OPERATING SYSTEMS

by
Marwa Yusuf

Lecture 8
Thursday 19-11-2020

Chapter 2 (2.3.6 to 2.3.10)

Processes and Threads

Mutex

- Simpler version of semaphore (no counting, just 1 or 0).
- Easy and more efficient to implement.
- 2 states: **unlocked** (0) or **locked** (any other state).
- 2 operations:
  - ***mutex_lock*** (successful if initially unlocked, blocked otherwaise).
  - ***mutex_unlock*** awakes one sleeping process at random.
- As they are simple, easily implemented in user space given TSL or XCHG.

Mutex (cont.)

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Mutex (cont.)

- **mutex_trylock()**
- With threads sharing is no problem, but what about processes?
  - Share memory in kernel.
  - Some OSs allow sharing memory area in user space.
  - Share a file.

Futex

- Skipped.

Mutex in Pthreads

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Condition Variables

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Condition Variables (cont.)

```
#include <stdio.h>
#include <pthread.h>

#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                    /* used for signaling */
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);     /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                         /* put item in buffer */
                pthread_cond_signal(&condc);        /* wake up consumer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}

void *consumer(void *ptr)                       /* consume data */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);     /* get exclusive access to buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                         /* take item out of buffer */
                pthread_cond_signal(&condp);        /* wake up producer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}
```

Condition Variables (cont.)

```
int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

Monitors

- Semaphores are error-prone. Try changing the order.
- **Monitor**: a collection of procedures, variables and data structures grouped together (like class), procedures can be accessed from outside (e.g. *public*) while variables are not.
- Only one process active on a monitor procedure in a time.
- Handled by the compiler.
- It's a programming language feature (Not supported in C)
- Condition variables help in blocking whenever needed, allowing another process to proceed.
- When another process is awakened, what happens to the original? Suspended, ended (signal must be the last instruction) or continued.
- A signal may be lost if no one is waiting (use variables to track).

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

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

Monitors (cont.)

```
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 (cont.)

```
public class ProducerConsumer {
        static final int N = 100;       // constant giving the buffer size
        static producer p = new producer( );     // instantiate a new producer thread
        static consumer c = new consumer( ); // instantiate a new consumer thread
        static our_monitor mon = new our_monitor( );       // instantiate a new monitor

        public static void main(String args[ ]) {
            p.start( );       // start the producer thread
            c.start( );       // start the consumer thread
        }

        static class producer extends Thread {
            public void run( )  { // run method contains the thread code
                int item;
                while (true) {      // producer loop
                    item = produce_item( );
                    mon.insert(item);
                }
            }
            private int produce_item( ) { ... }       // actually produce
        }

        static class consumer extends Thread {
            public void run( )  { run method contains the thread code
                int item;
                while (true) {      // consumer loop
                    item = mon.remove( );
                    consume_item (item);
                }
            }
            private void consume_item(int item) { ... }// actually consume
        }
```

```
static class our_monitor {   // this is a monitor
    private int buffer[ ] = new int[N];
    private int count = 0, lo = 0, hi = 0;   // counters and indices

    public synchronized void insert(int val) {
        if (count == N) go_to_sleep( );     // if the buffer is full, go to sleep
        buffer [hi] = val;  // insert an item into the buffer
        hi = (hi + 1) % N;        // slot to place next item in
        count = count + 1;       // one more item in the buffer now
        if (count == 1) notify( );        // if consumer was sleeping, wake it up
    }

    public synchronized int remove( ) {
        int val;
        if (count == 0) go_to_sleep( );      // if the buffer is empty, go to sleep
        val = buffer [lo];  // fetch an item from the buffer
        lo = (lo + 1) % N;        // slot to fetch next item from
        count = count – 1;       // one few items in the buffer
        if (count == N – 1) notify( );  // if producer was sleeping, wake it up
        return val;
    }
    private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
}
```

Monitors(cont.)

