Distributed Programming

Problems & Solutions

Agenda

- Concurrency Control
  - Software Algorithms
  - Semaphores (OS Support)
  - Monitors (Compiler Support)
  - Distributed Mututal Exclusion Algorithms

- Distributed Coordination
  - Synchronized Clocks
  - Logical Clocks
  - Election Algorithms

Strict Alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

First me, then you

Problems with strict alternation

- Employs busy waiting-while waiting for the cr, a process spins
- If one process is outside the cr and it is its turn, then other process has to wait until outside guy finishes both outside AND inside (cr) work

Peterson's Solution (1981)

```
#define FALSE  0
#define TRUE   1
#define N      2                               /* number of processes */

int turn;                                      /* whose turn is it? */
int interested[N];                             /* all values initially 0 (FALSE) */

void enter_region(int process);                /* process is 0 or 1 */
{
     int other;                                /* number of the other process */

     other = 1 – process;                      /* the opposite of process */
     interested[process] = TRUE;               /* show that you are interested */
     turn = process;                           /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)                 /* process: who is leaving */
{
     interested[process] = FALSE;              /* indicate departure from critical region */
}
```

-

Peterson

- Process 0 & 1 try to get in simultaneously
- Last one in sets turn: say it is process 1
- Process 0 enters (turn= = process is False)

Semaphores

- Semaphore is an integer variable
- Used to sleeping processes/wakeups
- Two operations, down/P/Wait and up/V/Signal
- Down checks semaphore. If not zero, decrements semaphore. If zero, process goes to sleep
- Up increments semaphore. If more then one process asleep, one is chosen randomly and enters critical region (first does a down)
- ATOMIC IMPLEMENTATION-interrupts disabled

Producer Consumer with Semaphores

- 3 semaphores: full, empty and mutex
- Full counts full slots (initially 0)
- Empty counts empty slots (initially N)
- Mutex protects variable which contains the items produced and consumed

Producer Consumer with semaphores

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
        int item;

        while (TRUE) {                  /* TRUE is the constant 1 */
                item = produce_item( ); /* generate something to put in buffer */
                down(&empty);           /* decrement empty count */
                down(&mutex);           /* enter critical region */
                insert_item(item);      /* put new item in buffer */
                up(&mutex);             /* leave critical region */
                up(&full);              /* increment count of full slots */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                  /* infinite loop */
                down(&full);            /* decrement full count */
                down(&mutex);           /* enter critical region */
                item = remove_item( );  /* take item from buffer */
                up(&mutex);             /* leave critical region */
                up(&empty);             /* increment count of empty slots */
                consume_item(item);     /* do something with the item */
        }
}
```

• • •

Monitors

- Easy to make a mess of things using mutexes and condition variables. Little errors cause disasters.
  - Producer consumer with semaphores- **interchange two downs in producer code causes deadlock**
- Monitor is a language construct which enforces mutual exclusion and blocking mechanism
- C does not have monitor

Monitors

- Monitor consists of {procedures, data structures, and variables} grouped together in a “module”
- A process can call procedures inside the monitor, but cannot directly access the stuff inside the monitor

Monitor-a picture

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .        .        .
        end;
end monitor;
```

Onwards

- In a monitor it is the job of the compiler, not the programmer to enforce mutual exclusion.
- Only one process at a time can be in the monitor

  When a process calls a monitor, the first thing done is to check if another process is in the monitor. If so, calling process is suspended.
- Need to enforce blocking as well –
  - use condition variables
  - Use wait , signal ops on cv's

Condition Variables

- Monitor discovers that it can’t continue (e.g. buffer is full), issues a signal on a condition variable (e.g. full) causing process (e.g. producer) to block
- Another process is allowed to enter the monitor (e.g. consumer).This process can can issue a signal, causing blocked process (producer) to wake up
- Process issuing signal leaves monitor

Producer Consumer Monitor

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count – 1;
                if count = N – 1 then signal(full)
        end;

        count := 0;
end monitor;

procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Monitors:Good vs Bad

- The good-No messy direct programmer control of semaphores
- The bad- You need a language which supports monitors (Java - synchronized).
- OS’s are written in C

Semaphores:Good vs Bad

- The good-Easy to implement
- The bad- Easy to mess up

Reality

- Monitors and semaphores only work for shared memory
- Don’t work for multiple CPU’s which have their own private memory, e.g. workstations on an Ethernet

Message Passing

- Used when memory is not shared
- Information exchange between machines
- Two primitives
  - Send(destination, &message)
  - Receive(source,&message)
- Lots of design issues
  - Message loss
    - acknowledgements, time outs deal with loss
  - Authentication-how does a process know the identity of the sender? For sure, that is

Producer Consumer Using Message Passing

- Consumer sends N empty messages to producer
- Producer fills message with data and sends to consumer

Producer-Consumer Problem with Message Passing (1)

```
#define N 100                          /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                         /* message buffer */

    while (TRUE) {
        item = produce_item( );        /* generate something to put in buffer */
        receive(consumer, &m);         /* wait for an empty to arrive */
        build_message(&m, item);       /* construct a message to send */
        send(consumer, &m);            /* send item to consumer */
    }
}
```

.

Producer-Consumer Problem with Message Passing (2)

. . .

```
void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
    while (TRUE) {
        receive(producer, &m);              /* get message containing item */
        item = extract_item(&m);            /* extract item from message */
        send(producer, &m);                 /* send back empty reply */
        consume_item(item);                 /* do something with the item */
    }
}
```

Message Passing Approaches

- Have unique ID for address of recipient process
- Mailbox
  - In producer consumer, have one for the producer and one for the consumer
- No buffering-sending process blocks until the receive happens. Receiver blocks until send occurs (Rendezvous)

Next

Distributed Algorithms