Operating Systems

0107451

Chapter 2 Processes and Threads

Dr. Naeem Odat

Tafial Technical University
Department of Computer and Communications Engineering

2.3 Inter Process Communications (IPC)

IPC issues

Processes frequently need to pass information between each other. Some issues are there:

- How IPC is implemented (simple for threads but more involving for processes).
- Race conditions
- Synchronizations

Inter Process Communications (IPC)

Race Conditions

In situations, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions.

Race Conditions

```
x=0;
```

```
Thread 1:

x = x+1;
/*
ld r1,x
add r1,1
st r1,x
*/
```

```
Thread 2:

x = x+1;
/*
ld r1,x
add r1,1
st r1,x
*/
```

Final value of x could be 1 or depending on the interleaving.

Inter Process Communications (IPC)

Protect Shared Variables

If a variable is written by many threads, then:

- Each write should be atomic (nothing can interfere).
- Ensure other threads don't interfere with a sequence of instructions.
- Associate a mutex or a lock with this variable.

Inter Process Communications (IPC)

Protect Shared Variables

```
x=0;
```

```
Thread 1:

lock(mutex);
x = x+1;
/*
ld r1,x
add r1,1
st r1,x
*/
unlock(mutex);
```

```
Thread 2:

lock(mutex);
x = x+1;
/*
ld r1,x
add r1,1
st r1,x
*/
unlock(mutex);
```

Inter Process Communications (IPC)

Does this need a mutual exclusion?

```
My_balance=10;
Your_balance=10;
```

Thread 1:

```
My_balance=My_balance+1;
Your_balance=Your_balnce-1;
```

Thread 2:

```
Total=My_balance
+Your_balance;
```

Inter Process Communications (IPC)

Does this need a mutual exclusion?

```
My_balance=10;
Your_balance=10;
```

Thread 1:

```
My_balance=My_balance+1;
Your_balance=Your_balnce-1;
```

Thread 2:

```
Total=My_balance
+Your_balance;
```

Yes. Interleaving may cause thread 2 to see an inconsistent state.

Inter Process Communications (IPC)

Code with mutual exclusion?

Mutex b_mutex protects My_balance and Your_balance shared variables.

```
My_balance=10;
Your_balance=10;
```

```
Thread 1:
lock(b_mutex);

My_balance=My_balance+1;
Your_balance=Your_balnce-1;

unlock(b_mutex);
```

```
Thread 2:
lock(b_mutex);

Total=My_balance
+Your_balance;

unlock(b_mutex);
```

Inter Process Communications (IPC)

Does this need a mutual exclusion?

```
x=0;

Thread 1:                         Thread 2:

x = x+1;                          y = x;
```

Inter Process Communications (IPC)

Does this need a mutual exclusion?

```
x=0;

Thread 1:          Thread 2:

x = x+1;           y = x;
```

No. Reads and writes are atomic

Inter Process Communications (IPC)

Critical Regions

When a process has to access shared memory or files, or do other critical things that can lead to races. That part of the program where the shared memory is accessed is called the **critical region or critical section**.

Inter Process Communications (IPC)

Mutex and Critical Section

Only one thread can be in the critical section at a time.

```
Thread 1:

Enter_Mutex();
Critical_Section();
Exit_Mutex();
```

```
Thread 2:

Enter_Mutex();
Critical_Section();
Exit_Mutex();
```

Inter Process Communications (IPC)

Race Condition Solution

To have a good solution for race conditions, four conditions have to be hold:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block any process.
4. No process should have to wait forever to enter its critical region.

Inter Process Communications (IPC)

The Problem

How to implement Enter_Mutex() and Exit_Mutex()?:

- User space (no OS support).
- Inside the kernel.

Inter Process Communications (IPC)

Mutual Exclusion with busy waiting

To Achieve mutual exclusion there are several proposals:

- Disabling interrupts.
- Lock variables.
- Strict alternation.
- Peterson's solution.
- The TSL instruction.

Inter Process Communications (IPC)

Disabling interrupts

```
Disable interrupts
Set Lock
If (unsuccessful) then
    Enable interrupts
    Try again
Enable Interrupts
```

Inter Process Communications (IPC)

Disabling interrupts

Disabling interrupts is often a useful technique within the operating system itself but is not appropriate as a general mutual exclusion mechanism for user processes. It is not wise to let a user process disable interrupts and it cannot work with multiprocessors architecture.

Inter Process Communications (IPC)

Lock variables

```
                lock=0;
Thread 1                            Thread 2
Enter_Mutex:                        Enter_Mutex:
    while(lock==1);                     while(lock==1);
    lock=1;                             lock=1;

Critical_Section:                   Critical_Section:
    account++;                          account++;

Exit_Mutex:                         Exit_Mutex:
    lock=0;                             lock=0;
```

Problems?

Inter Process Communications (IPC)

Lock variables

```
                lock=0;
Thread 1                            Thread 2
Enter_Mutex:                        Enter_Mutex:
    while(lock==1);                     while(lock==1);
    lock=1;                             lock=1;

Critical_Section:                   Critical_Section:
    account++;                          account++;

Exit_Mutex:                         Exit_Mutex:
    lock=0;                             lock=0;
```

Problems?

Still have a race condition within the lock update itself.

