University of Central Punjab

FOIT (Operating Systems)
GL- 9
Synchronization
Software & Hardware Solutions

Objectives

- Students will be able to realize the critical section problem.
- Students will be able to identify the critical section in a program.
- Students will learn the software and various hardware solutions to critical section problem and their validity.
- Students will be able to solve some basic synchronization problems using the hardware solution.
- Students will learn how do the libraries implement a solution to the critical section problem using a hardware solution.

Observation: The shared variable "Count" is incremented and decremented for the same number of times using threads. The value of "Count" should be the same. Run the given code and check if it remains the same?

Incrementing and decrementing Count variable equal number of times simultaneously.

```
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<wait.h>
#include<pthread.h>

unsigned int count;
void* increment (void* ptr)
{
     for(unsigned int i=0;i<500000;i++)
     count++;
}
void* decrement (void* ptr)
{
     for(unsigned int i=0;i<500000;i++)
     count--;
}
int main()
{
count = 3;
printf("\nInitial value of Count is %d ",count);
pthread_t pid1,pid2;
pthread_create(&pid1,NULL,&increment,NULL);
pthread_create(&pid2,NULL,&decrement,NULL);
pthread_join(pid1,NULL);
pthread_join(pid2,NULL);
printf("\nFinal value of Count is %d ",count);
printf("\nIs it correct? \n");
return 0;
}
```

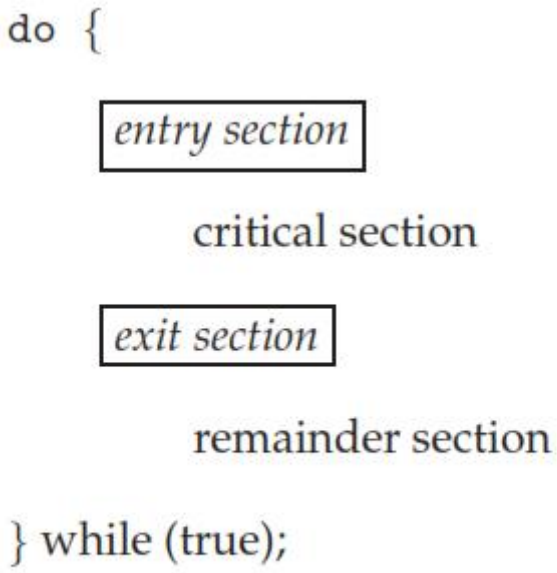
➔ **Run and submit the screenshot of the output of this program**

The difference in the value of "Count" variable after performing an equal number of increment and decrement operations is due to the fact that the shared variable can't be modified simultaneously. If we serialize the operation, the problem will

be solved. E.g. We increment Count variable first and then decrement it, we will get the same value. However, incrementing and decrementing simultaneously would result in unexpected results.

The section in a program where a shared variable is modified called "Critical Section" and the problem is known as "Critical Section Problem".

The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder Section. The general structure of a typical process P0, P1,...,Pn-1 is shown in the following Figure. The entry section and exit section are enclosed in boxes to highlight these important segments of code.



General structure of a typical process P_i.

A solution to a critical section problem must satisfy the following three requirements:

Mutual exclusion.
If process *P* is executing in its critical section, then no other processes can be executing in their critical sections.

Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

Bounded waiting. There exists a bound, or limit, 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.

The solution to the Critical Section Problem

There are three types of solutions for critical section problem

- Software solution
- Hardware Solution
- Library or OS provided Solutions (not covered in this lab)

