** (initial value = 20)
  - **`sofa`** (initial value = 4)
- Barber chair capacity
  - **`barber_chair`** (initial value = 3)
- Ensuring customers are in barber chair
  - **`cust_ready`** (initial value = 0)
    - barber waits for customer
  - **`finished`** (initial value = 0)
    - customer waits for haircut to finish
  - **`leave_b_chair`** (initial value = 0)
    - barber waits for chair to empty
- Paying and receiving
  - **`payment`** (initial value = 0)
    - cashier waits for customer to pay
  - **`receipt`** (initial value = 0)
    - customer waits for cashier to ack
- Coordinating barber functions
  - **`coord`** (initial value = 0)
    - wait for a barber resource to free up

A Barbershop Problem (cont'd)

- Shop and sofa capacity
  - `max_capacity` (: = 20)
  - `sofa` (: = 4)
- Barber chair capacity
  - `barber_chair` (: = 3)
- Ensuring customers are in barber chair
  - `cust_ready` (: = 0)
  - `finished` (: = 0)
  - `leave_b_chair` (: = 0)
- Paying and receiving
  - `payment` (: = 0)
  - `receipt` (:= 0)
- Coordinating barber functions
  - `coord` (:= 0)

Customer

```
P( max_capacity );
// enter shop
P( sofa );
// sit on sofa
P( barber_chair );
// get up from sofa
V( sofa );
// sit in barber chair
V( cust_ready );
P( finished );
// leave barber chair
V( leave_b_chair );
// pay
V( payment );
P( receipt );
// exit shop
V( max_capacity );
```

Barber

```
P( cust_ready );
P( coord );
// cut hair
V( coord );
V( finished );
// wait for customer to leave
P( leave_b_chair );
// tell next customer to hop on
V( barber_chair );
```

Cashier

```
P( payment );
P( coord );
// accept payment
V( coord );
V( receipt );
```


A Barbershop Problem (cont'd): Mutual Exclusion

- Shop and sofa capacity
  - `max_capacity` (: = 20)
  - `sofa` (: = 4)
- Barber chair capacity
  - `barber_chair` (: = 3)
- Ensuring customers are in barber chair
  - `cust_ready` (: = 0)
  - `finished` (: = 0)
  - `leave_b_chair` (: = 0)
- Paying and receiving
  - `payment` (: = 0)
  - `receipt` (:= 0)
- Coordinating barber functions
  - `coord` (:= 0)

Customer

```
P( max_capacity );
// enter shop
P( sofa );
// sit on sofa
P( barber_chair );
// get up from sofa
V( sofa );
// sit in barber chair
V( cust_ready );
P( finished );
// leave barber chair
V( leave_b_chair );
// pay
V( payment );
P( receipt );
// exit shop
V( max_capacity );
```

Barber

```
P( cust_ready );
P( coord );
// cut hair
V( coord );
V( finished );
// wait for customer to leave
P( leave_b_chair );
// tell next customer to hop on
V( barber_chair );
```

Cashier

```
P( payment );
P( coord );
// accept payment
V( coord );
V( receipt );
```

A Barbershop Problem (cont'd): Bounded Buffer

- Shop and sofa capacity
  - `max_capacity` (: = 20)
  - `sofa` (: = 4)
- Barber chair capacity
  - `barber_chair` (: = 3)
- Ensuring customers are in barber chair
  - `cust_ready` (: = 0)
  - `finished` (: = 0)
  - `leave_b_chair` (: = 0)
- Paying and receiving
  - `payment` (: = 0)
  - `receipt` (:= 0)
- Coordinating barber functions
  - `coord` (:= 0)

Customer

```
P( max_capacity );
// enter shop
P( sofa );
// sit on sofa
P( barber_chair );
// get up from sofa
V( sofa );
// sit in barber chair
V( cust_ready );
P( finished );
// leave barber chair
V( leave_b_chair );
// pay
V( payment );
P( receipt );
// exit shop
V( max_capacity );
```

