
High Performance Computing

Lecture 19

Matthew Jacob

Indian Institute of Science

Exceptions

- Certain exceptional events that occur during program execution, handled by the processor HW
- There are two kinds of exceptions
 1. **Traps**: Synchronous, software generated
 - Page fault, Divide by zero, System call
 2. **Interrupts**: Asynchronous, hardware generated
 - Timer, keyboard, disk

What Happens on an Exception

1. Hardware

- Saves processor state
- Transfers control to corresponding piece of OS code, called the **exception handler**

2. Software (exception handler)

- Takes care of the situation as appropriate
- Ends with **return from exception** instruction

3. Hardware (execution of RFE instruction)

- Restores the saved processor state
- Transfers control back to the saved PC value

Re-look at Process Lifetime

- Which process has the exception handling time accounted against it?
 - The process running at the time of the exception
- All interrupt handling time while process is in running state is accounted against it
 - As part of `running in system mode`

Concurrent Programming

- Until now: Program execution involved one flow of control through the program
- Concurrent programming is about programs with multiple flows of control
- For example: a program that runs as multiple processes cooperating to achieve a common goal
- To cooperate, processes must somehow communicate

Inter Process Communication (IPC)

1. Processes can communicate using files
 - ❑ Example: Communication between parent process and child process
 - ❑ Parent process creates 2 files before forking child process
 - ❑ Child inherits file descriptors from parent, and they share the file pointers
 - ❑ Can use one for parent to write and child to read, other for child to write and parent to read

Inter Process Communication (IPC)

1. Processes can communicate using files
2. OS supports something called a **pipe**
 - ❑ corresponds to 2 file descriptors (`int fd[2]`)
 - ❑ Read from `fd[0]` accesses data written to `fd[1]` in FIFO (First In First Out) order and vice versa

Other IPC Mechanisms

1. Processes can communicate using files
2. OS supports something called a pipe
3. Processes could communicate through variables that are shared between them
 - ❑ **Shared variables**, shared memory; other variables are **private** to a process
 - ❑ Special OS support for program to specify objects that are to be in shared regions of address space

Virtual Memory & Shared Variables ??

- Address translation is used to protect one process from another
 - Each process uses virtual addresses (0 .. 2^n-1)
 - Then, how can 2 processes share a variable?

Other IPC Mechanisms

1. Processes can communicate using files
2. OS supports something called a pipe
3. Processes could communicate through variables that are shared between them
4. Processes could communicate by sending and receiving **messages** to each other
 - ❑ Special OS support for these messages

More Ideas on IPC Mechanisms

5. Sometimes processes don't need to communicate explicit values to cooperate
 - ❑ They might just have to synchronize their activities
 - ❑ Example: Process 1 reads 2 matrices, Process 2 multiplies them, Process 3 writes the result matrix
 - ❑ Process 2 should not start work until Process 1 finishes reading, etc.
 - ❑ Called process **synchronization**
 - ❑ Synchronization primitives
 - Examples: **mutex lock**, **semaphore**, **barrier**

Programming With Shared Variables

- Consider a 2 process program in which both processes increment a shared variable

shared int X = 0;

P1:

X++;

P2:

X++;

- Q: What is the value of X after this?
- Complication: Remember that X++ compiles into something like

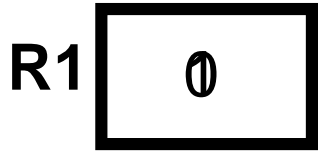
LW R1, 0(R2)

ADD R1, R1, 1

SW 0(R2), R1

shared int X = 0;

Process P1



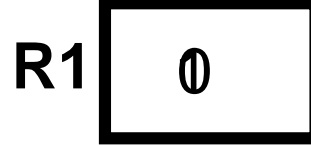
X++

LW R1, X

Context Switch

ADD R1, R1, 1
SW X, R1

Process P2



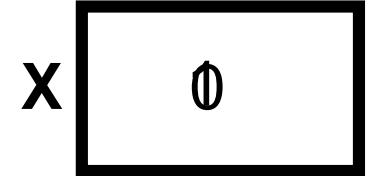
X++

LW R1, X

ADD R1, R1, 1

SW X, R1

Context Switch



LW R1, X

ADD R1, R1, 1

SW X, R1

*FINAL
VALUE
OF X
COULD
BE 1*

time

Problem with using shared variables

- Final value of X could be 1!
P1 loads X into R1, increments R1
P2 loads X into register before P1 stores new value into X
Net result: P1 stores 1, P2 stores 1
- Moral of example: Necessary to synchronize processes that are interacting using shared variables
- Problem arises when 2 or more processes try to update shared variable
- **Critical Section**: part of program where shared variable is accessed like this

