OPERATING SYSTEMS

Module3_Part2

Textbook : Operating Systems Concepts by Silberschatz

Process synchronisation

- when several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition.**

To avoid race condition, processes are to be synchronized in some way.

To do process synchronization, we should first to know what a **critical section** is.

Critical section

Consider a system consisting of *n* processes {Po, P1 , ... , *Pn* _ I}.

Each process has a segment of code, called a critical section in which the process may be changing common variables, updating a table, writing a file, and so on.(shared)

Part of the program where the shared memory is accessed is called **critical section** or critical region of **that process**

when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. If so we can avoid race conditions.

Critical section

- 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 in

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (TRUE);
```

Critical section

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

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

2. **Progress.** If no process is executing in its critical section 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, this selection cannot be postponed indefinitely.

3. **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.

Peterson’s solution

One of the process synchronization mechanism

- Peterson's solution is a classic software-based solution to the critical-section problem

- it provides a good algorithmic description of solving the critical-section problem

- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.

The processes are numbered *Po* and P1.

Peterson’s solution

- The two processes share two variables:
  - `int turn;`
  - `boolean flag[2]`
- The variable `turn` indicates whose turn it is to enter the critical section
- The `flag` array is used to indicate if a process is ready to enter the critical section.
  - `flag[i] = true` implies that process $\mathbf{P_i}$ is ready!

Algorithm for Process P_i

```
while (true){

        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
                ;

           /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

Structure for p0

Initially flag[0] and flag[1] assigns false

```
while (true){
        flag[0] = true;
        turn = 1;
        while (flag[1] && turn =
= 1)
                ;

/* critical section */

        flag[0] = false;

        /* remainder section */
}
```

structure for p1

```
while (true){
        flag[1] = true;
        turn = 0;
        while (flag[0] && turn =
= 0)
                ;

/* critical section */

        flag[1] = false;

        /* remainder section */
}
```

Peterson’s solution

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

1. **Mutual exclusion.**
2. **Progress.**
3. **Bounded waiting.**

Structure for p0

```
while (true){
        flag[0] = true;
        turn = 1;
        while (flag[1] && turn =
= 1)
                ;

/* critical section */

        flag[0] = false;

        /* remainder section */
}
```

structure for p1

```
while (true){
        flag[1] = true;
        turn = 0;
        while (flag[0] && turn =
= 0)
                ;

/* critical section */

        flag[1] = false;

        /* remainder section */
}
```

P0 and p1 willing
P1 in CS and p0 wants to enter
P0 in CS and p1 wants to enter
P0 wants to enter and P1 has no interest

Structure for p0

```
while (true){
        flag[0] = true;
        turn = 1;
        while (flag[1] && turn =
= 1)
                ;

/* critical section */

        flag[0] = false;

        /* remainder section */
}
```

P0 wants to enter and P1 has no interest
If p0 wants to enter ,it is possible, we can say there is progress

structure for p1

```
while (true){
        flag[1] = true;
        turn = 0;
        while (flag[0] && turn =
= 0)
                ;

/* critical section */

        flag[1] = false;

        /* remainder section */
}
```

Structure for p0

```
while (true){
        flag[0] = true;
        turn = 1;
        while (flag[1] && turn =
= 1)
                ;

/* critical section */

        flag[0] = false;

        /* remainder section */
}
```

structure for p1

```
while (true){
        flag[1] = true;
        turn = 0;
        while (flag[0] && turn =
= 0)
                ;

/* critical section */

        flag[1] = false;

        /* remainder section */
}
```

In Peterson algorithm a process will never wait longer than one turn for entrance to the critical section