BITS, PILANI – K. K. BIRLA GOA CAMPUS

Operating Systems

by

Dr. Shubhangi

1. if multiple threads: multiple writers → may lead to incoherency.

2. synchro: if they are executed parallely: should give the same result / output if they were run sequentially.

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

buffer condition.

```
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

The Producer Consumer Problem

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

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

PRODUCER

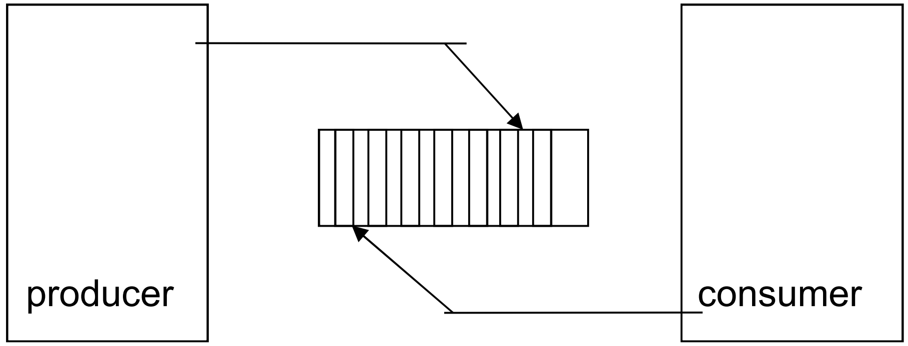
```
item   nextProduced;

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--;
}
```



Race Condition

critical section:- statement that uses common variable.

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

→ even though at a high programming language level it may seem like a single instruction, at the assembly language it may be implemented as multiple instructions.

indivisible / Atomic → either do it completely or don't do it at all.

- 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}

Registers (Internal / Hardware)

producer thread

consumer thread.

expected value should've been 5 since (5+1-1)

This happened because of pre-emption and this is referred to as "RACE CONDITION"

Solⁿ : Inside the critical section only one thread should be present at a time.
"mutual exclusion"
↳ mutually they decide amongst themselves : exclusive access critical section
∴ execution is made serial only for critical section, else it's parallel & multitasking.

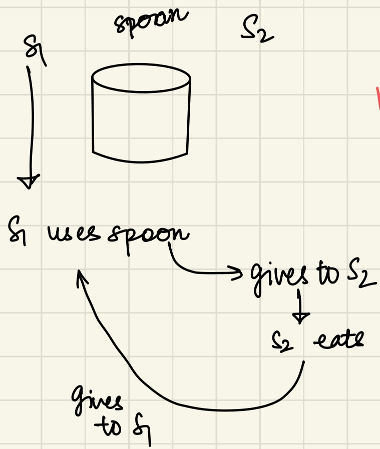
can we disable interrupts as a solⁿ? → event → interrupt → goes to processor

∴ for a uniprocessor system, yes. can be done easily

for a multiprocessor system, should we send interrupt to all processors? NO cause inefficient, leads to overhead.

↳ can we disable all interrupts to all processors

∴ disabling interrupts not an option for multiprocessor system.



but what if S_2 does not give back spoon?

S_1 wants to enter but can't enter

∴ we need Progress.

"strict alternation"

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.

if threads only reading, not an issue, if write then issue.

this code will be the same for both threads.

Critical Section — Code in which only one process can execute at any one time.

∴ critical section only when write is involved.

Exit Section — The end of the critical section, releasing or allowing others in.

Remainder Section — Rest of the code AFTER the critical section.

global/shared variable is not used.

PROCESS SYNCHRONIZATION

Critical Sections

Any solution must satisfy three conditions

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.

(check entry condition)

T_1 T_2 T_3

$\boxed{i{+}{+};}$ i++; i++; critical section

↳ if this is executing, then T_2, T_3 should not be execute i++.

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.

(check exit condition)

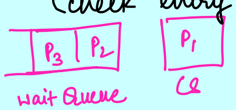
↳ Not waste time to decide., if a thread is in remainder section, it should not block another thread to entering critical section,

∴ "cannot keep the washroom locked when not in use"

"key to lock must be transferred to the other thread"

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

(check entry condition)



CS1. Waiting Queue

↳ number of critical sections: That many number of wait queues.

when P_1 is inside CS, P_2 P_3 P_4 are waiting,

then P_2 P_3 P_4 must wait for a bounded time.

should not happen that the same process gets into the critical section again.

PROCESS SYNCHRONIZATION

Two Processes Software

⓪ ∴ a single variable may be able to give mutual exclusion but cannot give progress as well.

① Turn as a temporary variable is not possible b/w more than 2 processes.

② we do not know priority. ∴ cannot maintain priority

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);
```

Entry Section

Critical Section

Exit Section

Remainder Section

for P_j

while (turn != j);

do

turn = i;

while (true);

is this mutually exclusive?

to check for mutual exclusion,

Assume P_i is in CS,

$\boxed{P_i}$

CS

$P_j \rightarrow$ can P_j access CS?

P_j can only access CS if while(turn != j); becomes false

∴ when turn == j

but when turn == j, P_i has entered into remainder section

∴ mutually exclusive.

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?

CHECKING FOR MUTUAL EXCLUSIVITY

mutual exclusion will not be there when both for P_i & P_j, critical section is entered.
only possible when while is broken for both.

to break while,
turn == i && turn == j
this is not possible, ∴ MUTUALLY EXCLUSIVE

CHECKING FOR PROGRESS

if P_i is in CS, ∴ turn = i

P_i → empties CS (turn = j)
CS

if P_j executes, then
it will make
turn = i
but what if P_i wants to
execute before P_j executes? → cannot.
∴ Progress is not achieved.