Critical Section Problem: Mutual Exclusion

- Must synchronize processes so that they access shared variable one at a time in critical section; called **Mutual Exclusion**
- Mutex Lock: a synchronization primitive
 - AcquireLock(L)
 - Done before critical section of code
 - Returns when safe for process to enter critical section
 - ReleaseLock(L)
 - Done after critical section
 - Allows another process to acquire lock

Implementing a Lock

```
int L=0;          /* 0: lock available */
```

```
AcquireLock(L):
```

```
    while (L==1); /* `BUSY WAITING' */
```

```
    L = 1;
```

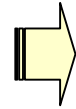
```
ReleaseLock(L):
```

```
    L = 0;
```

Why this implementation fails

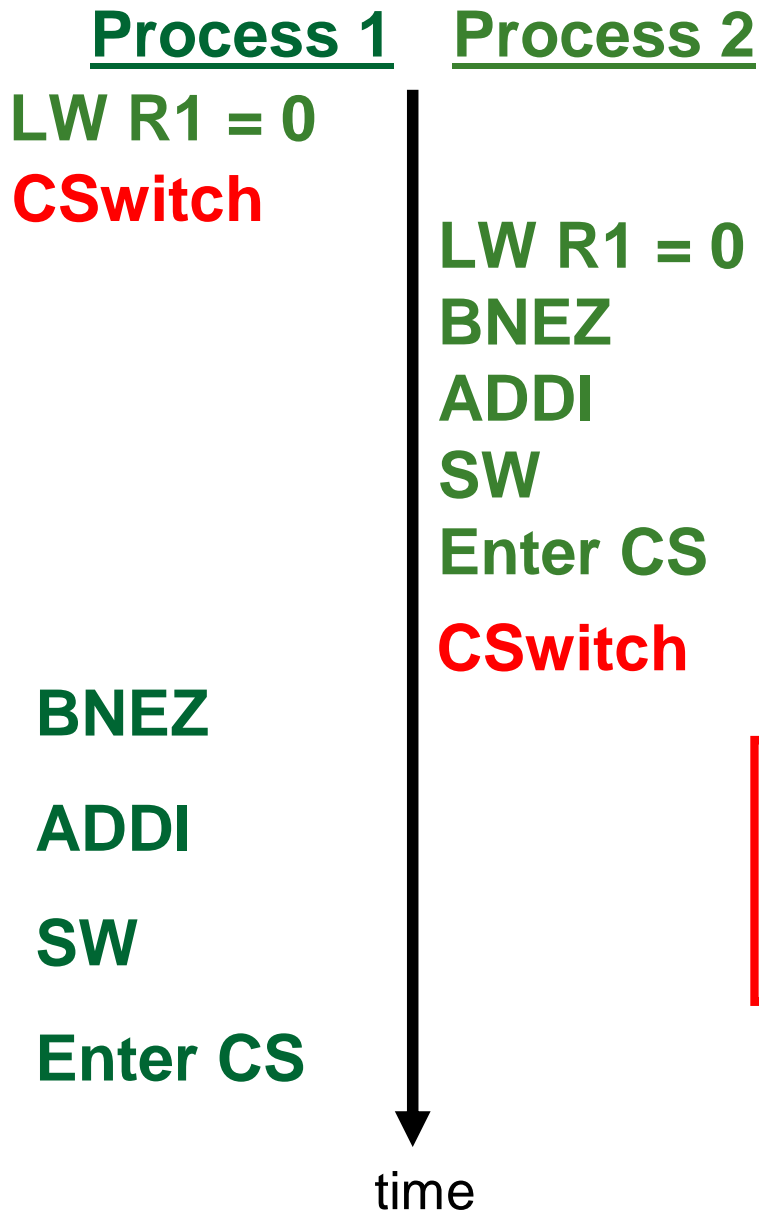
```
while ( L == 1 ) ;
```

```
L = 1;
```



```
wait: LW    R1, Addr(L)
      BNEZ R1, wait
      ADDI R1, R0, 1
      SW    R1, Addr(L)
```

Why this implementation fails



```
wait: LW    R1, Addr(L)
      BNEZ  R1, wait
      ADDI  R1, R0, 1
      SW    R1, Addr(L)
```

Assume that lock L is currently available ($L = 0$) and that 2 processes, P1 and P2 try to acquire the lock L

IMPLEMENTATION ALLOWS PROCESSES P1 and P2 TO BE IN CRITICAL SECTION TOGETHER!