Process Synchronization

Organized By: Vinay Arora

Disclaimer

This is NOT A COPYRIGHT MATERIAL

<u>Content has been taken mainly from the following books</u>:

Operating Systems Concepts By Silberschatz & Galvin,

Operating systems By D M Dhamdhere,

System Programming By John J Donovan

etc…

Process & Synchronization

- Process – Program in Execution.
- Synchronization – Coordination.
- Independent process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Buffer

```
#define BUFFER_SIZE 10
typedef struct {
    DATA            data;
} item;
item   buffer[BUFFER_SIZE];
int    in = 0;              // Location of next input to buffer
int    out = 0;             // Location of next removal from buffer
int    counter = 0;         // Number of buffers currently full
```

Bounded-Buffer – Shared-Memory Solution

- Shared Data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer [BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Can only use BUFFER_SIZE-1 elements

Producer – Consumer

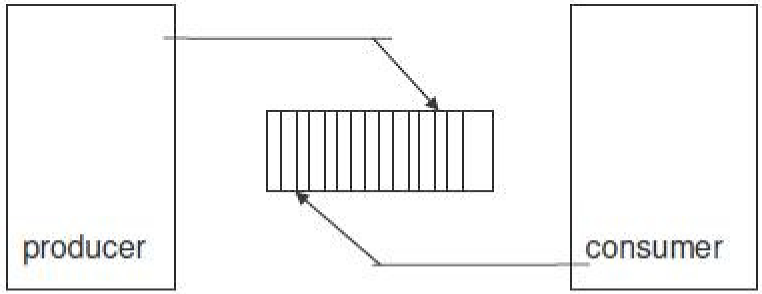
```
#define BUFFER_SIZE 10
typedef struct {
    DATA       data;
} item;
item    buffer[BUFFER_SIZE];
int     in = 0;
int     out = 0;
int      counter = 0;
```

PRODUCER

```
item   nextProduced;

while (TRUE) {
    while (counter == BUFFER_SIZE);
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

CONSUMER

```
item  nextConsumed;

while (TRUE) {
    while (counter == 0);
    nextConsumed = buffer[out];
    out = (out + 1) %
    BUFFER_SIZE;
    counter--;
}
```



Bounded-Buffer – Producer

```
while (true) {
      /* Produce an item */
       while (((in = (in + 1) % BUFFER SIZE count)  == out)
        ;   /* do nothing -- no free buffers */
       buffer[in] = item;
       in = (in + 1) % BUFFER SIZE;
     }
```

Bounded Buffer – Consumer

```
while (true) {
        while (in == out)
              ; // do nothing -- nothing to consume

       // remove an item from the buffer
       item = buffer[out];
       out = (out + 1) % BUFFER SIZE;
   return item;
    }
```

Setting Final value of Counter

Note that **counter++;** ← **this line is NOT what it seems!!**

is really --> register = counter
register = register + 1
counter = register

At a micro level, the following scenario could occur using this code:

TO;	**Producer**	**Execute**	**register1 = counter**	**register1 = 5**
T1;	**Producer**	**Execute**	**register1 = register1 + 1**	**register1 = 6**
T2;	**Consumer**	**Execute**	**register2 = counter**	**register2 = 5**
T3;	**Consumer**	**Execute**	**register2 = register2 - 1**	**register2 = 4**
T4;	**Producer**	**Execute**	**counter = register1**	**counter = 6**
T5;	**Consumer**	**Execute**	**counter = register2**	**counter = 4**

Buffer Types

- Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process
  - Unbounded-buffer places no practical limit on the size of the buffer
  - Bounded-buffer assumes that there is a fixed buffer size

RC & CS

- Race Condition – Where several processes access and manipulate the same data concurrently and the *outcome of the execution depends on the particular order* in which access takes place.

- Critical Section – Segment of code in which Process may be changing common variables, updating a table, writing a file and so on.

Peterson's Solution

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

Peterson's Solution