CHECKING FOR BOUNDED WAIT:-

- P_i ← P_j Busy waiting (BW)
  CS

Pi comes out but its timer has not yet expired
and Pi again wants to enter Critical Section
⇓
Not allowed cause turn is given to j

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.** each process has its own flags.

☞**flag [i] = true** ⇒ P_i ready to enter its critical section

Algorithm 2

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

```
do {
flag[j] = true;
while(flag[i])
CS
flag[j] = false;
RS
} while (1)
```

Are the three Critical Section Requirements Met?

P_i	P_j	what enters CS:-
T	T	whichever first
T	F	P_i
F	T	P_j
F	F	PROBLEM

Checking for Mutual Exclusivity

* if mutually exclusive, both P_i and P_j are in CS, therefore while(flag[j]) for P_i and while(flag[i]) for P_j is broken

* $\boxed{T_i}$ when T_i is inside, its flag is set to true. it will only enter when flag[j] = false
  ∴ this implies P_j cannot enter.

∴ MUTUALLY EXCLUSIVE

* this also has the possibility of a deadlock

starts

P_i = false P_j = false.

⇒ Both are waiting for each other to make the flag false.

Checking for Progress.

* is the exit condition giving a chance to other processes.

* if P_i executes and leaves it makes flag[i] = false, and flag[j] is false by default. ∴ P_i is not blocked from execution.

∴ No process is blocked from entering the critical section

∴ SATISFIES PROGRESS

Checking for Bound wait:

* if P_i. then P_j is waiting, it should never happen that P_j is always waiting.

* depends on CPU cycles, and when pre-emption has occured.

say $\boxed{P_i}$ ⟶ exits and sets its flag to false
CS

and is in the remainder section + has CPU cycles, meanwhile, P_i enters again, sets its flag to TRUE, and may again enter into critical section

loops back

∴ if both are false, any can enter

∴ can just be P_i P_i . . . . infinite times

∴ NOT BOUNDED WAIT.

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.

* was good but for 2 process systems only.

* software solution

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

Are the three Critical Section Requirements Met?

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

initially itself gives the turn to j

BUSY WAITING CONDITION

P_j is willing to enter CS and it is its turn } then P_i is not allowed to enter.

This is Peterson's Solution

combination of both.

checking for mutual exclusion

- both will be executing if P_i and P_j can both enter CS.

Assume that flag[i] = flag[j] = false
and P_i enters

∴ flag[i] = true and turn = j
flag[j] is still false.
∴ (flag[j] && turn == j) : false
$\underbrace{\text{flag[j]}}_{\text{false}}$ && $\underbrace{\text{turn == j}}_{\text{true}}$

∴ P_i executes.

If in b/w P_j wants to execute while P_i is in CS,
flag[j] = true,
turn = i

while ($\underbrace{\text{flag[i]}}_{\text{TRUE}}$ && $\underbrace{\text{turn == i}}_{\text{TRUE}}$)

cannot enter until flag[i] = false

P_i reaches remainder; flag[i] = false.

Now P_j can enter CS since
flag[i] = false.

∴ MUTUALLY EXCLUSIVE

Progress :-

even if P_i does not want to enter after P_j, it is still possible because
P_i will never set its flag to TRUE

∴ PROGRESS ACHIEVED

BOUNDED WAIT

It will never happen that P_j is waiting after P_i and P_i re-enters. This is ensured by "strict alteration"

∴ BOUNDED WAIT ACHIEVED

⇒ CHECKING FOR DEADLOCK.

strict Alteration ∴ no process is holding chance.

∴ no possibility for deadlock

QUESTION:
is any process waiting for the other process to give away the turn?

- this alteration would also be independant of CPU cycles.

PROCESS SYNCHRONIZATION

Critical Sections

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

- Indivisible instructions (what are they?)

Hardware

∴ queue forms for the key instead of the washroom itself.

software

- 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.

∴ location of queue matters.

in software : • A special node is made in the code itself
• boolean value of lock decides whether in critical section or not.

Here : global variable;

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.

test the value of the lock and return the

testing

if lock= F
return F

if lock =T
return T

```
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 */
```

if lock= f, set lock=T

if lock= T, let it remain T.

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

TSL $\boxed{\text{return lock} \\ \text{set lock = T}}$

```
boolean TSL( &target)
{
  boolean rv = target
   set lock = true
  return rv;
}
```

target → ~~F~~ → T
lock   lock.

passed by reference.

while (TSL(&lock))

F → rv is returned
rv

∴ while (F)
∴ can enter critical section

if P_1 → P_0

If P_1 tries to enter b4 P_2 leaves,

TSL(&target) → True

∴ rv = true
lock = true
return True

∴ while (TSL(&lock)) never breaks

P_0

P_0 exits and makes lock = false.

∴ **MUTUALLY EXCLUSIVE**

CHECKING PROGRESS

P_0 wants to re-enter,

P_1 block it? No

∴ **Progress is achieved**.

CHECKING BOUNDED WAIT

but it may happen that P_0 re-enters always and P_1 waits indefinitely

∴ **BOUNDED WAIT NOT ACHIEVED**.

Assume P_0 has set the lock to true, and has entered critical section,

T lock P_0 ← P_1

P_1 → checks the status of the lock ∴ has to BUSY WAIT.
while (true)

suppose CPU is with P_0 and exits critical section but still CPU is with P_0.

After exiting, it has set the value of lock to FALSE

∴ since it still has CPU cycles, it's BW condition becomes false instantly ∴ it re-enters critical section

∴ Alone lock would not work,
we need key for this.