



Operating Systems CS F372

Synchronization

BIJU K RAVEENDRAN

Bounded Buffer – Shared Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
```

```
typedef struct {
```

```
    ...
```


```
} item;
```

```
item buffer[BUFFER_SIZE];
```

```
int in = 0;
```

```
int out = 0;
```

```
int count = 0;
```



```
/* Producer */
```

```
while (true) {
```

```
    /* produce an item and put in nextProduced */
```

```
    while (count == BUFFER_SIZE); // do nothing
```

```
    buffer [in] = nextProduced;
```

```
    in = (in + 1) % BUFFER_SIZE;
```

```
    count++;
```

```
}
```

```
/* Consumer */
```

```
while (true) {
```

```
    while (count == 0); // do nothing
```

```
    nextConsumed = buffer[out];
```

```
    out = (out + 1) % BUFFER_SIZE;
```

```
    count--;
```

```
    /* consume the item in nextConsumed */
```

```
}
```

Producer – Consumer Problem

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes



Race Condition

- **count++** could be implemented as
register1 = count
register1 = register1 + 1
count = register1
- **count--** could be implemented as
register2 = count
register2 = register2 - 1
count = register2



Race Condition

Shared var int x =0;

Process #1

```
int main ( )  
{  
    x = x + 1;  
    return 0;  
}
```

Process #2

```
int main ( )  
{  
    x = x - 1;  
    return 0;  
}
```

Race Condition

$$\underline{x = x + 1}$$

Reg #1 \leftarrow x (load)
Reg#1 \leftarrow Reg#1 + 1 (compute)
x \leftarrow Reg #1 (store)

$$\underline{x = x - 1}$$

Reg #1 \leftarrow x (load)
Reg#1 \leftarrow Reg#1 - 1 (compute)
x \leftarrow Reg #1 (store)

Inter leaving can happen anywhere in this code

Maximum and Minimum values

- Race condition leads to unpredictable result
- Result can be with in a range
 - Previous example it is -1 to $+1$.
- Maximum value situation
 - If all $x - -$ operations getting nullify by $x + +$ operation
- Minimum value situation
 - If all $x + +$ operations getting nullify by $x - -$ operation

Race Condition

Shared var int x =0;

Process #1

```
int main ( )  
{ int i;  
  for(i=0;i<100;i++)  
    x = x + 1;  
  return 0;  
}
```

Process #2

```
int main ( )  
{ int i;  
  for(i=0;i<100;i++)  
    x = x - 1;  
  return 0;  
}
```

Find Maximum and Minimum value of x

Race Condition

Shared var int x =0;

Process #1

```
int main ( )  
{ int i;  
  for(i=0;i<100;i++) {  
    x = x + 1;  
    x = x - 1; }  
  return 0;  
}
```

Process #2

```
int main ( )  
{ int i;  
  for(i=0;i<100;i++) {  
    x = x + 1;  
    x = x - 1; }  
  return 0;
```

Find Maximum and Minimum value of x

Race Condition

- The value of shared variable in execution is dependent on the order of a shared variable or resource.
- Atomic operation
 - The execution of a bunch of operations are atomic if and only if the operation(s) either execute completely or the execution will not take place at all



Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \dots, p_{n-1}\}$
- 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 protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section



General Structure of a Process P_i

Only 2 processes say P_0 and P_1

General structure of Process P_i will be

do{

ENTRY SECTION

CRITICAL SECTION

EXIT SECTION

REMAINDER SECTION

} while(1);

Processes may share some common variable to
synchronize there action.

Solution to Critical Section Problem

1. Mutual Exclusion - If process 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 processes that will enter the critical section next cannot be postponed indefinitely
3. Bounded Waiting - A bound must exist 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
 - Assume that each process executes at a nonzero speed
 - No assumption concerning relative speed of the N processes



Algorithm 1

➤ Shared variables:

- **int** *turn*;
initially *turn* = 0
- $turn = i \Rightarrow P_i$ can enter its critical section

➤ Process P_i

do {

while ($turn \neq i$);

 critical section

$turn = i$;

 reminder section

}while(1);

Algorithm 1

Shared variable `int turn` (initial value 0)

Process 0

```
do {  
    while (turn != 0); // do nothing  
    <critical section>  
    turn = 1;  
    <reminder section >  
}while(1);
```

Process 1

```
do {  
    while (turn != 1); // do nothing  
    <critical section>  
    turn = 0;  
    <reminder section >  
}while(1);
```

➤ Satisfies mutual exclusion, but not progress

Algorithm 1

- This solution guarantees mutual exclusion
- Drawback 1: processes must strictly alternate
 - Pace of execution of one process is determined by pace of execution of other processes
- Drawback 2: if one processes fails other process is permanently blocked
- This problem arises due to fact that it stores name of the process that may enter critical section rather than the process state



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
- Process P_i
 - do {
 - $flag[i] = true$;
 - while** ($flag[j]$);
 - critical section
 - $flag[i] = false$;
 - remainder section
 - }while(1);

Algorithm 2

Shared variable Boolean flag[2] (flag[0] & flag[1] initial value false)

Process 0

```
do {  
    flag[0]=true;  
    while (flag[1]); // do nothing  
    <critical section>  
    flag[0]=false;  
    <reminder section >  
}while(1);
```

Process 1

```
do {  
    flag[1]=true;  
    while (flag[0]); // do nothing  
    <critical section>  
    flag[1]=false;  
    <reminder section >  
}while(1);
```

- Satisfies mutual exclusion, but not progress requirement.
- If flag[0]=flag[1]=true → infinite loop

Algorithm 2

- This approach Satisfy mutual exclusion
- This approach may lead to dead lock

What is wrong with this implementation?

- A process sets its state without knowing the state of other. Dead lock occurs because each process can insist on its right to enter critical section
- There is no opportunity to back off from this situation

