Disclaimer

This is NOT A COPYRIGHT MATERIAL

*Content has been taken mainly from the following books:*

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

```c
#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;

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

item nextConsumed;
CONSUMER
while (TRUE) {
    while (counter == 0);
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```
while (true) {
    /* Produce an item */
    while ( ((in = (in + 1) % BUFFER SIZE) == out) 
        ; /* do nothing -- no free buffers */ 
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}
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:

<table>
<thead>
<tr>
<th>TO;</th>
<th>Producer</th>
<th>Execute</th>
<th>register1 = counter</th>
<th>register1 = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1;</td>
<td>Producer</td>
<td>Execute</td>
<td>register1 = register1 + 1</td>
<td>register1 = 6</td>
</tr>
<tr>
<td>T2;</td>
<td>Consumer</td>
<td>Execute</td>
<td>register2 = counter</td>
<td>register2 = 5</td>
</tr>
<tr>
<td>T3;</td>
<td>Consumer</td>
<td>Execute</td>
<td>register2 = register2 - 1</td>
<td>register2 = 4</td>
</tr>
<tr>
<td>T4;</td>
<td>Producer</td>
<td>Execute</td>
<td>counter = register1</td>
<td>counter = 6</td>
</tr>
<tr>
<td>T5;</td>
<td>Consumer</td>
<td>Execute</td>
<td>counter = register2</td>
<td>counter = 4</td>
</tr>
</tbody>
</table>
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

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

Parenend

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;
```
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
Definition:

```c
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:

```c
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:

```c
    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
  - $\text{wait}(S)$ {
    - while $S <= 0$
    - ; // no-op
    - $S--$;
  }
  - $\text{signal}(S)$ {
    - $S++$;
  }
Producer Consumer

Producers

Buffer pool

Consumers
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

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

Parend

begin
Parbegin
var consumed : boolean;
repeat
consumed := false;
while consumed = false
if a full buffer exists
    { Consume a buffer }
    consumed := true;
{ Remainder of the cycle }
forever;
end.
Consumer
```
Solution to PC with Signaling

```plaintext
var
buffer : ... ;
buffer_full : boolean;
producer_blocked, consumer_blocked : boolean;

begin
buffer_full := false;
producer_blocked := false;
consumer_blocked := false;

Parbegin
repeat
  check_b_empty;
  { Produce in the buffer }
  post_b_full;
  { Remainder of the cycle }
  forever;
end;

Producer

Consumer
```

Parbegin
repeat
  check_b_full;
  { Consume from the buffer }
  post_b_empty;
  { Remainder of the cycle }
  forever;
end.
Indivisible Operations for PC

**Operations of producer**

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

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

**Operations of consumer**

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

procedure post_b_empty
begin
  buffer_full := false;
  if producer_blocked = true then
    producer_blocked := false;
    activate (producer);
  end;
end;
```
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/iiitmк/os/os_module03.htm
http://williamstallings.com/OS/Animations.html

etc…
Thnx...