Classic Problems in Concurrency

CS 472 Operating Systems

Indiana University – Purdue University Fort Wayne

Classic problems in concurrency

- We investigate two classic concurrency problems from chapters 5 and 6
  - Readers/Writers problem
    - Section 5.6, pp. 245 – 249
  - Dining philosophers problem
    - Section 6.6, pp. 282 – 286

Readers / Writers Problem

a) Any number of readers may read simultaneously

b) Only one writer may write at a time

c) While a writer writes, no reader may read

General mutual exclusion …

- would work but does not take advantage of writers not reading nor of readers not writing
- Fails to permit allowable operations like two readers at once
- unnecessary and much too slow

Semaphore solution

- Try giving readers priority (Figure 5.22, page 246)

Reader protocol to read

```
wait(x);
readcount++;
if(readcount==1)
   wait(wsem);
signal(x);
<read critical section>;
wait(x);
readcount--;
if(readcount==0)
   signal(wsem);
signal(x);
```

```
int readcount = 0
semaphore x {=1}
semaphore wsem {=1}
```

Writer protocol to write

```
wait(wsem);
<write critical section>;
signal(wsem);
```

- Incorrect solution: Writers can starve if there is a continuous sequence of readers

Semaphore solution

```
int readcount=0
int writecount=0
semaphore x {=1}
semaphore y {=1}
semaphore z {=1}
semaphore wsem {=1}
semaphore rsem {=1}
```

- Give writers priority (Figure 5.23, page 247)
- No new readers admitted when any writer intends to write
- *readcount* / *writecount* : used to see if 1 or more readers or writers are active
- *x, y* : semaphores protecting *readcount* and *writecount*
- *wsem* : enforces writing under mutual exclusion
- *rsem* : holds readers while writing occurs
- *z* : only allows one reader to wait on *rsem* at a time to allow a writer to enter after current reader finishes

Reader protocol to read

```
wait( z );
wait( rsem );
wait( x );
readcount++;
if ( readcount == 1 )
    wait( wsem );
signal( x );
signal( rsem );
signal( z );
<reader critical section>.
wait( x );
readcount--;
if ( readcount == 0 )
    signal( wsem );
signal( x );
```

Writer protocol to write

```
wait( y );
writecount++;
if ( writecount == 1 )
    wait( rsem );
signal( y );
wait( wsem );
<writer critical section>;
signal( wsem );
wait( y );
writecount--;
if ( writecount == 0 )
    signal( rsem );
signal( y );
```

Semaphore solution notes

- First reader blocks new writers
- Last reader allows new writer
- First writer blocks new readers
- Last writer allows new readers

Message passing solution

- Give writers priority
- One mailbox for each reader and writer : mbox[j]
- Use a controller process to manage shared data
- Three additional mailboxes
    - readrequest
    - writerequest
    - finished

Message passing solution

- A reader or writer wishing to access data area sends a request message to the appropriate mailbox
- Controller grants request with an “OK” message
- The reader or writer indicates completion with a “finished” message
- Controller services write requests before read requests

Message passing solution

- Variable count enforces mutual exclusion
- Meaning of count
  - Initialize to 100 (> max # of readers)
  - Count > 0 means no writers waiting but there may be readers active
  - Count = 0 means only outstanding request is to write
  - Count < 0 means write request(s) outstanding which are waiting for readers to exit

Reader(i) protocol

```
rmsg = i;
send( readrequest, rmsg );
receive( mbox[i], rmsg );
<reader critical section>;
rmsg = i;
send( finished, rmsg );
```

Writer(j) protocol

```
rmsg = j;
send( writerequest, rmsg );
receive( mbox[j], rmsg );
<writer critical sectioin>;
rmsg = j;
send( finished, rmsg );
```

Controller

```
while (true){
  if ( count > 0 )
    if ( ! empty( finished ) ){
      receive( finished, msg );
      count++;
    }else if ( ! empty( writerequest ) ){
      receive( writerequest, msg );
      writer_id = msg.id
      count = count - 100;
    }else if ( ! empty( readrequest ) ){
      receive( readrequest, msg );
      count--;
      send( msg.id, "OK" );     }
  if ( count == 0 ){
    send( writer_id, "OK" );
    receive( finished, msg );
    count = 100;                  }
  while ( count < 0 ){
    receive( finished, msg );
    count++;                      }
}
```

