CSCI 3753
Operating Systems

Classic Synchronization Problems

Lecture Notes By

Shivakant Mishra

Computer Science, CU-Boulder

Last Update: 10/03/17

Classic Synchronization Problems

- Bounded Buffer Producer-Consumer Problem
  - We have already seen this
- Readers-Writers Problem
- Dining Philosophers Problem
- These are not just abstract problems
  - They are representative of several classes of synchronization problems commonly encountered in the real world when trying to synchronize access to shared resources among multiple processes or threads

Readers writers problem

- A database is accessed by two types of processes: reader processes and writer processes
- Readers only read information from the database
- Writers modify the database

- Constraints
  - Writers must have exclusive access to the database
  - Multiple readers can access the database concurrently

Readers-Writers Problem: First Attempt

Semaphore mutex = 1;

```
Reader( )
{
   wait(mutex);
      read database
   signal(mutex);
}
```

```
Writer( )
{
   wait(mutex);
      write database
   signal(mutex);
}
```

Exclusive access to the writer processes is provided

BUT: No concurrency among reader processes

Readers-Writers Problem: Second Attempt

int rc = 0; /* Number of readers in the database */

Semaphore db = 1; /* controls access to database for writers */

```
Reader ()
    if (rc == 0) wait(db);
    rc++;

        read database

    rc--;
    if (rc == 0) signal(db);
```

```
Writer( )
{
   wait(db);
      write database
   signal(db);
}
```

Problem: Race condition in accessing rc

Readers-Writers Problem: Third Attempt

int rc = 0; /* Number of readers in the database */

Semaphore db = 1; /* controls access to database for writers */

Semaphore mutex = 1; /* controls access to variable rc */

```
Reader ()
    wait(mutex);
    if (rc == 0) wait(db);
    rc++;
    signal(mutex);

        read database

    wait(mutex);
    rc--;
    if (rc == 0) signal(db);
    signal(mutex);
```

```
Writer( )
{
    wait(db);
        write database
    signal(db);
}
```

Readers-Writers Problem: Third Attempt

- Semaphore mutex is used for mutual exclusion to update rc
- Semaphore db is used for exclusive database access for writer processes
- Multiple reader processes can access the database concurrently if there is no writer process

Problem: What happens to a writer if readers keep coming to read the database?

  - Writer starvation
  - Readers have priority over writers

Exercise

- Write a solution that gives preference to writer processes

- Write a solution that is fair to both readers and writers

Dining Philosophers Problem

- The most famous synchronization problem.
- Represents a situation that can occur in large community of processes that share a large pool of resources.

Dining Philosophers Problem

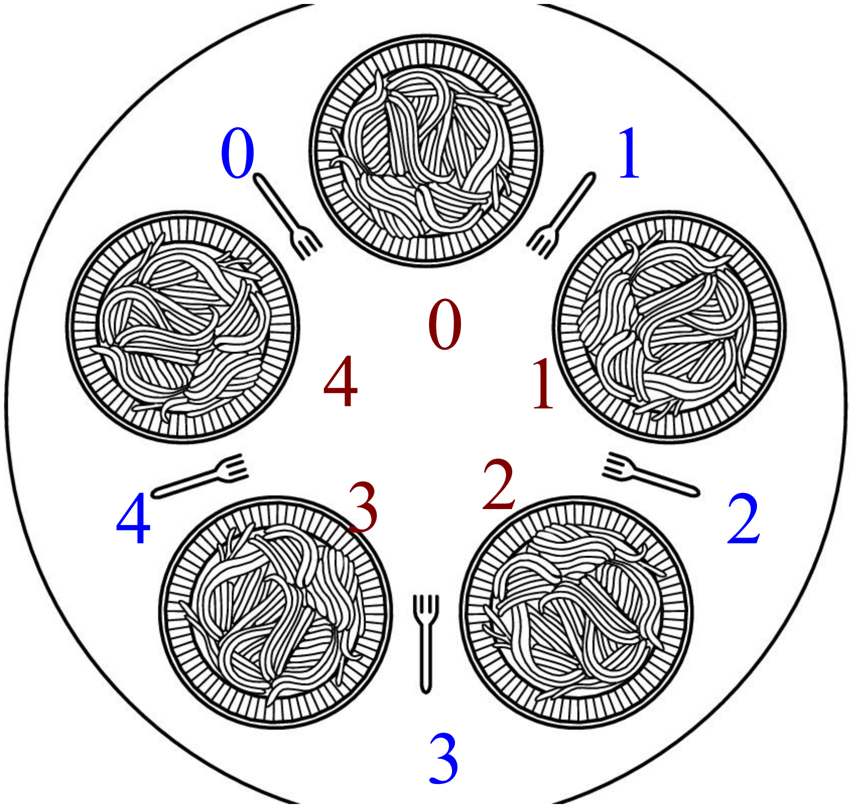
Five philosophers sit around a round dining table. A plate of noodles is placed in front of each philosopher, and a fork is placed between any two adjacent plates

A philosopher needs two forks to eat

All philosophers alternate between two activities: thinking and eating

Write a synchronization program that allows all five philosophers to run their lives

- Deadlock free
- Starvation free



What is the problem here?

