Lecture 8
Process Synchronization (cont’d)
Language Support for Synchronization

February 24, 2003

Outline

- Announcements
  - Lab 2 due today; Demos on Feb 26th and 27th
    - You are allowed to update your lab before demos (in case you want to)

- Process synchronization (cont’d)
  - Classical synchronization problems (cont’d)
    - Readers writers
    - Dining philosophers
  - A large synchronization problem
  - Language support for synchronization
    - Critical regions
      (if time permits)
    - Monitors

[ Silberschatz/Galvin/Gagne: Sections 7.5 – 7.8]

2/27/03

(Review)
Readers-Writers Using Semaphores
To allow multiple readers, synchronize only the first/last reader with writers

Reader(s)

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

\[
READ
\]

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

Writer(s)

\[
P(wsem); \quad P(x);
\]

\[
\text{stream of readers can starve writers}
\]

\[
\text{can release either waiting readers or writers}
\]

2/27/03

(Review)
Readers-Writers Using Semaphores: Writer-Priority
Have a writer block out subsequent readers (same as readers block out writers)

Reader

\[
P(rsem); \quad P(x);
\]

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

\[
V(rsem); \quad P(wsem);
\]

\[
V(wsem); \quad P(y);
\]

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

\[
V(y);
\]

\[
\text{readers can queue up preventing a waiting writer from setting rsem}
\]

Writer

\[
P(y);
\]

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

\[
V(y);
\]

2/27/03
Readers-Writers Using Semaphores: Writer-Priority (2)

**Reader**

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

**Writer**

- `P(y);`
- `wcount := wcount + 1;`
- `if (wcount == 1) then P(wsem);`
- `V(y); V(wsem);`
- `WRITE`
- `V(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 10 and 11.

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


- 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

Customer
P( max_capacity );
// enter shop
P( sofa );
// sit on sofa
P( barber_chair );
// cut hair
V( coord );
// get up from sofa
V( finished );
// sit in barber chair
V( cust_ready );
// wait for customer to leave
V( leave_b_chair );
// tell next customer to hop on
V( barber_chair );

Barber
P( cust_ready );
P( coord );
// cut hair
V( coord );
V( finished );
// sit on sofa
P( sofa );
// get up from sofa
V( finished );
// leave_b_chair (
V( cust_ready );
// tell next customer to hop on
V( barber_chair );

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

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)

Customer
P( max_capacity );
// enter shop
P( sofa );
// sit on sofa
P( barber_chair );
// cut hair
V( coord );
// sit in barber chair
V( cust_ready );
// get up from sofa
V( finished );
// leave_b_chair (initial value = 0)
V( cust_ready );
// tell next customer to hop on
V( barber_chair );

Barber
P( cust_ready );
P( coord );
// cut hair
V( coord );
V( finished );
// sit on sofa
P( sofa );
// get up from sofa
V( finished );
// leave_b_chair (initial value = 0)
V( cust_ready );
// tell next customer to hop on
V( barber_chair );

Cashier
P( payment );
P( coord );
// accept payment
V( coord );
V( receipt );
// exit shop
V( max_capacity );
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 );
  // cut hair
  P( coord );
  // wait for customer to leave
  V( finished );
  // tell next customer to hop on
  V( barber_chair );

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

4
1
2

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 today; Demos on Feb 26th and 27th
  - You are allowed to update your lab before demos (in case you want to)
- Process synchronization (cont’d)
  - Classical synchronization problems (cont’d)
    - Readers writers
    - Dining philosophers
  - A large synchronization problem
  - Language support for synchronization
    - Critical regions
    - (if time permits)
    - Monitors

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
  - P_1, P_2, P_3, P_4, P_5, …
    \[
    \begin{align*}
    P_1 & : & P( \text{sem}_1); & \text{P( sem}_1); & P( \text{sem}_1); & \text{P( sem}_1); & \\
    & & // do stuff & // do stuff & // do stuff &\
    P_2 & : & V( \text{sem}_1); & V( \text{sem}_1); & V( \text{sem}_1); &\
    & & // do stuff & // do stuff & // do stuff &\
    \end{align*}
    \]
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
      \[ V( mutex ); \text{CODE;} P( mutex ); \text{instead of} \]
      \[ P( mutex ); \text{CODE;} V( mutex ); \]

- can cause deadlock
  \[ P( seq ); \text{instead of} \]
  \[ V( seq ); \]

- 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
    \[
    \text{var } v: \text{shared } t; \\
    \text{region } v \text{ when } B \text{ do } S;
    \]
    \[
    \text{\begin{itemize}
    \item \( v \) is shared and of type \( t \)
    \item \( B \) is a Boolean expression
    \item \( S \) is a statement
    \end{itemize}}\

- 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

- 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

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

```plaintext
var delay: semaphore;
var count: integer;
count++;
if (count > 1) V(mutex);
P(delay);
else V(mutex);
P(delay);
```

```plaintext
var first, second: semaphore;
var fcount, scount: integer;
fcount++;
if (fcount > 0) V(second);
else V(mutex);
P(first);
fcount--;
scount++;
if (fcount > 0) V(first);
else V(mutex);
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) signaler 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

```plaintext
type bounded_buffer = monitor

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

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

```plaintext
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)

```plaintext
type bounded_buffer = monitor
  var buffer: array [0..N] of char;
  var in, out, count: integer;
  var notfull, notempty: condition;
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;

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

Use of Monitors: Dining Philosophers

- Goal: Solve DP without deadlocks
- Informally:
  - algorithm for Philosopher I
    ```plaintext
dp.pickup(i);
dp.eat();
dp.putdown(i);
```
  - use array to describe state
    ```plaintext
    var state: array [0..4] of (thinking, hungry, eating);
    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

```plaintext
type dining_philosophers = monitor
  var state: array [0..4] of (thinking, hungry, eating);
  var self: array [0..4] of condition;
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.