Inter Process Communications (IPC)

Strict alternation

```
                 turn=1;
Thread 1                         Thread 2
Enter_Mutex:                     Enter_Mutex:
    while(turn==2);                  while(turn==1);

Critical_Region:                 Critical_Region:
    account++;                       account++;

Exit_Mutex:                      Exit_Mutex:
    turn=2;                          turn=1;
```

A lock that uses busy waiting is called a spin lock.

Problems?

Inter Process Communications (IPC)

Strict alternation

```
                 turn=1;
Thread 1                          Thread 2
Enter_Mutex:                      Enter_Mutex:
    while(turn==2);                   while(turn==1);

Critical_Region:                  Critical_Region:
    account++;                        account++;

Exit_Mutex:                       Exit_Mutex:
    turn=2;                           turn=1;
```

A lock that uses busy waiting is called a spin lock.

Problems?

This construction at some point violates condition 3 and 4.

Inter Process Communications (IPC)

Peterson's Solution

```
#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 */
}
```

Inter Process Communications (IPC)

The TSL instruction

`TSL R, Lock`

- **TSL:** Test and Set Lock
- **R:** Register
- **Lock:** Memory location

This instruction atomically:

- Read Lock into R
- Store a non zero value in Lock

Inter Process Communications (IPC)

The TSL instruction

```
Enter_Mutex:
    TSL Reg, Lock
    If (Reg != 0) then
        Jump to Enter_Mutex

Critical_Section:
    account++

Exit_Mutex:
    Lock=0
```

Inter Process Communications (IPC)

To solve busy waiting problem

Yield to another thread if unable to lock first time

Inter Process Communications (IPC)

Eliminate busy waiting

```
Enter_Mutex:
    TSL Reg, Lock
    If (Reg != 0) then
        thread_yield()
        Jump to Enter_Mutex

Critical_Section:
    account++

Exit_Mutex:
    Lock=0
```

Inter Process Communications (IPC)

Disabling interrupts

```
Enter_Mutex:
    Disable interrupts
    Set Lock
    If (unsuccessful) then
        Enable interrupts
        thread_yield()
        Try again
    Enable Interrupts
```

Inter Process Communications (IPC)

Disabling interrupts

- Disabling interrupts during critical section:
  - Prevent context switch
  - Protect shared variables
- Then enable interrupts after critical section

No busy waiting

Problems?

Inter Process Communications (IPC)

Disabling interrupts - problems

- Critical section must be short, because no multiprogramming is possible during critical section.
- Cannot trust users to have a short critical section.

Used inside the kernel for mutual exclusion.

Inter Process Communications (IPC)

Disabling interrupts - multiprocessors

Disabling interrupts doesn't work, because preventing context switch doesn't ensure only one process is running.

Solution

Use hardware support (TSL instruction). Usually multiprocessors architecture comes equipped with this instruction.

Inter Process Communications (IPC)

POSIX mutexes

```
#include<pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
        const pthread_mutexattr_t *mutexattr);
pthread_mutex_t mutex= PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

1. sudo apt-get install manpages-posix-dev
2. man pthread_mutex_init

Inter Process Communications (IPC)

POSIX mutexes example

```
//shared by all threads
pthread_mutex_t m= PTHREAD_MUTEX_INITIALIZER;

int x=0;

pthread_mutex_lock(&m);
x=x+1;
pthread_mutex_unlock(&m);
```

Inter Process Communications (IPC)

Taking multiple locks

```
Thread 1
proc1(){
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);

  /* use objects 1 and 2 */

  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}
```

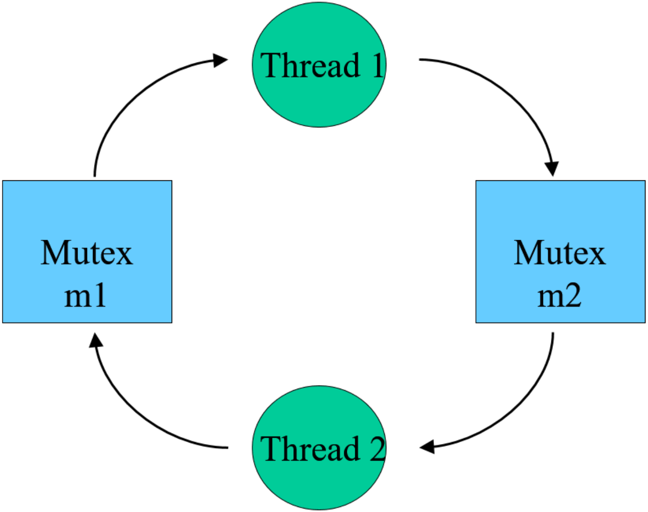
```
Thread 2
proc2(){
  pthread_mutex_lock(&m2);
  /* use object 2 */
  pthread_mutex_lock(&m1);

  /* use objects 1 and 2 */

  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
}
```

Inter Process Communications (IPC)

Deadlock (Resources graph)



Inter Process Communications (IPC)

Deadlock conditions

Mutual exclusion condition: Each resource assigned to 1 process or is available

Hold and wait condition: Process holding resources can request additional

No preemption condition: Previously granted resources cannot forcibly taken away

Circular wait condition: Must be a circular chain of 2 or more processes each is waiting for resource held by next member of the chain