Adjacent philosophers cannot eat at the same time

If a philosopher is eating and a neighbor is hungry, he/she has to wait until the first philosopher is done eating

What resource(s) is shared

Forks

Dining Philosophers: First Attempt

```
semaphore fork[5] = (1,1,1,1,1);

philosopher(int i) {
  while(TRUE) {
    // Think
    wait(fork[i]);
    wait(fork[(i+1) mod 5]);
        // EAT
    signal(fork[(i+1) mod 5]);
    signal(fork[i]);
  }
}
```

Problem

- On a fateful day, all philosophers decide to eat at the same time
- All philosophers pick up their right fork
- All philosophers now wait *forever* for their left fork to become available
  - ... *and die of starvation*
- Deadlock

Dining Philosophers: Second Attempt

```
semaphore fork[5] = (1,1,1,1,1);
semaphore mutex = 1;
philosopher(int i) {
  while(TRUE) {
    // Think
    wait(mutex);
    wait(fork[i]);
    wait(fork[(i+1) mod 5]);
    signal(mutex);
    // Eat
    signal(fork[(i+1) mod 5]);
    signal(fork[i]);
  }
}
```

This solution doesn't suffer from deadlock

Problem: May not allow two non-adjacent philosophers to eat at the same time

Dining Philosophers: Third Attempt

After picking up a fork, if a philosopher finds that the other fork is not available, she keeps down the fork, waits for some time, and tries again

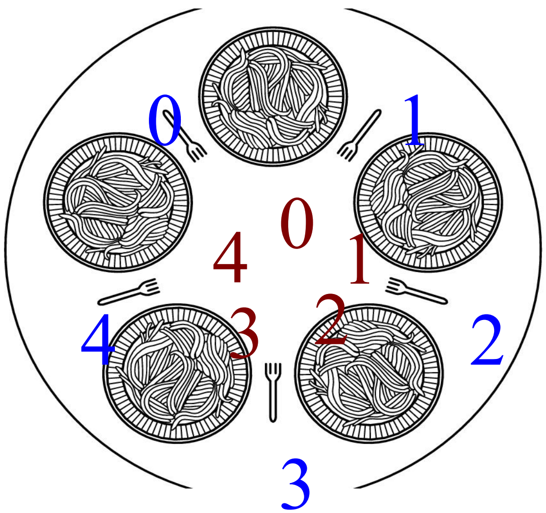
- Does this solution work?
- Starvation
- A deadlock-free solution is not necessarily starvation-free

Announcements

- Programming assignment two
  - Error in line 50 of the PA2_simple_char_driver.c
  - Return type of simple_char_driver_seek ( ) should be loff_t
  - Already corrected on Moodle
- Quiz during recitation on Friday, October 06
- Readings: Chapters 1, 2, 3, 4, 5, 6 and 13

Recap…

- Dining Philosophers problem
- Solution 1: each philosopher picks right fork and then the left fork
  - Potential for deadlock
- Solution 2: each philosopher picks the two forks in a critical section
  - May not allow a philosopher to eat even if both forks are available
- Solution 3: a philosopher puts down a fork if the other fork is not available, and retries
  - Starvation



Dining Philosophers: Some possible solutions

- Allow at most 4 philosophers at the same table when there are 5 resources
- Odd philosophers pick first left then right, while even philosophers pick first right then left
- Allow a philosopher to pick up forks *only if both are free.* This requires protection of critical sections to test if both forks are free before grabbing them.
  - We'll see this solution next using monitors
  - Also, there is a construct called an AND semaphore

Higher Level Synchronization Primitives

- Semaphores can result in deadlock due to programming errors
  - Forgot a wait() or signal(), or mis-ordered them, or duplicated them
- Relatively simple problems, such as the dining philosophers problem, can be very difficult to solve using low level constructs like semaphores
- Higher level synchronization primitives
  - AND synchronization
  - Events
  - Critical Conditional Regions
  - Condition Variables: *We will study this in recitation*
  - Monitors: *We will study this*
  - many others…

