Chapter 6
Process Synchronization

Zhi Wang
Florida State University

Content

- Critical section
- Peterson’s solution
- Synchronization hardware
- Semaphores
- Classic synchronization problems
- Monitors
- Synchronization examples
- Atomic transactions

Background

- Concurrent access to shared data may result in **data inconsistency**
  - data consistency requires orderly execution of cooperating processes
- Example:
  - consider a solution to the consumer-producer problem that fills all buffers
  - use an integer count to keep track of the number of full buffers
    - initially, count is set to 0
    - incremented by the producer after it produces a new buffer
    - decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
      /*produce an item and put in nextProduced  */
      while (counter == BUFFER_SIZE); // do nothing
      buffer [in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
}
```

Consumer

```
while (true)  {
      while (counter == 0); // do nothing
      nextConsumed =  buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      counter--;
      /*consume the item in nextConsumed*/
}
```

Race Condition

- counter++/-- could be implemented as
    - register1 = counter
    - register1 = register1 +/- 1
    - counter = register1
- Consider this execution interleaving with “count = 5” initially:
    - S0: producer: register1 = counter {register1 = 5}
    - S1: producer: register1 = register1+1 {register1 = 6}
    - S2: consumer: register2 = counter {register2 = 5}
    - S3: consumer: register2 = register2-1 {register2 = 4}
    - S4: producer: counter = register1 {count = 6 }
    - S5: consumer: counter = register2 {count = 4}

Critical Section

- Consider system of n processes {p_0, p_1, … p_{n-1}}
- Each process has a critical section segment of code
  - e.g., to change common variables, update table, write file, etc.
- Only one process can be in the critical section
  - when one process in critical section, no other may be in its critical section
  - each process must ask permission to enter critical section in entry section
  - the permission should be released in exit section

Critical Section

- General structure of process p_i is

```
do {
   entry section
      critical section
   exit section
      remainder section
} while (true)
```

Solution to Critical-Section

- **Mutual Exclusion**
  - only one process can execute in the critical section
- **Progress**
  - if no process is executing in its critical section
  - there exist some processes that wish to enter their critical section
  - these processes cannot be postponed indefinitely
  - only these processes participate in the decision of who to enter CS
- **Bounded waiting**
  - a process should not be able to keep entering its critical section if there are other processes waiting to enter the critical section
  - it prevents **starvation**
  - no assumption concerning ***relative speed*** of the n processes

Peterson’s Solution

- Peterson’s solution solves **two-processes** synchronization
- It assumes that LOAD and STORE are **atomic**
    - **atomic**: execution cannot be interrupted
- The two processes share two variables
    - int **turn**: whose turn it is to enter the critical section
    - Boolean **flag[2]**: whether a process is ready to enter the critical section

Peterson’s Solution

- P_0:

```
do {
      flag[0] = TRUE;
      turn = 1;
      while (flag[1] && turn == 1);
      critical section
      flag[0] = FALSE;
      remainder section
} while (TRUE);
```

- P_1:

```
do {
      flag[1] = TRUE;
      turn = 0;
      while (flag[0] && (turn == 0));
      critical section
      flag[1] = FALSE;
      remainder section
} while (TRUE);
```

- mutual exclusion?
- progress?
- bounded-waiting?

Synchronization Hardware

- Many systems provide hardware support for critical section code
- **Uniprocessors: disable interrupts**
  - currently running code would execute without preemption
  - generally too inefficient on multiprocessor systems
    - need to disable all the interrupts
    - operating systems using this not scalable
- Modern machines provide special **atomic** hardware instructions
  - **test-and-set**: either test memory word and set value
  - **swap**: swap contents of two memory words
  - these instructions can be used to implement locks
    - usually called **spin lock**
    - ok for very short critical sections

Critical-section Using Locks

```
do {
     acquire lock
        critical section
     release lock
     remainder section
} while (TRUE);
```

Test-and-Set Instruction

- Defined as below, but **atomically**