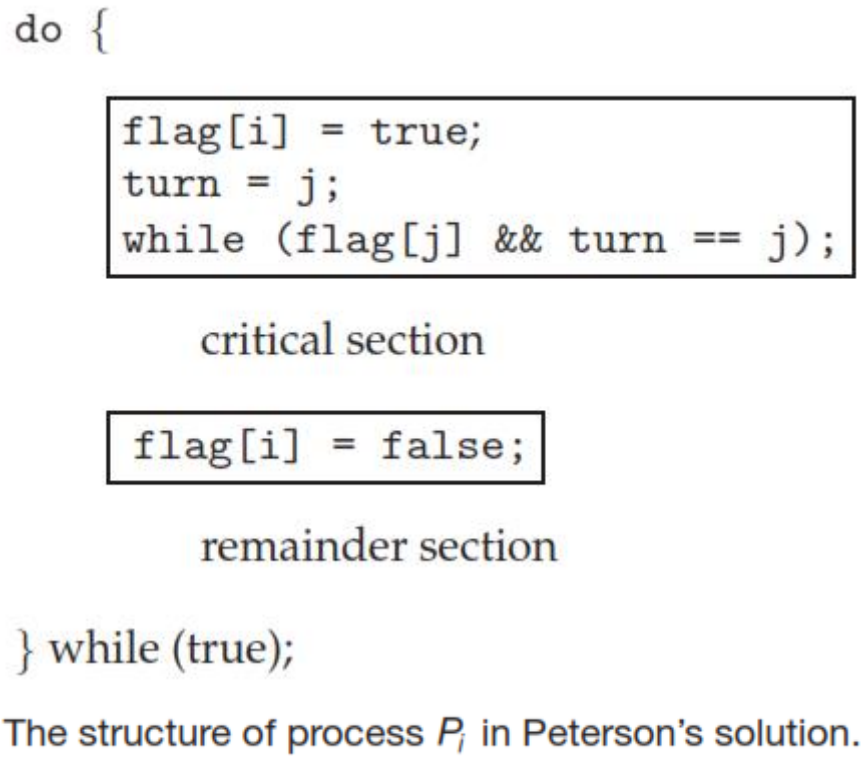
Software Solution

Peterson solution

Peterson's solution is restricted to two processes that alternate execution between their critical sections and the remainder sections. The processes are numbered P0 and P1. For convenience, when presenting P to denote the other process; that is, j equals 1 - i.
Peterson's solution requires the two processes to share two data items:

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

The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section. The flag array is used to indicate if a process is ready to enter its critical section. For example, if flag[i] is true, this value indicates that Pis ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in Figure shown below.



The structure of process P_i in Peterson's solution.

To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

The Peterson solution meets all of the three conditions mentioned above.

Incrementing and decrementing Count variable equal number of times simultaneously using **Peterson** solution

`#include<stdio.h>`
`#include<pthread.h>`
 	
`int flag[2]={0,0};`
`int turn=0;`
 	
`long count;`
`void* increment (void* ptr)`

```
{
      for(unsigned int i=0;i<1000000;i++)
      {
            flag[0] = 1;
            turn = 1;
            while(flag[1]==1 && turn==1){}
            count++;
            flag[0]=0;
      }
      pthread_exit(0);
}
void* decrement (void* ptr)
{
      for(unsigned int i=0;i<1000000;i++)
      {
            flag[1] = 1;
            turn = 0;
            while(flag[0]==1 && turn==0){}
            count--;
            flag[1] = 0;
      }
      pthread_exit(0);
}
int main()
{

count = 100;
pthread_t pid1,pid2;

printf("\nInitial value of Count is %ld ",count);
pthread_create(&pid1,NULL,&increment,NULL);
pthread_create(&pid2,NULL,&decrement,NULL);

pthread_join(pid1,NULL);
pthread_join(pid2,NULL);

printf("\nFinal value of Count is %ld ",count);
printf("\nIs it correct? \n");
return 0;
}
```

➔ **Run and submit the screenshot of the output of this program**

Does Peterson's solution work? If no, why?

Most modern CPUs reorder memory accesses to improve execution efficiency. Such processors invariably give some way to force

ordering in a stream of memory accesses, typically through a memory barrier instruction. Implementation of Peterson's and related algorithms on processors that reorder memory accesses generally requires the use of such operations to work correctly to keep sequential operations from happening in an incorrect order. Note that reordering of memory accesses can happen even on processors that don't reorder instructions.