```
Var         flag : array [0…1] of Boolean;
            Turn : 0..1;
Begin
            Flag[0] = false;
            Flag[1] = false;
Parbegin
      Repeat
            Flag[0] = true
            Turn = 1
            While flag[1] && turn==1
               Do {nothing};
            {Critical Section}
            Flag[0] = false;
            {Remainder}
      Forver;

      Repeat
            Flag[1] = true
            Turn = 0
            While flag[0] && turn==0
               Do {nothing};
            {Critical Section}
            Flag[1] = false;
            {Remainder}
                  Forver;
Parend
end
```

Peterson's Solution

```
flag[0]   = 0;
flag[1]   = 0;
turn;
```

```
P0: flag[0] = 1;
    turn = 1;
    while (flag[1] == 1 && turn == 1)
    {
           // busy wait
    }
    // critical section
       ...
    // end of critical section
    flag[0] = 0;
```

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

Synchronization Hardware

- Many Systems provide hardware support for critical section code
- Uni-Processors – Could disable Interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
- Modern machines provide special atomic hardware instructions
    - Atomic :- Uninterruptible
  - Either Test memory word and Set value
  - Or Swap contents of two memory words

TestAndSet Instruction

Definition:

```
boolean TestAndSet (boolean *target)
{
     boolean rv = *target;
     *target = TRUE;
     return rv:
}
```

Solution using TestAndSet

- Shared Boolean variable Lock, Initialized to FALSE.

  Solution:

```
do {
   while ( TestAndSet (&lock ))
            ;   /* do nothing

      //    critical section

   lock = FALSE;

      //     remainder section

 } while ( TRUE);
```

Swap Instruction

Definition:

```
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp:
}
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE, Each process has a local Boolean variable key.

Solution:

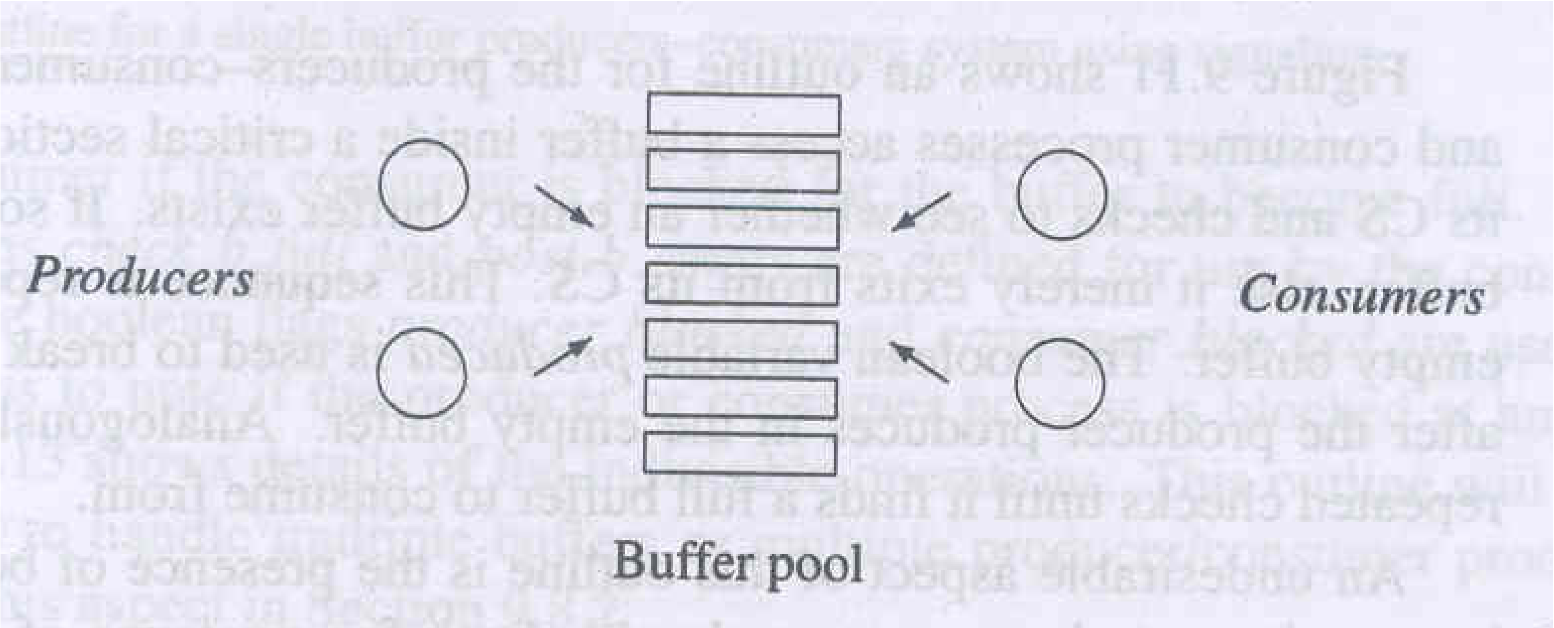
```
do {
      key = TRUE;
       while ( key == TRUE)
            Swap (&lock, &key );

              //    critical section

       lock = FALSE;

            //      remainder section

     } while ( TRUE);
```

