Process Synchronization

Cooperating processes — one that can affect or be affected by other processes executing in the ~~pro~~ system.

(shared region of data — that the cooperating ~~processes can access &~~ manipulate)

can either directly share a logical address space (both code and data)

can be allowed share data only through files and messages.

Problems : 1) concurrent access to shared data may result in data inconsistency

Solution : Orderly execution of cooperating processes that share a logical address space so that data consistency is maintained.

* Shared-memory system — Producer-Consumer Problem.

  * Buffer — Producer produces into and keeps in it
    Consumer consumes the inf

* A producer process produce into
  └ that is consumed by a consumer process

One solution to producer-consumer problem uses shared memory.

* To allow producer-consumer process to run we must have a buffer of items that can be filled by the producer and emptied by the consumer.

Buffer will reside in the region of memory that is shared by the producer and consumer process.

* Producer can produce one item
Consume consume another item

Producer and consumer must be synchronized so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers (i) Bounded buffer
- Fixed size
- Consumer must wait if the buffer is empty
- Producer must wait if the buffer is full.

(ii) Unbounded buffer
- No limit on the size of the buffer.
- Consumer may have to wait for new item.
- Producer can always produce new item.

Bounded buffer -

Need to keep track of the items present

lets say, variable - that is going to count no of items present in a buffer.

when produces produces - variable increased

consumer consumes - " decreased

counter variable = 0 — initialized to 0.

variable-name

Add new item to buffer - counter ++

Remove " " " - counter --

Execute seperately - Producer & Consumer

table variable-value initial value - ৪৮ সমান হবে।

Kintu concurrently hole কী হবে?

counter ++ → $register_1$ = counter [value stored in $register_1$]

$register_1$ = $register_1$ + 1 [value incremented]

counter = $register_1$ [Assigned to counter]

counter -- → $register_2$ = counter

$register_2$ = $register_2$ - 1

counter = $register_2$

counter current value = 5

T_0	Producer	execute	reg_1 =	counter	{reg_1 = 5}
T_1	"	"	reg_1 =	reg_1 + 1	{reg_1 = 6}
T_2	consumer	"	reg_2 =	counter	{reg_2 = 5}
T_3	consumer	"	reg_2 =	reg_2 - 1	{reg_2 = 4}
T_4	producer	"	counter =	reg_1 "	{counter = 6}
T_5	consumer	"	" =	reg_2	{ " = 4

Gets hold of the counter first

The process which assigns the value to the counter first : ans.

Incorrect state

— Because we allowed both process to manipulate the "counter" variable concurrently.

A situation like this,

where several processes access and manipulate the same data concurrently,

& outcome of the execution depends on the particular order in which the access takes place is called race condition.

The Critical Selection Problem

Consider, a system of n process $\{P_0, P_1 \cdots P_n\}$

Each process — has a segment of code.

called critical section

Data Manipulation

(I) process may be changing common variables

(II) updating a file.

(III) writing a file & so on.

When a process is executing in its critical section, no other process can execute in its critical section.

No two processes are executing in their critical section at the same time

one process executing in their critical section
└ Making some change to the shared data

So no other process should be allowed to execute at that time.

Critical - section problem
└ To design a protocol that processes can use to cooperate

* Each process must request permission to enter its critical section
* Section of code implementing this req - entry section
* Critical code - may be followed by an exit section.
* Remaining code - remainder section.

```
do {
    entry section
    critical section
    exit section
    remaindersection
} while (TRUE);
```

A solution to the CRITICALSECTION PROBLEM must satisfy the following 3 requirements.

(1) Mutual exclusion

If Process P_I is executing in its critical section, no other processes can be executing in their critical section.

(2) Progress :

If no process is executing in their critical section — and some processes wants to enter their critical section, then only the processes which are NOT EXECUTING IN THEIR REMAINDER SECTION can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

(3) Bounded waiting:

Will exist a limit on the no of times a process is allowed to enter its critical section after another process has made a req to enter its critical section and it is granted.

PETERSON'S SOLUTION

* Software based solution
* May not work on morder computer architectures.
* Provides a good algorithmic description of solving the critical-section problem & illustrates some of the complexities involved in designing software that address the requirements of critical section problem.

→ Restricted to two process
→ Alternates execution between critical section and remainder section.

P_i P_j

→ Two data items to be share between the two processes.

