Lecture - X
Process Synchronization & Deadlocks

Tevfik Koşar

University at Buffalo
October 4th, 2011

Roadmap

• Classic Problems of Synchronization
  - Readers and Writers Problem
  - Dining-Philosophers Problem
  - Sleeping Barber Problem
• Deadlocks

Readers-Writers Problem

• Multiple Readers and writers concurrently accessing the same database.
• Multiple Readers accessing at the same time --> OK
• When there is a Writer accessing, there should be no other processes accessing at the same time.

Readers-Writers Problem (Cont.)

• The structure of a writer process
  ```
  do {
    wait (wrt) ;
    // writing is performed
    signal (wrt) ;
  } while (true)
  ```

Readers-Writers Problem

• The structure of a reader process
  ```
  do {
    wait (mutex) ;
    readercount ++ ;
    if (readercount == 1) wait (wrt) ;
    signal (mutex)
    // reading is performed
    wait (mutex) ;
    readercount -- ;
    if readercount == 0) signal (wrt) ;
    signal (mutex) ;
  } while (true)
  ```

Dining Philosophers Problem

• Five philosophers spend their time eating and thinking.
• They are sitting in front of a round table with spaghetti served.
• There are five plates at the table and five chopsticks set between the plates.
• Eating the spaghetti requires the use of two chopsticks which the philosophers pick up one at a time.
• Philosophers do not talk to each other.
• Semaphore chopstick [5] initialized to 1

• The structure of a writer process
**Dining-Philosophers Problem (Cont.)**

- The structure of Philosopher $i$:

```c
Do {
    wait (chopstick[i]);
    wait (chopstick[(i + 1) % 5]);

    // eat
    signal (chopstick[i]);
    signal (chopstick[(i + 1) % 5]);

    // think
} while (true);
```

**To Prevent Deadlock**

- Ensures mutual exclusion, but does not prevent deadlock
- Allow philosopher to pick up her chopsticks only if both chopsticks are available (i.e., in critical section)
- Use an asymmetric solution: an odd philosopher picks up first her left chopstick and then her right chopstick; and vice versa

**Problems with Semaphores**

- Wrong use of semaphore operations:
  - Semaphores $A$ and $B$, initialized to 1
    ```
    P1 b(1); P1 b(1)
    wait (A); wait(B) wait (B); wait (A)
    ```
    ➔ Deadlock
  - signal (mutex) ... wait (mutex)
    ➔ violation of mutual exclusion
  - wait (mutex) ... wait (mutex)
    ➔ Deadlock
  - Omitting of wait (mutex) or signal (mutex) (or both)
    ➔ violation of mutual exclusion or deadlock

**Semaphores**

- Inadequate in dealing with deadlocks
- Do not protect the programmer from the easy mistakes of taking a semaphore that is already held by the same process, and forgetting to release a semaphore that has been taken
- Mostly used in low level code, e.g., operating systems
- The trend in programming language development, though, is towards more structured forms of synchronization, such as monitors

**Monitors**

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```c
monitor monitor-name {
    // shared variable declarations
    procedure P1 (...) {
        ...
    }

    procedure P2 (...) {
        ...
    }

    initialization code (...){ ...
    }
}
```

- A monitor procedure takes the lock before doing anything else, and holds it until it either finishes or waits for a condition

**Monitor - Example**

As a simple example, consider a monitor for performing transactions on a bank account.

```c
monitor account {
    int balance := 0;

    function withdraw(int amount) {
        if amount < 0 then error "Amount may not be negative"
        else if balance < amount then error "Insufficient funds"
        else balance := balance - amount
    }

    function deposit(int amount) {
        if amount < 0 then error "Amount may not be negative"
        else balance := balance + amount
    }
}
```
Condition Variables

- Provide additional synchronization mechanism
- condition x, y;

- Two operations on a condition variable:
  - x.wait() - a process invoking this operation is suspended
  - x.signal() - resumes one of processes (if any) that invoked x.wait()

If no process suspended, x.signal() operation has no effect.

Solution to Dining Philosophers using Monitors