```
bool test_set (bool *target)
{
      bool rv = *target;
      *target = TRUE;
      return rv:
}
```

Lock with Test-and-Set

- shared variable: bool **lock** = FALSE

```
do {
    while (test_set(&lock));   // busy wait
    critical section
    lock = FALSE;
    remainder section
} while (TRUE);
```

- Mutual exclusion?
- progress?
- bounded-waiting?

Swap Instruction

- Defined as below, but **atomically**

```
void swap (bool *a, bool *b)
{
    bool temp = *a;
    *a = *b;
    *b = temp:
}
```

Lock with Swap

- shared variable: bool **lock** = FALSE
- each process has a local variable: **key**

```
do {
    key = TRUE;
    while ( key == TRUE) swap (&lock, &key);
    critical section
    lock = FALSE;
    remainder section
} while (TRUE);
```

- Mutual exclusion? Progress? Bounded-waiting?

Bounded Waiting for Test-and-Set Lock

```
do {
      waiting[i] = TRUE;
      key = TRUE;
      while (waiting[i] && key) key=test_set(&lock);
      waiting[i] = FALSE;
      critical section
      j = (i + 1) % n;
      while ((j != i) && !waiting[j]) j = (j + 1) % n;
      if (j == i)
         lock = FALSE;
      else
         waiting[j] = FALSE;
      …
} while (TRUE);
```

Semaphore

- **Semaphore** S is an integer variable
    - e.g., to represent *how many units of a particular resource is available*
- It can only be updated with two atomic operations: **wait** and **signal**
    - **spin lock** can be used to guarantee atomicity of wait and signal
    - originally called P and V (Dutch)
    - a simple implementation with busy wait can be:

```
wait(s)
{
    while (s <= 0) ; //busy wait
    s--;
}
```

```
signal(s)
{
    s++;
}
```

Semaphore

- **Counting semaphore**: allowing arbitrary resource count
- **Binary semaphore**: integer value can be only 0 or 1
    - also known as **mutex lock** to provide mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
      wait (mutex);
      critical section
      signal (mutex);
      remainder section
} while (TRUE);
```

Semaphore w/ Waiting Queue

- Associate a waiting queue with each semaphore
    - place the process on the waiting queue if **wait** cannot return immediately
    - wake up a process in the waiting queue in **signal**
- There is no need to busy wait
- Note: wait and signal must still be atomic

Semaphore w/ Waiting Queue

```
wait(semaphore *S)
{
    S->value--;
     if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S)
{
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
     }
}
```

Deadlock and Starvation

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

```
let S and Q be two semaphores initialized to 1

      P0                          P1

   wait (S);                   wait (Q);

   wait (Q);                   wait (S);

   …                            …

   signal (S);                 signal (Q);

   signal (Q);                 signal (S);
```

- **Starvation**: indefinite blocking
  - a process may never be removed from the semaphore's waiting queue
  - does starvation indicate deadlock?

Priority Inversion

- **Priority Inversion**: a higher priority process is **indirectly** preempted by a lower priority task
  - e.g., three processes, **P_L**, **P_M**, and **P_H** with priority $P_L < P_M < P_H$
  - **P_L** holds a lock that was requested by **P_H** ⇨ **P_H** is blocked
  - **P_M** becomes ready and preempted the **P_L**
  - It effectively "inverts" the relative priorities of **P_M** and **P_H**
- Solution: **priority inheritance**
  - temporary assign the highest priority of waiting process (P_H) to the process holding the lock (P_L)

Classical Synchronization Problems

- Bounded-buffer problem
- Readers-writers problem
- Dining-philosophers problem

Bounded-Buffer Problem

- Two processes, the producer and the consumer share **n** buffers
  - the producer generates data, puts it into the buffer
  - the consumer consumes data by removing it from the buffer
- The problem is to make sure:
  - **the producer won’t try to add data into the buffer if its full**
  - **the consumer won’t try to remove data from an empty buffer**
  - also call producer-consumer problem
- Solution:
  - n buffers, each can hold one item
  - semaphore **mutex** initialized to the value **1**
  - semaphore **full** initialized to the value **0**
  - semaphore **empty** initialized to the value **N**