- turn — int turn;
  - Indicates whose turn is it to enter its critical section.
- flag - boolean flag[2]
  - Used to indicate if a process is ready to enter its critical section.

flag[i] = TRUE
- means P_i is ready to enter its critical section.

* When a process wants to ~~int~~ enter critical section
  flag[P_i] = TRUE
  set the flag = TRUE

P_i wants to enter its critical section,
turn = P_j

(humble algorithm)

Structure of P_i

As long as while condition is true

```
do{
    flag[i] = true;
    turn = j;
    while(flag[j] && turn==[j]);
        critical section.
    flag[i] = false;
    remainder section

} while(TRUE);
```

P_j same.

turn = i;

while (flag[i] && turn==[i]);

turn = j set hoy (execute করবে)

turn = i set hoy

Now, flag[j] && turn==[j] false hoye jay

So P[i] critical section e dhuke.

আর, P_j er khetre flag[i] && turn==[i] true থাকে, তাহলে loop চলতে থাকবে।

So while P_i is executing (critical section) P_j is not.

So after P_i completes execution flag[i] = false

তখন P_j - এর 'while' condition false হয়ে যায়,
তখন P_j executes its critical section.

* In this way Peterson solves the critical section problem.

Test and Set Lock (Hardware based)

* shared lock variable (0, 1) — সব Always value
  - 0 → Unlocked
  - 1 → Locked

* Before entering the critical section, a process inquires about the lock (0 or 1)

* If it is locked, it keeps on waiting till it becomes free.

* Not locked, takes the lock, sets to locked and executes the critical section.

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

shared lock variable.

Atomic (Single Operation that remains uninterrupted)

Process P1

```
do {
while (TestAndSet(&lock));
critical section
lock = False
//remainder section
} while (True);
```

variable

returned 0

break and execute critical section.

target set to True

lock is set to 1

After execution.

lock is again set to False

Process 2

return 1

loop will be continued.

After P1 completes execution

lock = FALSE

lock = 0

target = 0

rv = 0

0 get returned.

while condition False hoy

break and execute critical section

i) satisfy Mutual-exclusion
ii) Doesn't satisfy bounded waiting
- P1 executing
- P2 tries and gets preempted
- P3 arrives, P1 completes executing
- P3 requests
- P3 starts executing
- P2 arrives and P3 is executing
- P2 keeps on waiting.

Semaphores

Edsger Dijkstra (SW based)

* manage concurrent processes by using simple integer value known as semaphore.

Variable - Non negative
- shared between threads
- Process synchronization in multiprocessing environment

* S = integer variable.
can be accessed by processes through - wait()
signal()

Two atomic operations.

wait() → P (to test)
signal() → V (to increment)

wait():

Needs to be accessed by wait()

```
P (Semaphore S) {
    while( S<=0 )
        ;
    S--;
}
```

signal() (When the process that was making use of the semaphore to enter critical section completes its operation and comes out of it)

```
V (semaphore S) {
    S++;
}
```

All the modifications to the integer value of semaphore in wait() and signal must be executed indivisibly.

When one process modifies semaphore value, no other process can modify the semaphore value at that time.

CS CamScanner

Types of Semaphore

Binary semaphore
- 0 or 1
- Mutex locks (Locks that provide mutual exclusion)

* $S < 0$ or $S = 0$
  - process already executing in critical section
  - So other processes has to wait (shared resources already in use)
* $S = 1$
  - free for any process to execute

Initially semaphore = 1

Counting Semaphore (i) Range over an unrestricted domain (values).

(ii) used to control access to a resource that has multiple instances.

(iii) Multiple values

$P_1, P_2, P_3 \longrightarrow R_1, R_2$ [$S = 2$]

- Resources = 2

P_1, P_2 [Ekshathe kaay korbe]

P_3 [wait korbe]

Disadvantages

* Requires busy waiting.

└ When a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.

└ Wastes CPU cycles that some other processes might use productively.

└ Spinlock because the process spins while waiting for the lock.

To overcome the busy waiting, we can modify the definition of the wait() and signal() semaphore operations.

When, Execute wait() and find semaphore value is not positive, it must wait.

Rather than engaging in busy waiting, the process can itself

Block operation — Places the process into the waiting queue of the semaphore

the state of the process is no waiting state.

Control is transferred to the CPU which selects other process for execution

Deadlocks & Starvation

Two or more process are waiting indefinitely for an event that can be caused only by one of the waiting process.