Hardware Solution

Disable Hardware Interrupts

Hardware interrupts are disabled by clearing the interrupt flag in a flag register. "CLI" instruction is used to clear the interrupt flag. Let's see if the "CLI" instruction executes successfully.

Disabling Hardware Interrupts Programmatically.

```
#include<stdio.h>

int main()
{
printf("Hello!, This program is going to clear carry flag.\n");
asm("clc;"); /* clear the carry flag using inline assembly */

printf("Carry flag cleared!\n");

printf("This program is attempting to disable interrupts using
CLI (Clear Interrupt Flag) instruction.\n");

printf("If you see segmentation fault, process is
trapped.\n\n\n");

asm("cli;"); /* inline assembly instruction to clear interrupt
flag */

printf("\nInterrupts disbled!!!");
return 0;
}
```

➔ **Run and submit the screenshot of the output of this program**

Did it work?

No, because the instruction for clearing the interrupt flag is a privileged instruction and is trapped if executed in user-

mode. However, the instruction to clear the carry flag is not privileged.

Compare and Set instruction

Many modern computer systems provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically**—that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner.

"**CMPXCHG**" is an instruction that was introduced in the Intel486 microprocessor for the first time to cater to the critical section problem. It is used with a **lock** prefix for atomic execution. The following figure shows its operation.

Format	**CMPXCHG DEST, SRC** *DEST can be variable or register whereas SRC will be a register*
Operation	TEMP ← DEST IF accumulator = TEMP THEN ZF ← 1; DEST ← SRC; ELSE ZF ← 0; accumulator ← TEMP; DEST ← TEMP; FI;

Incrementing and decrementing Count variable equal number of times simultaneously using **Compare&set (hardware)** solution.

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

long compare_and_set(long *flag, long expected, long new);
     asm(  /* compile with 64-bit compiler*/
     "compare_and_set:;"
     "movq %rsi,%rax;"
     "lock cmpxchgq %rdx,(%rdi);"
     "retq;"
     );

long count, flag=0, expected=0, new=1;
void* increment (void* ptr)
{
     for(unsigned int i=0;i<2000000;i++)
     {
          while(compare_and_set(&flag,expected,new)!=0);
          count++;
          flag=0;
     }
     pthread_exit(0);
}
void* decrement (void* ptr)
{
     for(unsigned int i=0;i<2000000;i++)
     {
          while(compare_and_set(&flag,expected,new)!=0);
          count--;
          flag = 0;
     }
     pthread_exit(0);
}
int main()
{
count = 100;
pthread_t pid1,pid2;

printf("\nInitial value of Count is %ld ",count);
pthread_create(&pid1,NULL,&increment,NULL);
pthread_create(&pid2,NULL,&decrement,NULL);

pthread_join(pid1,NULL);
pthread_join(pid2,NULL);

printf("\nFinal value of Count is %ld ",count);
```

```
printf("\nIs it correct? \n");
return 0;
}
```

➔ **Run and submit the screenshot of the output of this program**

Did it work?

Yes. But the incorporation of assembly language instructions in a high-level language code is not only difficult to manage but also makes the code unportable.

Usually, libraries that implement Mutex or Semaphore locks make use of this hardware solution. You can also write your own functions equivalent to mutex and semaphore locks.

University of Central Punjab

FOIT (Operating Systems)
GL- 9
Synchronization
Software & Hardware Solutions

Graded Tasks

Task1 (To be submitted in the lab)

Write a program that creates two threads. Thread #1 sends 10 integer numbers through a shared variable to Thread #2. Thread #2 receives the numbers and displays them on the screen. The numbers should be received and displayed in the same order they were sent. Use the hardware solution for synchronization.

Thread #2 should read a shared variable only if it has been written by Thread #1. Otherwise, it should wait. Similarly, Thread #1 should write to the shared variable only if it has been read by Thread #2. You may use a flag variable to implement this check.

Task 2 (to be submitted in lab)

Write a program that creates ten threads to sum an array of 1000 elements. Instead of returning a partial sum from each thread, each thread should accumulate its partial sum into the shared variable 'sum', ensuring that only one thread writes to it at a time.