- Problems:
  - Not all languages support them (like c and Pascal).
  - Semaphores also are not supported directly, but can be implemented very easily without the need for compiler cooperation.
  - Work with multiple CPUs sharing the same memory, but not distributed systems with multiple memories.
    - What can we do in this case?

Message Passing

- Interprocess Communication approach.
  - send(destination, &message);
  - receive(source, &message); //can receive from **ANY**
- System calls not language constructs, so can be put in library procedures.
- Receiver can either **block** waiting for a message or return with error code.

Message Passing Design Issues

- Issues not faced with semaphores or monitors, specially in case of communication through networks.
- Message lost:
  - Receiver send **acknowledgment**.
- **Acknowledgment** lost, duplicate message:
  - Add **consecutive sequence numbers** in messages.
- These are typical issues of networking.
- Naming processes to avoid ambiguity.
- Authentication: how to know the other process is not an imposter.
- If processes are on the same machine, semaphores and monitors are faster.

Message Passing (Producer-Consumer)

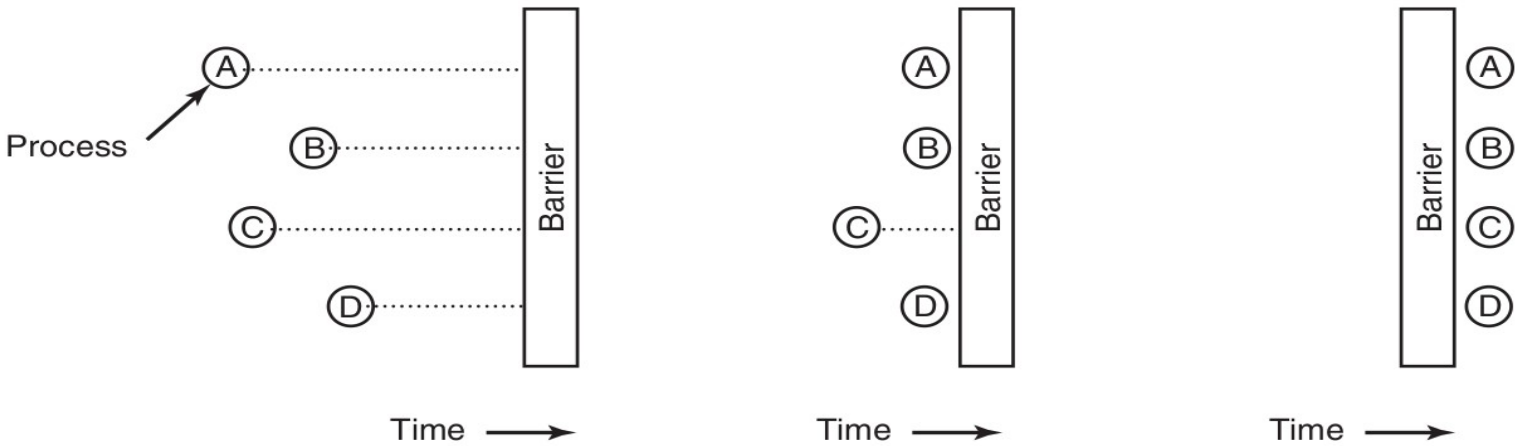
```
void producer(void)
{
        int item;
        message m;                                  /* message buffer */

        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 */
        }
}


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 (Producer-Consumer)

- Assume OS save (buffer) messages till consumed.
- To design message passing:
  - Unique addresses for process where they can receive messages directly.
  - **Mailbox** data structure assigned for the process (ex: producer-consumer).
  - No buffer at all: sender blocks until receive happens and vice versa.
    - **Rendezvous**: easier to implement, but less flexible.
- Usually usde for parallel programming.
  - Ex: MPI (Message-Passing Interface)

Barriers

- Intended for groups of processes rather than 2.
- Synchronize progress, all process wait at the barrier to advance together.
- Ex: Updating parts of a very large matrix in a loop.



Avoiding Locks

- Section 2.3.10 skipped.