CS3600 — Systems and Networks

SPRING 2013

Lecture 7: Synchronization

Prof. Alan Mislove (amislove@ccs.neu.edu)

Background

Processes often need to coordinate and share information

But, concurrent access to shared data may result in data inconsistency

Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

This lecture: how do we ensure correct execution when multiple processes may be accessing the same data?

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information (repeatedly) that is consumed by a *consumer* process

- Processes allowed to *share memory*

- How can we implement a producer and consumer using shared memory?
  - Assume two shared variables: buffer[] and counter

Producer

```
in = 0;

while (true) {

  /*  produce an item */

  while (counter == BUFFER_SIZE) {} // do nothing
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  counter++;
}
```

Consumer

```
out = 0;

while (true)  {
  while (counter == 0) {} // do nothing
  nextConsumed =  buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  counter--;

  /*  consume the item */

}
```

We have a shared integer **counter** that keeps track of the number of full buffer entries. Initially, counter is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Does this solution work?

Race Condition

`counter++` could be implemented as

```
load %eax counter
add %eax %eax 1          (%eax++)
store counter %eax
```

`counter--` could be implemented as

```
load %eax counter
add %eax %eax -1         (%eax--)
store counter %eax
```

Consider this execution interleaving with "counter = 5" initially:

S0: producer execute `load %eax counter` {producer's %eax = 5}
S1: producer execute `add %eax %eax 1` {producer's %eax = 6}
S2: consumer execute `load %eax counter` {consumer's %eax = 5}
S3: consumer execute `add %eax %eax -1` {consumer's %eax = 4}
S4: producer execute `store %eax counter` {counter = 6 }
S5: consumer execute `store %eax counter` {counter = 4}

Generalization: Critical Section Problem

Consider system of n processes $\{p_0, p_1, \ldots p_{n-1}\}$

Each process has **critical section** segment of code

Process may be changing common variables, updating table, writing file, etc

When one process in critical section, no other may be in its critical section

Critical section problem is to design protocol to solve this

Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Especially challenging with preemptive kernels

Critical Section

General structure of process p_i is

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

Figure 6.1 General structure of a typical process P_i.

Reqs. for solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

- Assume that each process executes at a nonzero speed
- No assumption concerning **relative speed** of the n processes

Peterson's Solution

Two process solution

Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted

The two processes share two variables:

```
int turn;
Boolean flag[2];
```

The variable **turn** indicates whose turn it is to enter the critical section

The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i]** = true implies that process P_i is ready!

Algorithm for Process P_i

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j) {}

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Synchronization Hardware

Many systems provide hardware support for critical section code

Uniprocessors – could disable interrupts

- Currently running code would execute without preemption
- Generally too inefficient on multiprocessor systems
  - Operating systems using this not broadly scalable

Modern machines provide special atomic hardware instructions

- Atomic = non-interruptable
- Either test memory word and set value
- Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock

        critical section

    release lock

        remainder section

} while (TRUE);
```

TestAndSet Instruction

```
boolean TestAndSet (boolean *target) {

   boolean rv = *target;
   *target = TRUE;
   return rv:

}
```

Solution using TestAndSet

Shared boolean variable lock, initialized to FALSE

Solution:

```
lock = false;

do {
   // busy wait while lock is true
   while ( TestAndSet (&lock )) {}

//       critical section

   lock = FALSE;

//       remainder section

} while (TRUE);
```

Swap Instruction

```
void Swap (boolean *a, boolean *b) {

  boolean temp = *a;
  *a = *b;
  *b = temp:

}
```

Solution using Swap

Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key

Solution:

```
lock = FALSE;
do {
   key = TRUE;

   // try to grab the lock
   while (key == TRUE)
      Swap (&lock, &key );

//         critical section

      lock = FALSE;

//         remainder section

} while (TRUE);
```

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
  waiting[i] = TRUE;
  key = TRUE;

  while (waiting[i] && key)
    key = TestAndSet(&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;

//       remainder section

} while (TRUE);
```

Semaphore

Synchronization tool that does not (necessarily) require busy waiting

Semaphore *S* – integer variable

Two standard operations modify S: wait() and signal()

Originally called P() and V()

Less complicated

Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
  while S <= 0
    ; // no-op
  S--;
}


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

Semaphore as General Synchronization Tool

Counting semaphore – integer value can range over an unrestricted domain

Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement

Also known as **mutex locks**

Can implement a counting semaphore S as a binary semaphore

Provides mutual exclusion

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

Semaphore Implementation

Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time

Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section

- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied

Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation without busy waiting

With each semaphore there is an associated waiting queue

Each entry in a waiting queue has two data items:

- value (of type integer)
- pointer to next record in the list

Two operations:

- **block** – place the process invoking the operation on the appropriate waiting queue
- **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Semaphore Implementation without busy waiting

Implementation of wait:

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

Implementation of signal:

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

Bounded-Buffer Problem

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 (Cont.)

The structure of 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 (Cont.)

The structure of the consumer process

```
do {

  wait (full);
  wait (mutex);

  //  remove an item from  buffer

  signal (mutex);
  signal (empty);

  //  consume the item

} while (TRUE);
```

Monitors

A high-level abstraction that provides a convenient and effective mechanism for process synchronization

Abstract data type, internal variables only accessible by code within the procedure

Only one process may be active within the monitor at a time

But not powerful enough to model some synchronization schemes

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

 procedure Pn (…) {……}

 Initialization code (…) { … }

}
```

Problems with synchronization

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

P_0	P_1
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 queue in which it is suspended

Priority Inversion – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Solved via **priority-inheritance protocol**