Bounded-Buffer Problem

- The producer process:

```
do {
   //produce an item

   …

   wait(empty);

   wait(mutex);

   //add the item to the  buffer

   …

   signal(mutex);

   signal(full);

} while (TRUE)
```

Bounded Buffer Problem

- The consumer process:

```
do {
    wait(full);
    wait(mutex);
    //remove an item from  buffer
    …
    signal(mutex);
    signal(empty);
    //consume the item
    …
} while (TRUE);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - readers: only read the data set; they do not perform any updates
  - writers: can both read and write
- The readers-writers problem:
  - allow multiple readers to read at the same time (**shared access**)
  - only one single writer can access the shared data (**exclusive access**)
- Solution:
  - semaphore **mutex** initialized to 1
  - semaphore **wrt** initialized to 1
  - integer **read_count** initialized to 0

Readers-Writers Problem

- The writer process

```
do {
   wait(wrt);
   //write the shared data
   …
   signal(wrt);
} while (TRUE);
```

Readers-Writers Problem

- The structure of a reader process

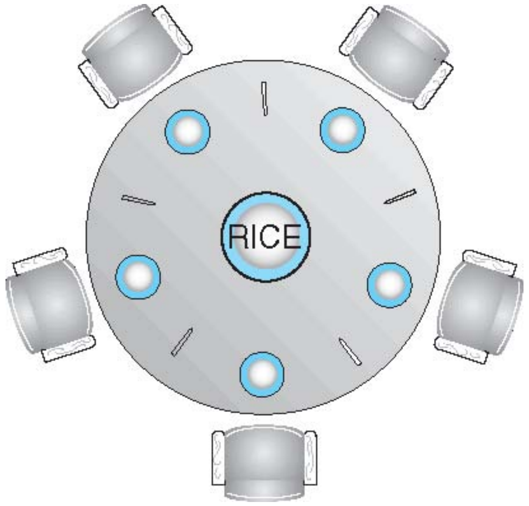
```
do {
   wait(mutex);
   readcount++ ;
   if (readcount == 1)
       wait(wrt) ;
   signal(mutex)

   //reading data
   …
   wait(mutex) ;
   readcount--;
   if (readcount == 0)
       signal(wrt) ;
   signal(mutex) ;
} while(TRUE);
```

Readers-Writers Problem Variations

- Two variations of readers-writers problem (different **priority** policy)
    - no reader kept waiting unless writer is updating data
    - once writer is ready, it performs write ASAP
- Which variation is implemented by the previous code example???
- Both variation may have starvation leading to even more variations
    - how to prevent starvation

Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
  - they sit in a round table, but don’t interact with each other
- They occasionally try to pick up 2 chopsticks (one at a time) to eat
  - one chopstick between each adjacent two philosophers
  - need both chopsticks to eat, then release both when done
  - Dining-philosopher problem represents **multi-resource synchronization**
- Solution (assuming **5 philosophers**):
  - semaphore **chopstick[5]** initialized to 1



Dining-Philosophers Problem

- Philosopher i (out of 5):

