BITS, PILANI – K. K. BIRLA GOA CAMPUS

Operating Systems

by

Dr. Shubhangi

Synchronization

BACKGROUND

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the

consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

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
}
```

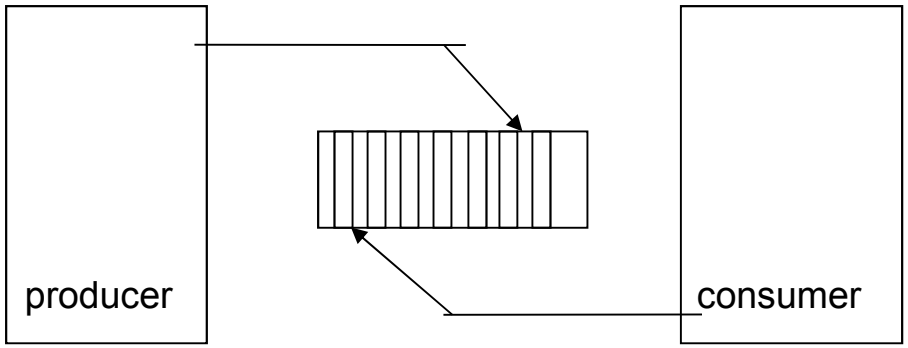
PROCESS SYNCHRONIZATION

A **producer** process "produces" information "consumed" by a **consumer** process.

```
item   nextProduced;

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

PRODUCER



The Producer Consumer Problem

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

```
item  nextConsumed;

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

CONSUMER

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
- Consider this execution interleaving with “count = 5” initially:
  - S0: producer execute register1 = count {register1 = 5}
  - S1: producer execute register1 = register1 + 1 {register1 = 6}
  - S2: consumer execute register2 = count {register2 = 5}
  - S3: consumer execute register2 = register2 - 1 {register2 = 4}
  - S4: producer execute count = register1 {count = 6 }
  - S5: consumer execute count = register2 {count = 4}

PROCESS SYNCHRONIZATION

Critical Sections

A section of code, common to n cooperating processes, in which the processes may be accessing common variables.

A Critical Section Environment contains:

Entry Section	Code requesting entry into the critical section.
Critical Section	Code in which only one process can execute at any one time.
Exit Section	The end of the critical section, releasing or allowing others in.
Remainder Section	Rest of the code AFTER the critical section.

PROCESS SYNCHRONIZATION

Critical Sections

The critical section must ENFORCE ALL THREE of the following rules:

Mutual Exclusion: No more than one process can execute in its critical section at one time.

Progress: If no one is in the critical section and someone wants in, then those processes not in their remainder section must be able to decide in a finite time who should go in.

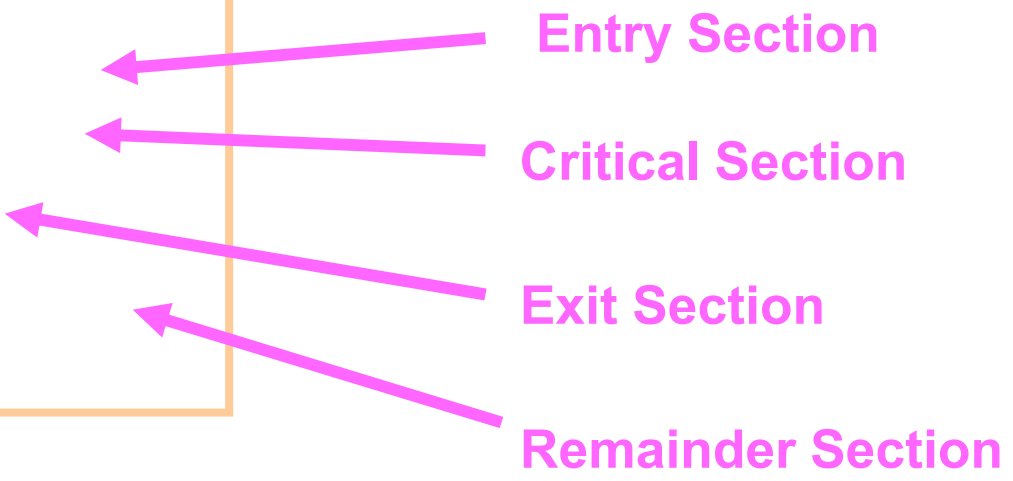
Bounded Wait: All requesters must eventually be let into the critical section.

PROCESS SYNCHRONIZATION

Two Processes Software

Here's an example of a simple piece of code containing the components required in a critical section.

```
do {
while  ( turn  ^=  i );
/* critical section  */
turn  =  j;
/* remainder section */
 } while(TRUE);
```



PROCESS SYNCHRONIZATION

Two Processes Software

Here we try a succession of increasingly complicated solutions to the problem of creating valid entry sections.

NOTE: In all examples, **i** is the current process, **j** the "other" process. In these examples, envision the same code running on two processors at the same time.

TOGGLED ACCESS:

```
do {
while  ( turn  ^=  i );
/* critical section  */
turn  =  j;
/* remainder section */
 } while(TRUE);
```

Algorithm 1

Are the three Critical Section Requirements Met?

PROCESS SYNCHRONIZATION

Two Processes Software

FLAG FOR EACH PROCESS GIVES STATE:

Each process maintains a flag indicating that it wants to get into the critical section. It checks the flag of the other process and doesn't enter the critical section if that other process wants to get in.

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

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

Are the three Critical Section Requirements Met?

PROCESS SYNCHRONIZATION

Two Processes Software

FLAG TO REQUEST ENTRY:

- Each processes sets a flag to request entry. Then each process toggles a bit to allow the other in first.
- This code is executed for each process i.

Algorithm 3

Shared variables

☞**boolean flag[2]**;
initially **flag [0] = flag [1] = false.**

☞**flag [i] = true** $\Rightarrow$ P_i ready to enter its critical section

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

Are the three Critical Section Requirements Met?

This is Peterson's Solution

PROCESS SYNCHRONIZATION

Critical Sections

The hardware required to support critical sections must have (minimally):

- Indivisible instructions (what are they?)
- Atomic load, store, test instruction. For instance, if a store and test occur simultaneously, the test gets EITHER the old or the new, but not some combination.
- Two atomic instructions, if executed simultaneously, behave as if executed sequentially.

PROCESS SYNCHRONIZATION

Hardware Solutions

Disabling Interrupts: Works for the Uni Processor case only. WHY?

Atomic test and set: Returns parameter and sets parameter to true atomically.

```
while ( test_and_set ( lock ) );
/* critical section */
lock = false;
```

Example of Assembler code:

```
GET_LOCK:   IF_CLEAR_THEN_SET_BIT_AND_SKIP <bit_address>
            BRANCH   GET_LOCK                          /* set failed */
                     -------                           /* set succeeded */
```

Must be careful if these approaches are to satisfy a bounded wait condition - must use round robin - requires code built around the lock instructions.