```c
monitor DP {
  enum { THINKING, HUNGRY, EATING } state[5];
  condition self[5]; // to delay philosopher when he is hungry but unable to get chopsticks

  initialization_code() {
    for (int i = 0; i < 5; ++i)
      state[i] = THINKING;
  }

  void pickup(int i) {
    state[i] = HUNGRY;
    test(i); // only if both neighbors are not eating
    if (state[i] != EATING) self[i].wait;
  }

  void putdown(int i) {
    state[i] = THINKING;
    test((i + 4) % 5);
    test((i + 1) % 5);
  }
}
```

Solution to Sleeping Barber Problem

- Based upon a hypothetical barber shop with one barber, one barber chair, and a number of chairs for waiting customers
- When there are no customers, the barber sits in his chair and sleeps
- As soon as a customer arrives, he either awakens the barber or, if the barber is cutting someone else's hair, sits down in one of the vacant chairs
- If all of the chairs are occupied, the newly arrived customer simply leaves

```c
void test(int i) {
  if ((state[i] == HUNGRY) 
      && (state[(i + 1) % 5] != EATING) 
      && (state[(i + 4) % 5] != EATING)) {
    state[i] = EATING;
    self[i].signal();
  }
}

void putdown(int i) {
  state[i] = THINKING;
  test((i + 4) % 5);
  test((i + 1) % 5);
}
```

Implementation:

- Semaphore Customers
- Semaphore Barber
- Semaphore accessSeats (mutex)
- int NumberOfFreeSeats

The Barber(Thread):

```c
while(true) // runs in an infinite loop
{ // customer
  Customers.wait() // tries to acquire a customer - if none is available he's going to sleep
  accessSeats.wait() // at this time he has been awakened - want to modify the number of available seats
  NumberOfFreeSeats++ // / one chair gets free
  Barber.signal() // the barber is ready to cut
  accessSeats.signal() // we don't need the lock on the chairs anymore // here the barber is cutting hair
}
```
The Customer(Thread):

while (notCut) // as long as the customer is not cut
{
    accessSeats.wait() // tries to get access to the chairs
    if (NumberOfFreeSeats>0) { // if there are any free seats
        NumberOfFreeSeats -- // sitting down on a chair
        Customers.signal() // notify the barber, who's waiting that there is
        a customer
        accessSeats.signal() // don't need to lock the chairs anymore
        Barber.wait() // now it's this customer's turn, but wait if the barber
        is busy
        notCut = false
    } else // there are no free seats // tough luck
    accessSeats.signal() // but don't forget to release the lock on the
    seats }

The Deadlock Problem

- A set of blocked processes each holding a resource and
  waiting to acquire a resource held by another process
  in the set.
- Example
  - System has 2 disk drives.
  - $P_1$ and $P_2$ each hold one disk drive and each needs another one.
- Example
  - semaphores $A$ and $B$, initialized to 1
    
    \[
    \begin{align*}
    P_0 & : P_1 \\
    \text{wait}(A); & \text{wait}(B) \\
    \text{wait}(B); & \text{wait}(A)
    \end{align*}
    \]

Deadlock vs Starvation

- Deadlock - two or more processes are waiting indefinitely for an
  event that can be caused by only one of the waiting processes

- Starvation - indefinite blocking. A process may never be removed
  from the semaphore queue in which it is suspended.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

1. Mutual exclusion: nonshared resources; only one process at a time can use a specific resource
2. Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
3. No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task

Bridge Crossing Example

- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
Deadlock Characterization (cont.)

Deadlock can arise if four conditions hold simultaneously.

4. **Circular wait**: there exists a set \( \{P_0, P_1, \ldots, P_n\} \) of waiting processes such that 
   - \( P_0 \) is waiting for a resource that is held by \( P_1 \),
   - \( P_1 \) is waiting for a resource that is held by \( P_2 \),
   - \( \ldots \),
   - \( P_{n-1} \) is waiting for a resource that is held by \( P_n \), and
   - \( P_n \) is waiting for a resource that is held by \( P_0 \).

Summary

- Classic Problems of Synchronization
  - Readers and Writers Problem
  - Dining-Philosophers Problem
  - Sleeping Barber Problem
- Deadlocks

Next Lecture: Deadlocks - II

Acknowledgements

- “Modern Operating Systems” book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from UNR