Process Synchronization (cont’d)

February 19, 2003

Outline

- Announcements
  - Lab 2 due Feb 24th
    - Defer extra credit part (priority scheduler) to Lab 3
    - Demos on Feb 26th and 27th
      - You are allowed to update your lab before demos (in case you want to)

- Process synchronization (cont’d)
  - Implementing synchronization primitives
    - Classical synchronization problems
      - Mutual exclusion
      - Sequencing
      - Bounded buffer
      - Readers writers
      - Dining philosophers

[ Silberschatz/Galvin/Gagne: Sections 7.3 – 7.5]

Review: Higher-level Synchronization Primitives

- Locks
  - Boolean variable L, with states **AVAILABLE** and **BUSY**
  - **LOCK**: while ( L != AVAILABLE ) wait-a-bit
    - L = BUSY;
  - **UNLOCK**: L = AVAILABLE; wake up a locked process (if any)

- Semaphores
  - Integer variable S
  - **WAIT**: while S <= 0 do wait-a-bit;
    - S := S-1;
  - **SIGNAL**: S := S + 1; wake up a waiting process (if any)

- Condition variables
  - Process queue, accessed within a critical section
  - **WAIT**: put process on queue (suspend)
  - **SIGNAL**: wake up one process from queue (resume)
    - Variants: Mesa-style (signalling process continues), Hoare-style

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
  \[\text{P}(S)\]
  \[
  \begin{align*}
  \text{DISABLE-INTERRUPTS} \\
  \text{while } S \leq 0 \text{ do wait-a-bit} \\
  \quad [\text{ENABLE-INTERRUPTS}; \text{YIELD}; \text{DISABLE-INTERRUPTS}] \\
  S := S-1; \\
  \text{ENABLE-INTERRUPTS}
  \end{align*}
  \]
  \[\text{V}(S)\]
  \[
  \begin{align*}
  \text{DISABLE-INTERRUPTS} \\
  S := S+1; \\
  \text{[ wake up a waiting process ]} \\
  \text{ENABLE-INTERRUPTS}
  \end{align*}
  \]

- 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)
  - 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

  ```
  \[
  \text{type semaphore = record} \\
  \quad \text{value: integer;} \\
  \quad \text{L: list of processes;} \\
  \quad \text{end;} \\
  \]
  \[
  \text{P}(S) : S.\text{value} := S.\text{value} - 1; \\
  \quad \text{if } (S.\text{value} < 0 ) \text{ then begin} \\
  \quad \quad \text{add process to S.L} \\
  \quad \quad \text{block;} \\
  \quad \text{end;} \\
  \]
  \[
  \text{V}(S) : S.\text{value} := S.\text{value} + 1; \\
  \quad \text{if } (S.\text{value} >= 0 ) \text{ then begin} \\
  \quad \quad \text{remove P from S.L} \\
  \quad \quad \text{wakeup(P);} \\
  \quad \text{end;} \\
  \]

  - 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
    ```
    \[
    \begin{align*}
    \text{boolean Test-and-set( boolean &target ) } \{ \\
    \quad \text{boolean rv = target;} \\
    \quad \text{target = true;} \\
    \quad \text{return rv;} \\
    \}
    \end{align*}
    \]

- Swap
  - atomically exchange the values of given variables X and Y
    ```
    \[
    \begin{align*}
    \text{temp} = X; \quad X = Y; \quad Y = \text{temp;} \\
    \end{align*}
    \]
  - can emulate test-and-set
    ```
    \[
    \begin{align*}
    \text{boolean Test-and-set( boolean &target ) } \{ \\
    \quad \text{boolean t := true;} \\
    \quad \text{swap (target, t);} \\
    \quad \text{return t;} \\
    \}
    \end{align*}
    \]
Implementing Locks Using Test-and-Set

**LOCK:**

\[
\text{lock : boolean := false}
\]

\[
\text{while Test-and-set(lock) wait-a-bit}
\]

**UNLOCK**

\[
\text{lock := false}
\]

- **Properties of this implementation**
  - Mutual exclusion?
    - First process \( P_i \) entering critical section sets \( \text{lock := true} \)
    - Test-and-set (from other processes) evaluates to true after this
    - When \( P_i \) exits, \( \text{lock} \) is set to false, so the next process \( P_j \) to execute the instruction will find \( \text{test-and-set = false} \) 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 7.3 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
  - Another possibility: interrupts never “force” a context switch
    - They just set flags, or wake up processes
  - Primitives
    - \( \text{sleep} \) (address);
    - \( \text{wake_up} \) (address); -- wakes up all processes sleeping on address
  - Typical code
    
    \[
    \begin{align*}
    \text{ENTRY:} & \quad \text{while (locked) sleep(bufaddr);} \\
    & \quad \text{locked = true;} \\
    \text{EXIT:} & \quad \text{locked = false; wake_up (bufaddr);} \\
    \end{align*}
    \]

Synchronization Primitives in Real OSes (cont’d)

- **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
    - Condition variables
      - Wait and signal
    - Reader-writer locks
      - For frequent mostly read-only accesses

Outline

- **Announcements**
  - Lab 2 due Feb 24th
  - Defer extra credit part (priority scheduler) to Lab 3
  - Demos on Feb 26th and 27th
    - Will allow you to update your lab before demos (in case you want to)

- **Process synchronization (cont’d)**
  - Implementing synchronization primitives
    - Classical synchronization problems
      - Mutual exclusion
      - Sequencing
      - Bounded buffer
      - Readers writers
      - Dining philosophers

[Silberschatz/Galvin/Gagne: Sections 7.3 – 7.5]
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 (cont’d)

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
  \[
  P(S) ; \\
  \text{CRITICAL SECTION} \\
  V(S) ;
  \]

- Sequencing: Semaphore initialized to 0
  \[
  P(S) ; \\
  V(S) ; \\
  P(S) ; \\
  V(S) ; \\
  P(S) ; \\
  A() ;
  \]
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);
P(wsem);
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

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

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

```
Reader

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

Writer

```
P(y);
wcount := wcount + 1;
if (wcount == 1) then P(rsem);
V(y);
P(wsem);
WRITE
V(wsem);
P(wsem);
P(y);
wcount := wcount - 1;
if (wcount == 0) then V(rsem);
V(y);
```
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 (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)
  - `finished` (initial value = 0)
  - `leave_b_chair` (initial value = 0)
- Paying and receiving
  - `payment` (initial value = 0)
  - `receipt` (initial value = 0)
- Coordinating barber functions
  - `coord` (initial value = 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 ); // tell next customer to hop on
P( finished ); // tell next customer to hop on
V( barber_chair );
```

Barber
```
P( cust_ready ); // cut hair
P( coord ); // cut hair
V( coord ); // 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 ); // accept payment
V( receipt ); // exit shop
```

A Barbershop Problem (cont’d): Mutual Exclusion

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 ); // tell next customer to hop on
P( finished ); // tell next customer to hop on
V( barber_chair );
```

Barber
```
P( cust_ready ); // cut hair
P( coord ); // cut hair
V( coord ); // 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 ); // accept payment
V( receipt ); // exit shop
```

A Barbershop Problem (cont’d): Bounded Buffer

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 ); // tell next customer to hop on
P( finished ); // tell next customer to hop on
V( barber_chair );
```

Barber
```
P( cust_ready ); // cut hair
P( coord ); // cut hair
V( coord ); // 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 ); // accept payment
V( receipt ); // exit shop
```
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

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