Kemerburgaz University
Istanbul, Turkey
Information Technology department

The project topic is:

Peterson’s algorithm semaphores implementation

Prepared By : Laila Gouma Bawendi

ID:163110443

OUTLINES

- what is Peterson’s algorithm.
- • Peterson’s General algorithms.
- • What is Semaphores.
- • Semaphores implementation.
- • Semaphore Implementation Busy waiting.
- • semaphores implementation for solving critical section problem.

what is peterson’s algorithms ?

- Peterson's algorithm (or Peterson's solution) is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication. It was formulated by Gary L. Peterson in 1981. While Peterson's original formulation worked with only two processes, the algorithm can be generalized for more than two.

Peterson's Solution

- The algorithm satisfies the three essential criteria to solve the critical section problem, provided that changes to the variables turn, flag[0], and flag[1] propagate immediately and atomically. The while condition works even with preemption.
- The three criteria are mutual exclusion, progress, and bounded waiting.
- Since turn can take on one of two values, it can be replaced by a single bit, meaning that the algorithms requires only three bits of memory.

Peterson's General algorithms.

```
bool flag[2] = {F;F};
int turn;
P0:       flag[0] = true;
P0_gate: turn = 1;
          while (flag[1] && turn == 1)
          {
              // busy wait
          }
          // critical section
          ...
          // end of critical section
          flag[0] = false;
```

```
P1:       flag[1] = true;
P1_gate: turn = 0;
          while (flag[0] && turn == 0)
          {
              // busy wait
          }
          // critical section
          ...
          // end of critical section
          flag[1] = false;
```

Semaphores

- **A Semaphore S is an integer variable that, apart from initialization, can only be accessed through 2 *atomic and mutually exclusive*.**

- two main operations:
  - **wait** (or **acquire**)
  - **signal** (or **release**)

Busy Waiting Semaphores

- The simplest way to implement semaphores.
- Useful when critical sections last for a short time, or we have lots of CPUs.
- S initialized to positive value (to allow someone in the beginning).

```
wait(S):
 while S<=0 do ;
 S--;
```

```
signal(S):
 S++;
```

Using semaphores for solving critical section problems

- For n processes
- Initialize semaphore “mutex” to 1
- Then only one process is allowed into CS (mutual exclusion)
- To allow k processes into CS at a time, simply initialize mutex to k

```
Process Pi:
repeat
   wait(mutex);
    CS
   signal(mutex);
    RS
forever
```

Synchronizing Processes using Semaphores

- **Two processes:**
  - P_1 and P_2
- **Statement S_1 in P_1 needs to be performed before statement S_2 in P_2**
- **We want a way to make P_2 wait**
  - until P_1 tells it is OK to proceed

- **Define a semaphore "synch"**
  - Initialize synch to 0
- **Put this in P_2:**

  wait(synch);

  S_2;
- **And this in in P_1:**

  S_1;

  signal(synch);

Busy-Waiting Semaphores: Observations

- **When S>0:**
  - the number of processes that can execute wait(S) without being blocked = S
- **When S=0: one or more processes are waiting on S**
- **Semaphore is never negative**
- **When S becomes >0, the first process that tests S enters enters its CS**
  - random selection (a race)
  - fails bounded waiting condition

- THANK YOU