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – Integer Variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated

- Can only be accessed via two indivisible (atomic) operations
  - ```
    wait (S) {
         while S <= 0
              ; // no-op
           S--;
     }
    ```
  - ```
    signal (S) {
      S++;
    }
    ```

Producer Consumer



Solution to PC must satisfy 3 conditions

1. A producer must not overwrite a full buffer.
2. A consumer must not consume an empty buffer.
3. Producers and consumers must access buffers in a mutually exclusive manner.

Solution to PC with Busy Wait

```
begin
Parbegin
    var produced : boolean;              var consumed : boolean;
    repeat                               repeat
        produced := false;                   consumed := false ;
        while produced = false               while consumed = false
            if an empty buffer exists            if a full buffer exists
            then                                 then
                { Produce in a buffer }              { Consume a buffer }
                produced := true;                    consumed := true;
        { Remainder of                       { Remainder of
          the cycle }                          the cycle }
    forever;                             forever;
Parend
end.
            Producer                             Consumer
```

Solution to PC with Signaling

```
var
    buffer : ... ;
    buffer_full : boolean;
    producer_blocked, consumer_blocked : boolean;
begin
    buffer_full := false;
    producer_blocked := false;
    consumer_blocked := false;
Parbegin
  repeat                                  repeat
     check_b_empty;                          check_b_full;
     { Produce in the buffer }               { Consume from the buffer }
     post_b_full;                            post_b_empty;
     { Remainder of the cycle }              { Remainder of the cycle }
  forever;                                forever;
Parend;
end.
           Producer                              Consumer
```

Indivisible Operations for PC

```
procedure check_b_empty
begin
    if buffer_full = true
    then
        producer_blocked := true;
        block (producer);
end;

procedure post_b_full
begin
    buffer_full := true;
    if consumer_blocked = true
    then
        consumer_blocked := false;
        activate (consumer);
end;
```

Operations of producer

```
procedure check_b_full
begin
    if buffer_full = false
    then
        counsumer_blocked := true;
        block (consumer);
end;

procedure post_b_empty
begin
    buffer_full := false;
    if producer_blocked = true
    then
        producer_blocked := false;
        activate (producer);
end;
```

Operations of consumer

Reference List

Operating Systems Concepts By Silberschatz & Galvin,

Operating systems By D M Dhamdhere,

System Programming By John J Donovan,

www.os-book.com

www.cs.jhu.edu/~yairamir/cs418/os2/sld001.htm

http://gaia.ecs.csus.edu/~zhangd/oscal/pscheduling.html

http://www.edugrid.ac.in/iiitmk/os/os_module03.htm

http://williamstallings.com/OS/Animations.html

etc…

Thnx…