Barber

```
P( cust_ready );
P( coord );
// cut hair
V( coord );
V( finished );
// wait for customer to leave
P( leave_b_chair );
// tell next customer to hop on
V( barber_chair );
```

Cashier

```
P( payment );
P( coord );
// accept payment
V( coord );
V( receipt );
```

9/25/2007 31

A Barbershop Problem (cont'd): Sequencing

- Shop and sofa capacity
  - `max_capacity` (: = 20)
  - `sofa` (: = 4)
- Barber chair capacity
  - `barber_chair` (: = 3)
- Ensuring customers are in barber chair
  - `cust_ready` (: = 0)
  - `finished` (: = 0)
  - `leave_b_chair` (: = 0)
- Paying and receiving
  - `payment` (: = 0)
  - `receipt` (:= 0)
- Coordinating barber functions
  - `coord` (:= 0)

Customer

```
P( max_capacity );
// enter shop
P( sofa );
// sit on sofa
P( barber_chair );
// get up from sofa
V( sofa );
// sit in barber chair
V( cust_ready );
P( finished );
// leave barber chair
V( leave_b_chair )
// pay
V( payment );
P( receipt );
// exit shop
V( max_capacity );
```

Barber

```
P( cust_ready );
P( coord );
// cut hair
V( coord );
V( finished );
// wait for customer to leave
P( leave_b_chair );
// tell next customer to hop on
V( barber_chair );
```

Cashier

```
P( payment );
P( coord );
// accept payment
V( coord );
V( receipt );
```

9/25/2007 32

A Barbershop Problem (cont'd)

- Some problems with the current solution
  - since all customers are waiting on the same semaphore (`finished`), the one who started earliest is released when a barber does **V(** `finished`**)**
    - even if the haircut is not done
  - similar problem with the cashier and the `pay` and `receipt` semaphores
    - cashier may accept money from one customer and release another
  - a customer needs to wait on the sofa even if a barber chair is free

- All of these can be solved using additional semaphores

Outline

Announcements

- Lab 1 was due on Friday
  - Please see me after class if you have not handed this in

- Process synchronization primitives
  - (Review) Locks, Semaphores
  - Condition variables
  - Implementation techniques
- Classical synchronization problems
  - Mutual exclusion, sequencing, bounded buffer
  - Readers-writers, dining philosophers
  - A larger example
- Language support for synchronization
  - Conditional critical regions
  - Monitors

[Silberschatz/Galvin/Gagne: Chapters 6.4-6.8]

Limitations of Semaphores

- No abstraction and modularity
  - a process that uses a semaphore has to know which other processes use the semaphore, and how these processes use the semaphore
  - a process cannot be written in isolation

- Consider sequencing between three processes
  - P_1, P_2, P_3, P_1, P_2, P_3, …

P_1	P_2	P_3
`P( sem1 );`	`P( sem2 );`	`P( sem3 );`
`// do stuff`	`// do stuff`	`// do stuff`
`V( sem2 );`	`V( sem3 );`	`V( sem1 );`

What happens if there are only two processes?
What happens if you want to use this solution for four processes?

Limitations of Semaphores (cont'd)

- Very easy to write incorrect code
  - changing the order of P and V
    - can violate mutual exclusion requirements

```
V( mutex ); CODE; P( mutex ); instead of
P( mutex ); CODE; V( mutex );
```

    - can cause deadlock

```
P( seq ); instead of
V( seq );
```

  - similar problems with omission

- Extremely difficult to verify programs for correctness
- ► Need for still higher-level synchronization abstractions!