```
do  {
    wait(chopstick[i]);
    wait(chopStick[(i+1)%5]);
    eat
    signal(chopstick[i]);
    signal(chopstick[(i+1)%5]);
    think
} while (TRUE);
```

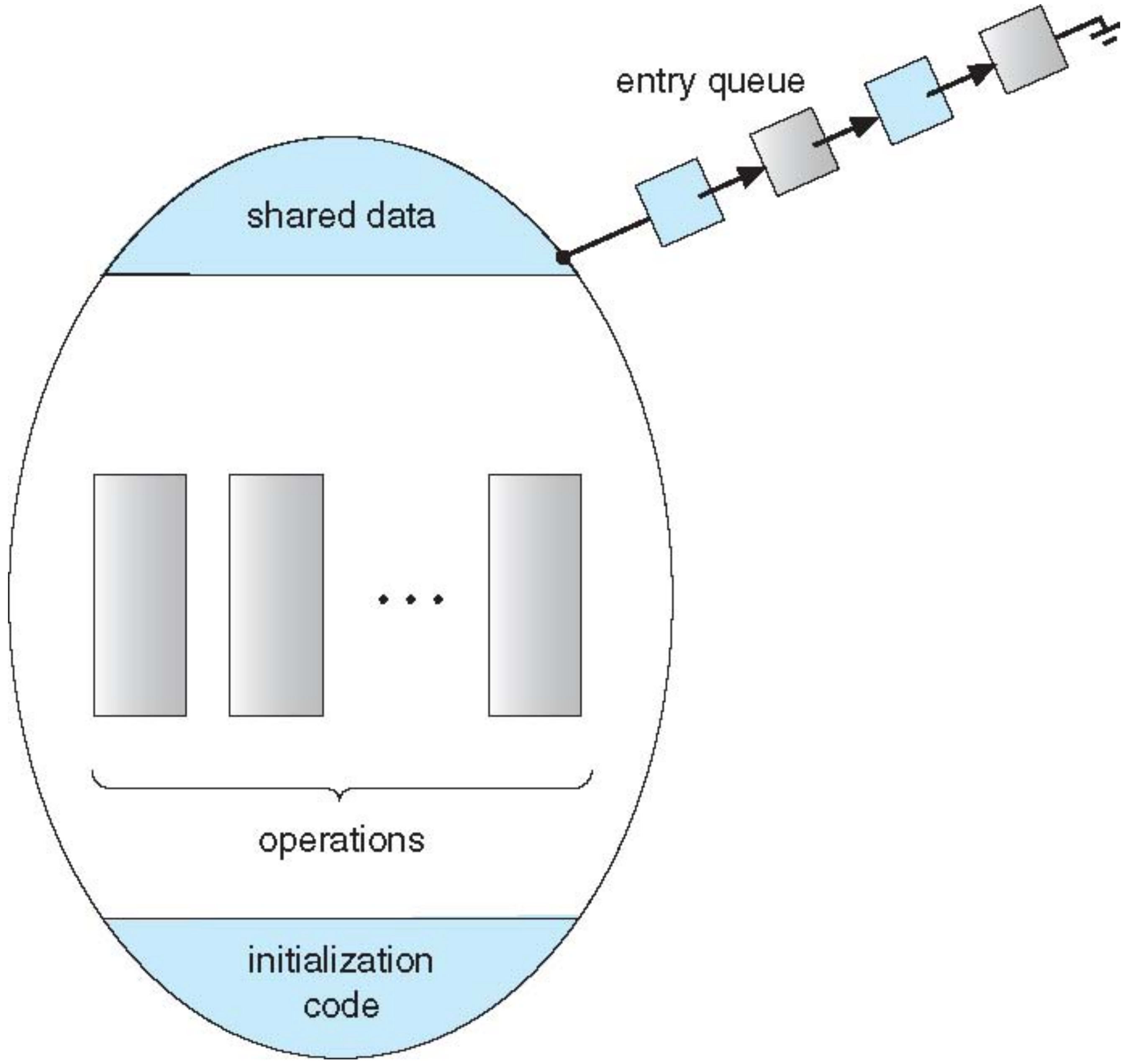
- What is the problem with this algorithm?
  - **deadlock** and **starvation**

Monitors

- Monitor is a high-level abstraction to provide synchronization
- Monitor is an abstract data type
    - similar to classes in object-oriented programming
    - internal variables only accessible by code within the procedure
- **Only one thread may be active within the monitor at a time**!!!

