Computer Systems Organization II (Honors)  
(Introductory Operating Systems)

Lecture 8  
Classical Process Synchronization Problems

February 14, 2000

Outline

• Announcements
  – Writeup 2 will be handed out on Feb 16th, and will be due on Feb 28th
  – Lab 2 is due Feb 16th: TA will send out demo times
  – Questions?

• Process Synchronization
  – Review: locks, semaphores, condition variables
  – 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]

(Review) Locks (Mutexes)

• Locks (also known as Mutexes)
  – 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) 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
(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

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 (contd.)

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
   - 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 intialized to 0

<table>
<thead>
<tr>
<th>process 1</th>
<th>process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P(S); ]</td>
<td>[ B(); V(S); ]</td>
</tr>
<tr>
<td>[ A(); ]</td>
<td>[ ]</td>
</tr>
</tbody>
</table>

## 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

\[
\text{repeat}
\]

\[
\begin{align*}
&\text{// produce an item in nextp} \\
&\quad P(\text{empty}); \\
&\quad P(\text{mutex}); \quad \text{// add nextp to buffer} \\
&\quad V(\text{mutex}); \\
&\quad V(\text{full}); \\
&\quad \text{until false;}
\end{align*}
\]

### Consumer

\[
\text{repeat}
\]

\[
\begin{align*}
&\quad P(\text{full}); \\
&\quad P(\text{mutex}); \quad \text{// remove item into nextc} \\
&\quad V(\text{mutex}); \\
&\quad V(\text{empty}); \quad \text{// consume item in nextc} \\
&\quad \text{until false;}
\end{align*}
\]

## Readers-Writers Using Semaphores

### Reader

\[
\begin{align*}
&\quad P(x); \\
&\quad rcount := rcount + 1; \quad \text{if (rcount == 1) then } P(\text{wsem}); \\
&\quad V(x); \\
&\quad \text{READ} \\
&\quad P(x); \\
&\quad rcount := rcount - 1; \quad \text{if (rcount == 0) then } V(\text{wsem}); \\
&\quad V(x);
\end{align*}
\]

### Writer

\[
\begin{align*}
&\quad P(x); \\
&\quad rcount := rcount + 1; \quad \text{if (rcount == 1) then } P(\text{wsem}); \\
&\quad V(x); \\
&\quad \text{WRITE} \\
&\quad P(wsem); \\
&\quad \text{READ} \\
&\quad P(x); \\
&\quad rcount := rcount - 1; \quad \text{if (rcount == 0) then } V(\text{wsem}); \\
&\quad V(x);
\end{align*}
\]

Stream of readers can starve writers

Readers can queue up preventing a waiting writer from setting rsem

## Readers-Writers Using Semaphores: Writer-Priority

### Reader

\[
\begin{align*}
&\quad P(x); \\
&\quad rcount := rcount + 1; \quad \text{if (rcount == 1) then } P(\text{wsem}); \\
&\quad V(x); \\
&\quad \text{READ} \\
&\quad P(x); \\
&\quad rcount := rcount - 1; \quad \text{if (rcount == 0) then } V(\text{wsem}); \\
&\quad V(x);
\end{align*}
\]

### Writer

\[
\begin{align*}
&\quad P(x); \\
&\quad wcount := wcount + 1; \quad \text{if (wcount == 1) then } P(\text{rsem}); \\
&\quad V(x); \\
&\quad \text{WRITE} \\
&\quad P(\text{rsem}); \\
&\quad \text{READ} \\
&\quad P(x); \\
&\quad wcount := wcount - 1; \quad \text{if (wcount == 0) then } V(\text{rsem}); \\
&\quad V(x);
\end{align*}
\]

Can release either waiting readers or writers

Writers can queue up preventing a waiting reader from setting rsem
Readers-Writers Using Semaphores: Writer-Priority (2)

**Reader**

\[
P(z); \quad P(rsem); \quad P(x); \quad \text{rcount} := \text{rcount} + 1; \quad \text{if (rcount == 1) then } P(wsem); \quad V(x); \quad V(rsem); \quad V(z); \quad \text{READ}
\]

\[
P(x); \quad \text{rcount} := \text{rcount} - 1; \quad \text{if (rcount == 0) then } V(wsem); \quad V(x);
\]

**Writer**

\[
P(y); \quad \text{wcount} := \text{wcount} + 1; \quad \text{if (wcount == 1) then } P(rsem); \quad V(y); \quad P(wsem); \quad \text{WRITE}
\]

\[
P(y); \quad \text{wcount} := \text{wcount} - 1; \quad \text{if (wcount == 0) then } V(rsem); \quad V(y);
\]

---

Dining Philosophers Using Semaphores

**Philosopher_i**

\[
P(\text{chopstick}[i]); \quad P(\text{chopstick}[i+1 \mod 5]); \quad \text{EAT}
\]

\[
V(\text{chopstick}[i]); \quad V(\text{chopstick}[i+1 \mod 5]);
\]

\[
P(y);
\]

\[
P(wsem); \quad \text{WRITE}
\]

\[
V(y);
\]

**Philosopher_{(j=i+1 \mod 5)}**

\[
P(\text{chopstick}[j]); \quad P(\text{chopstick}[j+1 \mod 5]); \quad \text{EAT}
\]

\[
V(\text{chopstick}[j]); \quad V(\text{chopstick}[j+1 \mod 5]);
\]

\[
P(y);
\]

\[
P(wsem); \quad \text{WRITE}
\]

\[
V(y);
\]

---

**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.

---

Dining Philosophers Using Semaphores - 2

**Philosopher_{[\text{even } i]}**

\[
P(\text{chopstick}[i]); \quad P(\text{chopstick}[i+1 \mod 5]); \quad \text{EAT}
\]

\[
V(\text{chopstick}[i]); \quad V(\text{chopstick}[i+1 \mod 5]);
\]

**Philosopher_{[\text{odd } i]}**

\[
P(\text{chopstick}[i+1 \mod 5]); \quad P(\text{chopstick}[i]); \quad \text{EAT}
\]

\[
V(\text{chopstick}[i+1 \mod 5]); \quad V(\text{chopstick}[i]);
\]

- 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

Customer
- P( max_capacity );
- P( sofa );
- P( barber_chair );
- V( cust_ready );
- P( finished );
- V( leave_b_chair );
- V( payment );
- P( receipt );

Barber
- P( cust_ready );
- P( coord );
- P( sofa );
- P( barber_chair );
- P( finished );
- V( cust_ready );
- V( leave_b_chair );
- V( payment );
- P( receipt );

Cashier
- P( payment );
- P( coord );
- P( receipt );
- V( coord );
- V( receipt );
A Barbershop Problem (contd.): Sequencing

- Shop and sofa capacity
  - \texttt{max_capacity} \(:= 20\)
  - \texttt{sofa} \(:= 4\)

- Barber chair capacity
  - \texttt{barber_chair} \(:= 3\)

- Ensuring customers are in barber chair
  - \texttt{cust_ready} \(:= 0\)
  - \texttt{finished} \(:= 0\)
  - \texttt{leave_b_chair} \(:= 0\)

- Paying and receiving
  - \texttt{payment} \(:= 0\)
  - \texttt{receipt} \(:= 0\)

- Coordinating barber functions
  - \texttt{coord} \(:= 0\)

Next Lecture

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

Readings

- Silberschatz/Galvin: Sections 6.5-6.8