Process Synchronization

Peterson’s solution
&
Semaphore

Background

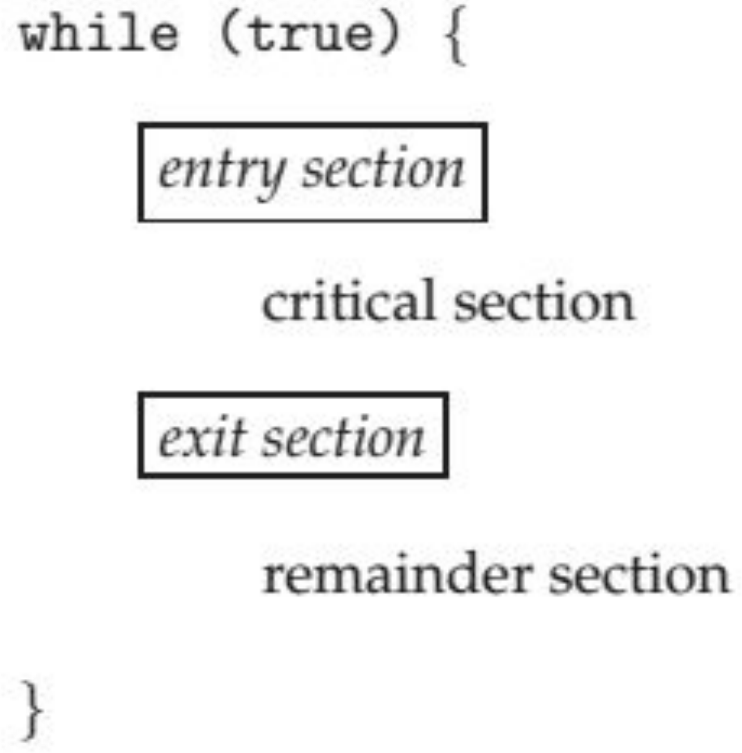
- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution.
- Concurrent access to **shared data** may result in data **inconsistency**.
- Maintaining data consistency requires mechanisms to ensure the **orderly execution** of cooperating processes.
- We illustrated previously the problem when we considered the **Bounded Buffer problem** with use of a **counter** that is updated concurrently by the producer and consumer; which lead to **race condition**.

Critical Section Problem

- Consider system of ***n*** processes {$\boldsymbol{p_0}$, $\boldsymbol{p_1}$, … $\boldsymbol{p_{n-1}}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design a protocol that the processes can use to cooperate.
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process $\boldsymbol{P_i}$



```
while (true) {

        entry section

            critical section

        exit section

            remainder section

}
```

Critical-Section Problem

Requirements for solution to critical-section problem

1. **Mutual Exclusion** - If process $\boldsymbol{P_i}$ 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 and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next 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.

Software Solution Algorithm1

- Two process solution
- The two processes share one variable:
  - *int **turn***;
- The variable ***turn*** indicates whose turn it is to enter the critical section

Algorithm for Process P_i

```
while (true){

   turn = i;
   while (turn = = j)
      ;

   /* critical section */

   turn = j;

   /* remainder section */

}
```

Algorithm 1

- Process P_0

```
hile (1) {
 while (turn != 0) ;
     critical section
 turn = 1;
     reminder section
```

- Process P_1

```
while (1) {
   while (turn != 1) ;
       critical section
   turn = 0;
       reminder section
}
```

- Satisfies mutual exclusion, but not progress

Correctness of Algorithm 1

- Mutual exclusion is preserved

  Pi enters critical section only if:

  turn = 1

  and turn cannot be both 0 and 1 at the same time
- What about the Progress requirement?
- What about the Bounded-waiting requirement?

Algorithm 2

- Shared variables
  - **boolean flag[2]**;
    initially **flag [0] = flag [1] = false.**
  - **flag [i] = true** $\Rightarrow$ P_i ready to enter its critical section

Algorithm 2

- Process P_0

```
while (1){
    flag[0] := true;
    while (flag[1]) ;                    critical
ection
    flag [0] = false;
        remainder section
```

- Process P_1

```
while (1){
    flag[1] := true;
    while (flag[0]) ;
critical section
    flag [1] = false;
        remainder section
        }
```

- Satisfies mutual exclusion, but not progress

Correctness of Algorithm 2

- Mutual exclusion is preserved
- What about the Progress requirement?
- What about the Bounded-waiting requirement?

Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Peterson's Solution
- Two process 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 */

}
```

Algorithm 3

- Process P_0

```
le (1) {
flag [0]:= true;
turn = 1;
while (flag [1]==1 and turn == 1) ;
     critical section
flag [0] = false;
     remainder section
```

- Process P_1

```
while (1) {
     flag [1]:= true;
     turn = 0;
     while (flag [0]==1 and turn == 0) ;
          critical section
     flag [1] = false;
          remainder section
}
```

Correctness of Algorithm 3

- Provable that the three CS requirement are met:
    1. Mutual exclusion is preserved

       Pi enters CS only if:

       either flag[j] = false or turn = i
    2. Progress requirement is satisfied
    3. Bounded-waiting requirement is met

- Meets all three requirements; solves the critical-section problem for two processes.
- Peterson’s Solution

Semaphore

- Semaphore is a synchronization tool that provides more sophisticated ways for processes to synchronize their activities.
- Semaphores are most commonly used for mutual exclusion to **solve** the **race condition** problem.
- Semaphore can deal with **N** process critical section problem.
- A semaphore is a non-negative integer with two access primitives, **wait** and **signal**.

Types of Semaphores

1) Counting semaphores:

   The value of a counting semaphore can range over an unrestricted domain.

. It is used to control access to a resource that has multiple instances.

2) Binary semaphore:

- The value of a binary semaphore can range only between 0 and 1.
- Its value is initialized to 1.
- The wait operation only works when the semaphore is 1.
- The signal operation succeeds when semaphore is 0.
- It is easier to implement binary semaphores than counting semaphores.

Semaphore

- Semaphore ***S*** can only be accessed via two indivisible (atomic) operations.

wait()

The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

signal()

The signal operation increments the value of its argument S.

- Both operations wait() and signal() are atomic.

Semaphore

- Definition of the **wait() operation**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definition of the **signal() operation**

```
signal(S) {
    S++;
}
```

Semaphore

Busy waiting:

- The repeated execution of a loop while waiting for an event to occur.
- waiting to enter its critical section.
- This continues to waste CPU cycles. The CPU is not engaged in any real productive activity during this period.

Semaphore

```
while (1)
{
  Entry section;
      critical section
  Exit section;
      remainder section
}
```

Semaphore

```
while (1)
{
  wait(S);
     critical section
  signal(S);
     remainder section
}
```

Semaphore

```
while (1)
{
  wait(S);
     critical section
  signal(S);
     remainder section
}
```

```
wait(S) {
   while (S <= 0)
     ; // busy wait
   S--;
}
```

```
signal(S) {
   S++;
}
```

Applications of Semaphore

- With semaphores we can solve various synchronization problems
- To decide the ordering of the processes
- Resource management when we have more than one copy of a resource.

Deciding order of execution

- Suppose, we have three processes, P1, P2 and P3.
- if we want to execute these processes in some order.
- For example: P3 → P1 → P2
    - Semaphores S1=S2=0

Wait(S1)	Wait(S2)	
P1	P2	P3
Signal(S2)		Signal(S1)

Resource Management

- Semaphores may also be used for system resource control.
- For example, suppose there are three printers to be shared by multiple processes.
- To ensure a printer is only being used by one process, we can use a semaphore to control the allocation of the printers.
- The printer semaphore is initialized three, the number of available shared system resources, in this case, printers.

Resource Management

```
while (1)
{
  Entry section;
     critical section
  Exit section;
     remainder section
}
```

Resource Management

```
while (1)
{
  wait(S);
     critical section
  Signal(S);
     remainder section
}
```

- Semaphore S must be initialized to 3.

Disadvantages of semaphore

- One of the biggest limitations is that semaphores may lead to priority inversion; where low priority processes may access the critical section first and high priority processes may access the critical section later.
- With improper use, a process may block indefinitely. Such a situation is called **Deadlock**.
- Another disadvantage of the semaphore definition that it require busy waiting.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.