```
monitor monitor–name
{
      // shared variable declarations
      procedure P1 (…) { …. }
      …
      procedure Pn (…) {……}
      Initialization code (…) { … }
}
```

Schematic View of a Monitor



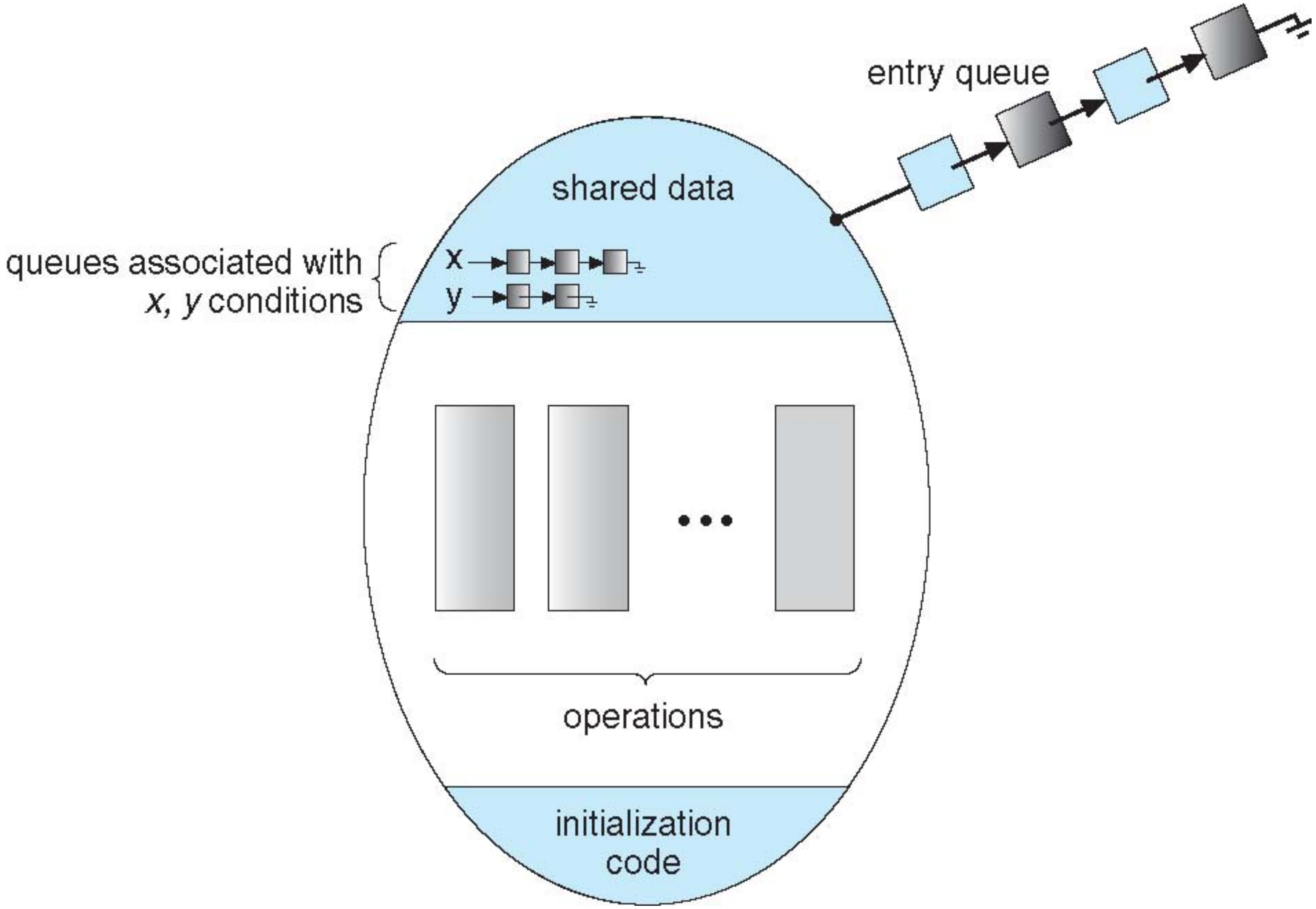
Problems with Monitor

- Monitor can provide mutual exclusion
  - only one thread (process) can be active within a monitor
- Threads may need to wait until some condition *P* holds true
- Busy waiting in monitor does not work
  - only one thread can be active within a monitor ⇨
  - if it busy-waits, others cannot enter monitor ⇨
  - condition *P* may rely on other thread’s operations
- Solution: *condition variable*

Condition Variable

- **Condition variable** is a waiting queue in monitor, on which a thread may wait for some condition to become true
  - each condition variable is associated with an assertion $\mathbf{P_c}$
  - thread waiting on a CV is not considered to be occupying the monitor
  - other thread may enter monitor and signal CV when $\mathbf{P_c}$ becomes valid
- Two operations on a condition variable:
  - **wait**: suspend the calling thread until signal
  - **signal**: resumes one thread (if any) waiting on the CV
    - if no thread on the variable, signal has no effect on the variable

Monitor with Condition Variables



Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
    enum { THINKING; HUNGRY, EATING} state[5] ;
    condition self[5];

    void pickup (int i) {
        state[i]=HUNGRY;
        test(i);
        if (state[i]!=EATING)
        self[i].wait;
    }

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

Solution to Dining Philosophers

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

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

Solution to Dining Philosophers

- Each philosopher *i* invokes the operations in the following sequence:

