Lecture 7
Classical Synchronization Problems

February 20, 2002

Outline
- Announcements
  - Lab 2 due Feb 25th, demos on Feb 27th, 28th
    - Defer extra credit part (priority scheduler) to Lab 3
- Process synchronization
  - Classical synchronization problems
    - Mutual exclusion, Sequencing, Producer consumer,
    - Readers-writers, Dining philosophers
  - A larger synchronization problem:
    - (if time permits) Language support for synchronization
      - Critical regions
      - Monitors
      - Message passing

[ Silberschatz/Galvin/Gagne: Sections 7.5 – 7.8]
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(y);
wcount := wcount + 1;
if (wcount == 1) then P(rsem);
V(y);
READ
P(wsem);
WRITE
V(wsem);
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);
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)
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.

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

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 := 20
  - sofa := 4

- Barber chair capacity
  - barber_chair := 3

- Ensuring customers are in barber chair
  - cut_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 );
// in barber chair
V( cut_ready );
P( finished );
// leave barber chair
V( leave_b_chair );
// pay
V( payment );
P( coord );
// accept payment
V( receipt );
V( max_capacity );
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

Outline

- Announcements
  - Lab 2 due Feb 25th, demos on Feb 27th, 28th
  - Defer extra credit part (priority scheduler) to Lab 3

- Process synchronization
  - Classical synchronization problems
    - Mutual exclusion, Sequencing, Producer consumer,
    - Readers-writers, Dining philosophers
  - A larger synchronization problem
    - (if time permits) Language support for synchronization
      - Critical regions
      - Monitors
      - Message passing

{ Silberschatz/Galvin/Gagne: Sections 7.5 – 7.8}

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
  - why?

- Consider sequencing between three processes
  - P1, P2, P3, P1, P2, P3, ...

\[
\begin{array}{ccc}
P_1 & P_2 & P_3 \\
\begin{array}{c}
P(\text{sem}_1); \quad P(\text{sem}_2); \quad P(\text{sem}_3); \\
/ / \text{do stuff} \quad / / \text{do stuff} \quad / / \text{do stuff} \\
V(\text{sem}_1); \quad V(\text{sem}_2); \quad V(\text{sem}_3); \\
\end{array}
\end{array}
\]

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

Limitations of Semaphores (contd.)

- 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
    - V(seq); instead of
      - P(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)
  - Concurrent Pascal
  - 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

- 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

```
Next Lecture (February 25th)

- Language support for synchronization (cont’d)
  - Monitors
  - Message passing

- CPU Scheduling

- Reading
  - Silberschatz/Galvin/Gagne: Chapter 6