(Review) 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( $L$ )
  ```

(Review) Semaphores

- **Semaophores**
  - 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
(Review) 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

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
    - 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)
  
  ```
  DISABLING-INTERRUPTS
  while S <= 0 do wait-a-bit <ENABLE-INTERRUPTS; YIELD CPU>
  S := S-1;
  ENABLE-INTERRUPTS
  ```

  V(S)
  
  ```
  DISABLING-INTERRUPTS
  S := S+1;
  wake up a waiting process
  ENABLE-INTERRUPTS
  ```

- Drawback
  - a process spins on this loop till it gets a chance to 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)
  - WHY?

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;
  if ( S.value < 0 ) then begin
    add process to S.L
    block;
  end;
V(S): S.value := S.value + 1;
  if ( S.value <= 0 ) then begin
    remove P from S.L
    wakeup(P);
  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
    
    ```
    function test-and-set(var target:boolean) boolean;
    begin
    test-and-set := target;
    target := true;
    end;
    ```

- **Swap**
  - atomically exchange the values of given variables X and Y
    ```
    temp = X; X = Y; Y = temp;
    ```
  - can emulate test-and-set
    ```
    function test-and-set(var v: boolean): boolean
    var t := true;
    swap (v, t);
    return t;
    ```

Implementing Locks Using Test-and-Set

LOCK:

\[ L : boolean := false \]

while \( test-and-set(L) \) wait-a-bit

UNLOCK:

\[ L := false \]

Properties of this implementation?

- **Mutual exclusion**
  - first process \( P_i \) entering critical section sets \( L := true \)
  - \( test-and-set \) (from other processes) evaluates to true after this
  - when \( P_i \) exits, \( L \) 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 6.3 for a solution to this problem

Outline

- **Announcements**
  - Lab 2 is due February 14, 2001
  - Questions?

- **Process Synchronization**
  - Review: locks, semaphores, condition variables
  - Implementing the primitives
  - Classical synchronization problems
    - Mutual exclusion
    - Sequencing
    - Producer consumer
    - Readers-writers
    - Dining philosophers
  - An example synchronization problem

\{ Silberschatz/Galvin: Sections 6.4-6.5, Stallings: pages 216-223\}
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

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

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
  P(S);
P() ;
V(S);

  process 2
  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)

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

### Consumer(s)

```c
repeat
  P(full);
  P(mutex);
  // remove item into 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);
V(rsem);
READ
```

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

### Writer(s)

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

Readers-Writers Using Semaphores: Writer-Priority

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

### Reader

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

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

• 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 10 and 11.

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

• 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 (contd.)

• 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 (contd.)

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

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

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

Cashier
- P( payment );
  // accept payment
- P( coord );
  // exit shop
- V( receipt );

A Barbershop Problem (contd.): Mutual Exclusion

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

Customer
- P( max_capacity );
- P( coord );
  // cut hair
- V( coord );
  // wait for customer to leave
- P( leave_b_chair );
  // tell next customer to hop on
- V( barber_chair );
- V( max_capacity );

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

Cashier
- P( payment );
  // accept payment
- P( coord );
  // exit shop
- V( receipt );

A Barbershop Problem (contd.): Bounded Buffer

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

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

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

Cashier
- P( payment );
  // accept payment
- P( coord );
  // exit shop
- V( receipt );

A Barbershop Problem (contd.): Sequencing

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

Customer
- P( max_capacity );
- P( coord );
  // cut hair
- V( coord );
  // wait for customer to leave
- P( leave_b_chair );
  // tell next customer to hop on
- V( barber_chair );
- V( max_capacity );

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

Cashier
- P( payment );
  // accept payment
- P( coord );
  // exit shop
- V( receipt );
A Barbershop Problem (contd.)

- 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

Next Lecture

- Language support for process synchronization
  - Critical regions
  - Monitors
  - Message passing

Readings

- Silberschatz/Galvin: Sections 6.5-6.8