Language Support

- Helps simplify expression of synchronization
  - more convenient
  - more secure
  - less buggy
- We shall examine two fundamental constructs
  - conditional critical regions
  - monitors
- These constructs can be found in several concurrent languages
  - Communicating Sequential Processes (CSP) *critical regions*
  - Concurrent Pascal *monitors*
  - object-oriented languages: Modula-2, Concurrent C, Java
  - Ada83, Ada95

Conditional Critical Regions

- A high-level language declaration
  - informally, it can be used to specify that while a statement *S* is being executed, no more than one process can access a distinguished variable *v*
  - notation

```
var v: shared t;
region v when B do S;
```

  - ***v*** is shared and of type *t*
    - can only be accessed within a region statement
  - ***B*** is a Boolean expression
  - ***S*** is a statement
    - can be a compound statement
- Semantics
  - A process is guaranteed mutually exclusive access to the region `v`
  - Checking of `B` and entry into the region happens atomically

Conditional Critical Regions: Benefits

Bounded-buffer producer/consumer

```
var buffer : shared record
   pool: array [0..n-1] of item;
   count, in, out: integer;
end;

Producer:
region buffer when count < n
  do begin
     pool[in] := nextp;
     in := (in + 1) mod n;
     count := count + 1;
  end;

Consumer:
region buffer when count > 0
  do begin
     nextc := pool[out];
     out := (out + 1) mod n;
     count := count - 1;
  end;
```

- Guards against simple errors associated with semaphores
  - e.g., changing the order of P and V operations, or forgetting to put one of them
- Division of responsibility
  - the *developer* does not have to program the semaphore or alternate synchronization explicitly
  - the *compiler* ``automatically'' plugs in the synchronization code using predefined libraries
  - once done carefully, *reduces* likelihood of mistakes in designing the delicate synchronization code

Conditional Critical Regions: Implementation

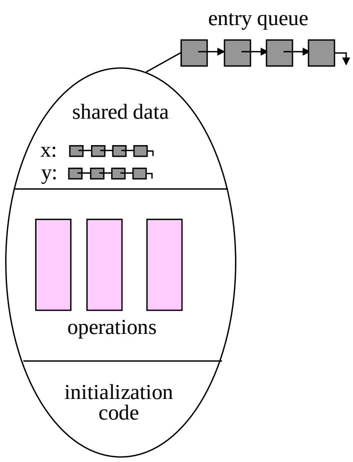
```
var mutex: semaphore;

P( mutex );
while not B
   do begin
        try-and-enter;
   end;
S;
leave-critical-region;
```

```
var delay: semaphore;
var count: integer;

count++ ;
V( mutex );
P( delay );
// check condition
if ( not B )
  if ( count > 1 )
    // release another
    V( delay );
    P( delay );
  else
    V( mutex );
    P( delay );
else count-- ;

if ( count > 0 )
   V( delay );
else V( mutex );
```

```
var first, second: semaphore;
var fcount, scount: integer;

fcount++ ;
if ( scount > 0 ) V( second );
else V( mutex );
P( first );
fcount-- ;
scount++ ;
if ( fcount > 0 ) V( first );
else V( second );
P( second );
scount-- ;

if ( fcount > 0 ) V( first );
else if ( scount > 0 ) V( second );
else V( mutex );
```

Language Support (2): Monitors

- An abstract data type
  - private data
  - public procedures
    - only one procedure can be in the monitor at one time
    - each procedure may have
      - local variables
      - formal parameters
  - condition variables
    - queues of processes
    - *wait*: block on a condition variable
    - *signal*: unblock a waiting process
      - no-op if no process is waiting
- Processes can only invoke the public procedures
  - raises the granularity of atomicity to a single user-defined procedure



Waiting in the Monitor

- Note that the semantics of executing a *wait* in the monitor is that several processes can be waiting "inside" the monitor at any given time but only one is executing
  - wait queues are internal to the monitor
  - there can be multiple wait queues
- Who executes after a signal operation? (say P signals Q)
  - (Hoare semantics) signallee Q continues
    - logically natural since the condition that enabled Q might no longer be true when Q eventually executes
      - P needs to wait for Q to exit the monitor
  - (Mesa semantics) signaller P continues
    - Q is enabled but gets its turn only after P either leaves or executes a *wait*
  - require that the *signal* be the last statement in the procedure
    - advocated by Brinch Hansen (Concurrent Pascal)
    - easy to implement but less powerful than the other two semantics