```
DiningPhilosophers.pickup (i)

EAT

DiningPhilosophers.putdown (i);
```

- Only one philosopher can be active in the monitor
  - it will start eating when neither neighbor is eating, otherwise it will wait
- No deadlock, but starvation is possible

Monitor Implementation

- Variables

```
semaphore mutex;   // (initially  = 1)
semaphore next;      // (initially  = 0)
int next_count = 0;
```

- Each procedure F will be replaced by

```
wait(mutex);
body of F;
if (next_count > 0)
    signal(next)
else
    signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Pthread CV Example

```
int     count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t)
{
    int i;
    long my_id = (long)t;

    for (i=0; i < TCOUNT; i++) {
        pthread_mutex_lock(&count_mutex);
        count++;

        /* Check the value of count and signal waiting thread when condition is
        reached.  Note that this occurs while mutex is locked. */
        if (count == COUNT_LIMIT) {
            pthread_cond_signal(&count_threshold_cv);
        }

        pthread_mutex_unlock(&count_mutex);
        /* Do some work so threads can alternate on mutex lock */
        sleep(1);
    }
    pthread_exit(NULL);
}
```

Pthread CV Example

```c
void *watch_count(void *t)
{
   long my_id = (long)t;

   printf("Starting watch_count(): thread %ld\n", my_id);

   /*
   Lock mutex and wait for signal.  Note that the pthread_cond_wait routine
   will automatically and atomically unlock mutex while it waits.
   Also, note that if COUNT_LIMIT is reached before this routine is run by
   the waiting thread, the loop will be skipped to prevent pthread_cond_wait
   from never returning.
   */
   pthread_mutex_lock(&count_mutex);
   while (count < COUNT_LIMIT) {
      pthread_cond_wait(&count_threshold_cv, &count_mutex);
      count += 125;
   }
   pthread_mutex_unlock(&count_mutex);
   pthread_exit(NULL);
}
```

Synchronization Examples

- Windows XP
- Linux

Windows XP Synchronization

- **interrupt mask**: protect access to global data on uniprocessor systems
- **spinlocks** on multiprocessor systems
  - spinlocking-thread will never be preempted
- **dispatcher objects** for user-land
  - to provide mutex, semaphore, event, and timer
  - either in the signaled state (object available) or non-signaled state (will block)

Linux Synchronization

- Linux:
  - prior to version 2.6, disables interrupts to implement short critical sections
  - version 2.6 and later, fully preemptive
- Linux provides:
  - **semaphores**
    - on single-cpu system, spinlocks replaced by enabling/disabling kernel preemption
  - **spinlocks**
  - **reader-writer locks**

End of Chapter 6