Peterson's Algorithm

Carlos Alberto Rincón Castro | carincon@uh.edu

Lecture Overview

- **Concurrency:** An example
- **Memory sharing in UNIX**
- **Peterson Algorithm:**
  - First attempt.
  - Second attempt.
  - Third attempt.
  - Fourth attempt.
  - Correct solution.

Concurrency: An example

Process Code

```
void echo()
{
        chin = getchar();
        chout = chin;
        putchar(chout);
}
```

Example: Executing two instances of the process at the same time

Process A

```
void echo()
{
        chin = getchar();
        chout = chin;
        putchar(chout);
}
```

Process B

```
void echo()
{
        chin = getchar();
        chout = chin;
        putchar(chout);
}
```

Concurrency: An example

Assuming that chin is a shared variable:

Process A

```
void echo()
{
        (1) chin = getchar();

        (3) chout = chin;
        (4) putchar(chout);
}
```

Process B

```
void echo()
{

        (2) chin = getchar();


        (5) chout = chin;
        (6) putchar(chout);
}
```

What is the output of process A and process B, if the getchar function from A receives a ‘C’ and the getchar function from B receives a ‘W’

Memory Sharing in UNIX

DEMO

Peterson's Algorithm

FIRST ATTEMPT

Assumption: Only one access to a memory location can be made at a time.
Global memory location "turn" is reserved for shared variable

turn = 0

Process 0

```
{
        while (turn != 0)
        /*do nothing*/;
        /*CS*/
        turn = 1;
}
```

Process 1

```
{
        while (turn != 1)
        /*do nothing*/;
        /*CS*/
        turn = 0;
}
```

FIRST ATTEMPT

- Guarantees Mutual Exclusion.
- Has two problems:
  - Processes must strictly alternate in their use of their CS; pace is dictated by the slower process.
  - If one process fails, the other one is permanently blocked; whether in CS or not.

Peterson's Algorithm

SECOND ATTEMPT

Need state information about both processes. flag[0] for P0 and flag[1] for P1
(Boolean vector flag; when one fails, the other can still access CS)
Each process may examine the other's flag, but may not alter it…

enum boolean {FALSE=0; TRUE=1};
boolean flag[2] = {FALSE, FALSE};

Process 0

```
{
        while (flag[1])
        /*do nothing*/;
        flag[0]=TRUE;
        /*CS*/
        flag[0]=FALSE;
}
```

Process 1

```
{
        while (flag[0])
        /*do nothing*/;
        flag[1]=TRUE;
        /*CS*/
        flag[1]=FALSE;
}
```

Peterson's Algorithm

SECOND ATTEMPT

- Does not Guarantees Mutual Exclusion.

A process can change its state after the other process has checked it, but before the other process can enter into critical section.

Process 0

{

(1) while (flag[1])
/*do nothing*/;

(3) flag[0]=TRUE;

/*CS*/
flag[0]=FALSE;

}

Process 1

{

(2) while (flag[0])
/*do nothing*/;

(4) flag[1]=TRUE;
/*CS*/
flag[1]=FALSE;

}

Peterson's Algorithm

THIRD ATTEMPT

Need state information about both processes. flag[0] for P0 and flag[1] for P1
(Boolean vector flag; when one fails, the other can still access CS)
Each process may examine the other's flag, but may not alter it…

enum boolean {FALSE=0; TRUE=1};
boolean flag[2] = {FALSE, FALSE};

Process 0

```
{
        flag[0]=TRUE;
        while (flag[1])
        /*do nothing*/;
        /*CS*/
        flag[0]=FALSE;
}
```

Process 1

```
{
        flag[1]=TRUE;
        while (flag[0])
        /*do nothing*/;
        /*CS*/
        flag[1]=FALSE;
}
```

Peterson’s Algorithm

THIRD ATTEMPT

- If both processes set their flags to TRUE at the same time, then they are in a loop for ever.

A process sets its flag without knowing other process’s status!!

Process 0

```
{
        (1) flag[0]=TRUE;

        (3) while (flag[1])
        /*do nothing*/;
        /*CS*/
        flag[0]=FALSE;

}
```

Process 1

```
{

        (2) flag[1]=TRUE;

        (4) while (flag[0])
        /*do nothing*/;
        /*CS*/
        flag[1]=FALSE;
}
```

Peterson's Algorithm

FOURTH ATTEMPT

Need state information about both processes. flag[0] for P0 and flag[1] for P1
(Boolean vector flag; when one fails, the other can still access CS)
Each process may examine the other's flag, but may not alter it…

enum boolean {FALSE=0; TRUE=1};
boolean flag[2] = {FALSE, FALSE};

Process 0

```
{
        flag[0]=TRUE;
        while (flag[1])
        {
                flag[0] = FALSE;
                /* delay */;
                flag[0] = TRUE;
        }
        /*CS*/
        flag[0]=FALSE;
}
```

Process 1

```
{
        flag[1]=TRUE;
        while (flag[0])
        {
                flag[1] = FALSE;
                /* delay */;
                flag[1] = TRUE;
        }
        /*CS*/
        flag[1]=FALSE;
}
```

FOURTH ATTEMPT

A possible execution:

P0 sets flag[0] to TRUE
P1 sets flag[1] to TRUE
P0 checks flag[1] FALSE
P1 checks flag[0] FALSE
P0 sets flag[0] TRUE
P1 sets flag[1] TRUE

- The above sequences could be extended indefinitely.
- Neither process could get into CS It is not a deadlock! It is a livelock!
- Any alteration in relative speeds of processes could make one process enter into CS.

Peterson's Algorithm

The Correct Solution

Need to observe the state of both processes, which process has the right to insist on entering into CS.

boolean flag[2];

int turn;

Process 0

```
{
        flag[0]=TRUE;
        turn =1;
        while (flag[1] && turn == 1)
                /* do nothing */;
        /*CS*/
        flag[0]=FALSE;
}
```

Process 1

```
{
        flag[1]=TRUE;
        turn = 0;
        while (flag[0] && turn == 0)
                /* do nothing */;
        /*CS*/
        flag[1]=FALSE;
}
```