Use of Monitors: Bounded-buffer

```
type bounded_buffer = monitor

  var buffer: array [0..N] of char;
  var in, out, count: integer;
  var notfull, notempty: condition;

  procedure entry append ...
  procedure entry remove ...

  begin
   in = 0; out = 0; count = 0;
  end;
```

```
procedure entry append(x: char);
  if (count==N) notfull.wait;
  buffer[in] := x;
  in := (in+1) mod N;
  count := count+1;
  notempty.signal;

procedure entry remove(x: char);
  if (count==0) notempty.wait;
  x := buffer[out];
  out := (out+1) mod N;
  count := count-1;
  notfull.signal;
```

Is this solution correct under all monitor semantics? (P signals Q)

- Hoare: Q continues, P suspends .... YES
- Mesa: P continues, Q is put into ready queue .... NO
- Brinch-Hansen: P exits monitor, Q continues ....YES

Use of Monitors: Bounded-buffer (Mesa Semantics)

```
type bounded_buffer = monitor

  var buffer: array [0..N] of char;
  var in, out, count: integer;
  var notfull, notempty: condition;

  procedure entry append ...
  procedure entry remove ...

  begin
    in = 0; out = 0; count = 0;
  end;
```

```
procedure entry append(x: char);
  while (count==N) notfull.wait;
  buffer[in] := x;
  in := (in+1) mod N;
  count := count+1;
  notempty.signal;

procedure entry remove(x: char);
  while (count==0) notempty.wait;
  x := buffer[out];
  out := (out+1) mod N;
  count := count-1;
  notfull.signal;
```

Use of Monitors: Dining Philosophers

- Goal: Solve DP without deadlocks
- Informally:
  - algorithm for Philosopher I

```
dp.pickup(i);
eat;
dp.putdown(i);
```

  - use array to describe state

```
var state: array [0..4] of
   (thinking, hungry,
   eating);
```

  - use array of condition variables to block on when required resources are unavailable

```
var self: array [0..4] of
   condition;
```

- pickup(i)
  - changes state to hungry
  - checks if neighbors are eating
  - if not, grabs chopsticks, and changes state to eating
  - otherwise, waits on self(i)
- putdown(i)
  - checks both neighbors
  - if either is hungry and can proceed, releases him/her

Dining Philosophers using Monitors - 2

```
type dining_philosophers = monitor

  var state: array [0..4] of
   (thinking, hungry, eating);
  var self: array [0..4] of
   condition;

  procedure entry pickup ...
  procedure entry putdown ...
  procedure test ...

  begin
   for i := 0 to 4 do
     state[i] := thinking;
  end;
```

```
procedure entry pickup(i: 0..4);
     state[i] := hungry;
     test(i);
     while ( state[i] != eating )
        self[i].wait;

procedure entry putdown(i: 0..4);
     state[i] := thinking;
     test (ln(i));
     test (rn(i));

procedure test(i: 0..4);
     if (state[ln(i)] != eating and
         state[i] == hungry and
         state(rn(i)) != eating)
       state[i] := eating;
       self[i].signal;
```

Dining Philosophers using Monitors - 3

- What is missing?
  - philosophers cannot deadlock but can starve
    - for example, we can construct timing relationships such that a waiting philosopher will be stuck in the "self" queue forever
  - monitors have to be enhanced with a fair scheduling policy to avoid starvation
    - both at the level of accessing the monitor
    - as well as to regulate "waking-up"' those that are waiting inside
  - how can this be done?
    - use fair enqueue and dequeue policies

Monitors: Other Issues

- Expressibility: Are monitors more/less powerful than semaphores or conditional critical regions?
  - these three constructs are equivalent
    - the same kinds of synchronization problems can be expressed in each
  - the other two can be implemented using any one of the constructs
    - e.g., critical regions and monitors using semaphores
      - we talked about how critical regions can be implemented
      - in Lab 2: you built condition variables using semaphores
        - » this implementation can be extended to build monitors

- Do monitors have any limitations?
  - absence of concurrency within a monitor
    - workarounds introduce all the problems of semaphores
    - monitor procedures will need to be invoked before and after
    - possibility of improper access, deadlock, etc.