Monitors

- Abstract data type (similar to C++ classes)
  - Monitors are found in high-level programming languages like Java and C#
- A monitor is a collection of functions, variables, and data structures
- Processes can access monitor variables only by calling functions in the monitor
- Each function in the monitor can only access variables declared locally within the monitor and its parameters
- At most one process may be active at any time in a monitor

- monitor *monitor_name* {

```
// shared local variables

function f1(...) {
...
}
...
function fN(...) {
...
}
init_code(...) {
...
}
}
```

Monitors and Condition Variables

- While the above definition of a monitor achieves mutual exclusion (hiding wait() and signal() from user), it loses the ability that semaphores had to enforce order
  - i.e. wait() and signal() are used to provide mutual exclusion, but the unique ability for one process to signal another blocked process using signal() is lost
- In general, there may be times when one process wishes to signal another process based on a condition, much like semaphores.
  - Thus, augment monitors with *condition variables*.

Monitors and Condition Variables

condition x, y;

- A condition variable *x* in a monitor allows three operations
  - x.wait()
    - blocks the calling process
    - can have multiple processes suspended on a condition variable, typically released in FIFO order, but textbook describes another variation specifying a priority p, i.e. call x.wait(p)
  - x.signal()
    - resumes exactly 1 suspended process. If none, then *no effect*.
  - x.queue()
    - Returns true if there is at least one process blocked on x

Monitors and Condition Variables

- Within a monitor, if a process P1 calls x.signal(), then normally that would wake another process P2 blocked on x.wait(). But we must avoid having two processes at the same time in the monitor, so need "wake-up" semantics on a x.signal():
  - Hoare semantics, also called signal-and-wait
    - The signaling process P1 either waits for the woken up process P2 to leave the monitor before resuming, or waits on another CV
  - Mesa semantics, also called signal-and-continue
    - The signaled process P2 waits until the signaling process P1 leaves the monitor or waits on another condition

Dining Philosophers: Monitor-based Solution

- Key insight: pick up 2 forks only if both are free
  - A philosopher can start eating only if both neighbors are not eating
    - Need to define a state for each philosopher
    - Philosopher state: thinking, eating
  - If one of my neighbors is eating, and I'm hungry, ask them to signal() me when they're done
    - Three states of each philosopher: thinking, eating and hungry
    - Need condition variables to signal() waiting hungry philosopher(s)
  - Also, need to Pickup() and Putdown() forks

Dining Philosophers: Monitor-based Solution

```
philosopher (int i)
{
  while (1) {
    //Think
    DiningPhilosophers.pickup(i);
    // pick up forks and eat
    DiningPhilosophers.putdown(i);
  }
}
```

Dining Philosophers: Monitor-based Solution

```
monitor DiningPhilosophers
{
   enum {Thinking, Hungry, Eating} state[5];
   condition self[5]; //to block a philosopher when hungry

   void pickup(int i) {
      //Set state[i] to Hungry
      //If at least one neighbor is eating, block on self[i]
      //Otherwise return
   }

   void putdown(int i) {
      //Change state[i] to Thinking and signal neighbors in
      //case they are waiting to eat
   }
```

... cond. to the next slide

... cond. from the previous slide

```
void test(int i) {
  //Check if both neighbors of i are not eating and i
  //is hungry
  //If so, set state[i] to Eating, and signal philosopher i
}

init( ) {
   for (int i = 0; i < 5; i++)
      state[i] = Thinking;
}
}
```

```
void pickup(int i) {
    //Set state[i] to Hungry
    //If at least one neighbor is eating, block on self[i]
    //Otherwise return
}
```

```
void pickup(int i) {
    state[i] = Hungry;
    test(i);
    if (state[i]!=Eating)
        self[i].wait;
}
```

```
void test(int i) {
  //Check if both neighbors of i are not eating and i is hungry
  //If so, set state[i] to Eating, and signal philosopher i
}
```

```
void putdown(int i) {
    //change state[i] to Thinking and signal neighbors in
    //case they are waiting to eat
  }
```

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

```
void test(int i) {
 //Check if both neighbors of i are not eating and i is hungry
 //If so, set state[i] to Eating, and signal philosopher i
 }
```

```
void test(int i) {
 //Check if both neighbors of i are not eating and i is hungry
 //If so, set state[i] to Eating, and signal philosopher i
 }
```

```
void test(int i) {
      if ((state[(i+1)%5] != Eating) &&
        (state[(i-1)%5] != Eating) &&
        (state[i] == Hungry))
      {
        state[i] = Eating;
        self[i].signal();
      }
   }
```

Dining Philosophers: Monitor-based Solution

- Is deadlock possible in this solution?
  - NO
- Is starvation possible in this solution?
  - YES