The dining philosophers problem

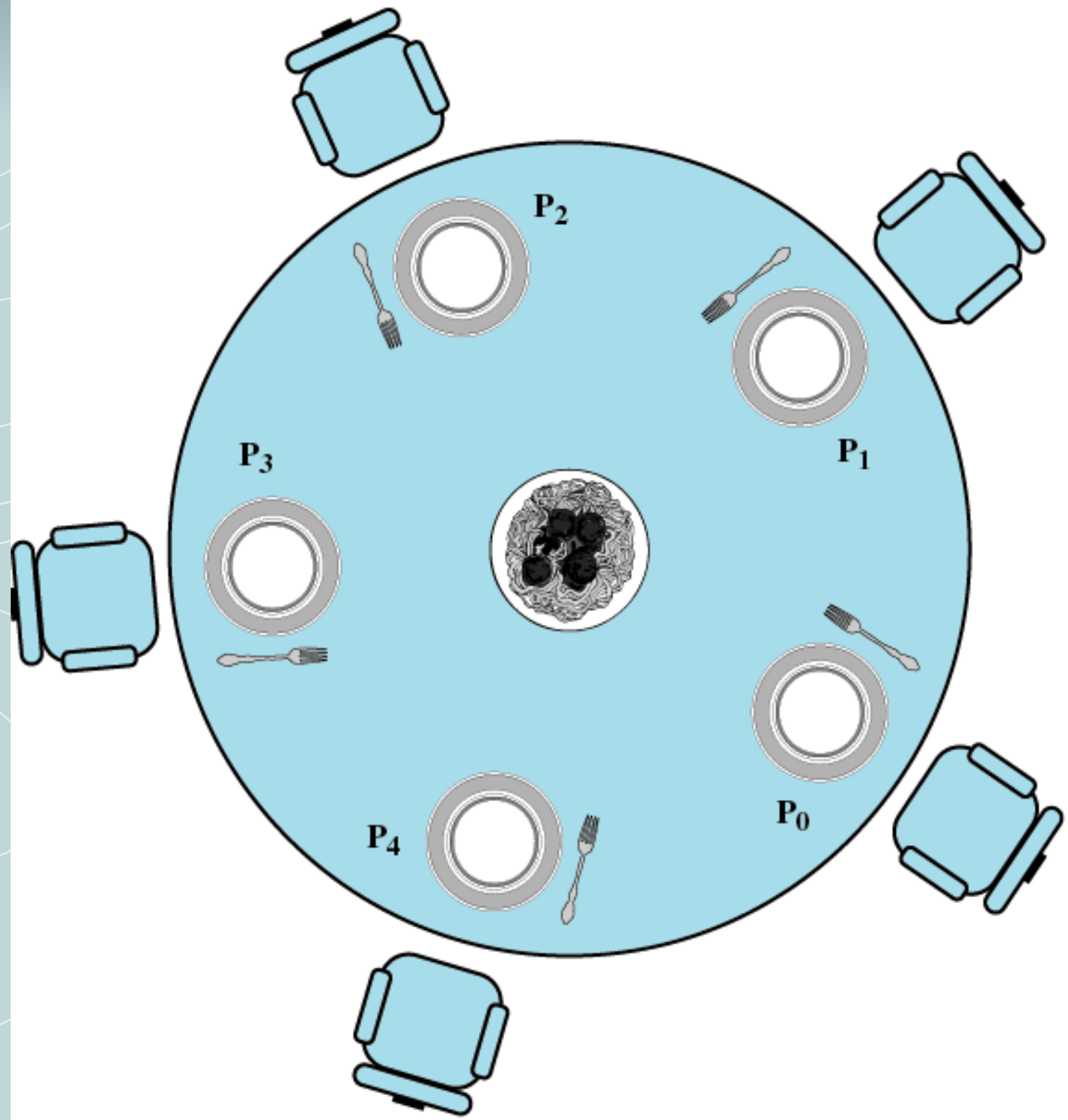

Figure 6.11 Dining Arrangement for Philosophers

The dining philosophers problem

- Each philosopher repeats:

  Think, Eat, Think, Eat, Think, Eat, Think, …
- Each fork is shared among the two neighbors
- To eat, a philosopher needs both adjacent forks
- We want to avoid deadlock and starvation
- Mutual exclusion is needed for each fork to ensure that each fork is used by only one philosopher at a time

The dining philosophers problem

A first solution using semaphores

```
/* program dining philosophers */
semaphore fork[5] = {1};
void philosopher(int i){
   while(true){
      think( );
      wait(fork[i]);              -- take the left fork
      wait(fork[(i+1) mod 5]);    -- then take the right fork
      eat( );
      signal(fork[(i+1) mod 5]);
      signal(fork[i]);
   }
}
void main( ){
   parbegin( philosopher(0), philosopher(1),
            philosopher(2),philosopher(3), philosopher(4) );
}
```

This solution can result in deadlock

The dining philosophers problem

A second solution using semaphores

This solution can result in starvation

```
/* program dining philosophers */
semaphore fork[5] = {1};
void philosopher(int i){
   while(true){
      think( );
      <take both forks at once when available>;
      eat( );
      <put down both forks at once>;
   }
}
void main( ){
   parbegin( philosopher(0), philosopher(1),
            philosopher(2),philosopher(3), philosopher(4) );
}
```

The dining philosophers problem

- This solution can result in starvation

Action	P0	P1	P2	P3	P4
	Number of forks available				
Initially	2	2	2	2	2
P1 takes	1	2	1	2	2
P3 takes	1	2	0	2	1
P2 tries & blocks	1	2	0	2	1
P1 returns	2	2	1	2	1
P1 takes	1	2	0	2	1
P3 returns	1	2	1	2	2
P3 takes	1	2	0	2	1
Etc.					

The dining philosophers problem

Final valid solution

```
/* program dining philosophers */
semaphore fork[5] = {1};
semaphore room = {4};
void philosopher(int i){
   while(true){
      think( );
      wait(room);
      wait(fork[i]);
      wait(fork[(i+1) mod 5]);
      eat( );
      signal(fork[(i+1) mod 5]);
      signal(fork[i]);
      signal(room);
   }
}
void main( ){
   parbegin( philosopher(0), philosopher(1), philosopher(2),
             philosopher(3), philosopher(4) );
}
```

Allow only four philosophers in the room at a time

The dining philosophers problem

- Another solution
  - This one uses a monitor
  - There is an array of five condition variables
    - One condition variable for each fork
  - There is a second **boolean** array that records the availability of each fork
  - The structure of this solution is similar to the failed first solution using semaphores
    - However, this solution does not suffer from deadlock because only one process at a time may be in the monitor

```
monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};       /* availability status of each fork */

void get_forks(int pid)         /* pid is the philosopher id number */
{
   int left = pid;
   int right = (pid++) % 5;
   /*grant the left fork*/
   if (!fork(left)
      cwait(ForkReady[left]);          /* queue on condition variable */
   fork(left) = false;
   /*grant the right fork*/
   if (!fork(right)
      cwait(ForkReady(right);          /* queue on condition variable */
   fork(right) = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (pid++) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])      /*no one is waiting for this fork */
      fork(left) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])     /*no one is waiting for this fork */
      fork(right) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```

```
void philosopher[k=0 to 4]          /* the five philosopher clients */
{
   while (true)
   {
      <think>;
      get_forks(k);          /* client requests two forks via monitor */
      <eat spaghetti>;
      release_forks(k);      /* client releases forks via the monitor */
   }
}
```

Figure 6.14 A Solution to the Dining Philosophers Problem Using a Monitor

Note: The monitor method *empty(c)* may be applied to a condition variable *c* to determine if the queue of processes waiting on *c* is empty or not