The event in question is the execution of a signal() operation. when such a state is reached, these processes are said to be deadlocked.

P1	P2
wait(S);	wait(Q);
wait(Q);	wait(S);
⋮	⋮
signal(S);	signal(Q);
signal(Q);	signal(S);

P1 wait() executes S
waits as P_2 is executing Q

P2 wait executes Q
waits as P_1 is executing S.

deadlocked processes

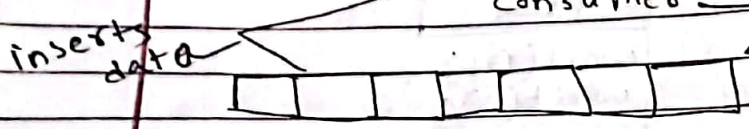
The Bounded Buffer Problem

↓

produce consumer Problem.

Problem statement: Buffer of n slots
Each slot is capable of one unit of data.

2 processes – Producer.
Consumer: remove data



i) Insert data – to empty slot
ii) Remove data – From a filled slot
iii) must not insert data – when the buffer is full
iv) must not remove data – " " " is empty.
v) should not insert and ~~do~~ remove simultaneously

Solution

3 semaphores used:

(i) m (mutex) → binary semaphore which is used to acquire & release the lock

(ii) empty whose initial value is the number of slots in the buffer, since initially all slots are empty.

(iii) A counting semaphore whose initial value is zero, full.

empty = 0 no empty slots
empty > 0 empty slot avail

Producer

```
do {
wait(empty); //wait until empty > 0
             & then decrement 'empty'   empty--
```

perform wait op on empty semaphore

```
wait(mutex); // acquire lock

/* add data to buffer */
signal(mutex); //release lock
signal(full); // increment 'full'   full++
} while(TRUE)
```

Consumer

```
do{
wait(full) // wait until full > 0 and
           then decrement 'full'

wait(mutex); // acquire lock
/* remove data from buffer */
signal(mutex); //release lock
signal(empty); // increment 'empty'
} while(TRUE)
```

The Readers Writers Problem

i) Database is to be shared among several concurrent processes.

ii) Some of these processes - (i) read the database
(ii) update (read and write) the database

Two types of processes Readers.
Writers.

* Two readers access the shared data simultaneously, no adverse effects.

* But, writer / reader or writer } Access together > chaos.

For this, we require that writers have exclusive access to the shared database.

SOLUTION

Two semaphores and an integer variable.

(1) mutex - binary semaphore initialized to 1
Ensure mutual exclusion when read-count is updated. i.e - readers enter or exit from critical section.

read-count → No of readers accessing the same file.

(2) wrt - semaphore (initialized to 1) common to both reader and writer processes.

(3) readcount Int variable (initialized to 0) that keeps track of how many processes are currently reading the object.

writer Process

do {
/* writer requests for critical section */
wait(wrt); If not being used by any other processes, then writer will get access to it.

/* performs the write — No other process reader/writer ~~can access~~ it.

// leaves the critical section
signal(wrt); — signals wrt variable. Other processes can take it.
} while(true);

Reader

```
do {
wait(mutex); // everytime readcnt is changed
                mutex is called.

readcnt ++ // No of readers increased by 1

if(readcnt ==1)(while exists 1reader)
wait(wrt) // ensures no writer can enter.
signal(mutex); // other readers can enter
                while this current reader
                is inside the critical section

/*current reader performs reading & finishes */
wait(mutex);
readcnt --; // reader wants to leave.

if (readcnt ==0) // no reader is left in the
                critical section.
signal(wrt); // writer can enter.
signal(mutex) // reader leaves
}while(true);
```

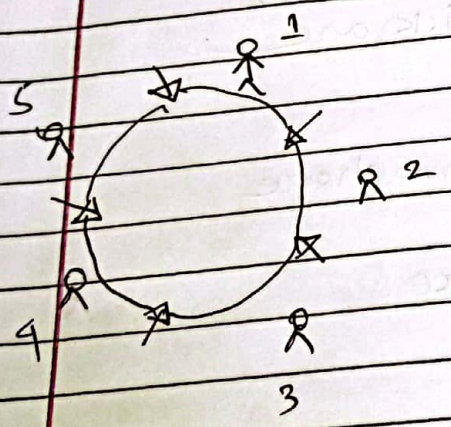
Dining Philosophers Problem

Philosopher either

Thinks | Eats

When a philosopher thinks, he does not interact with his colleagues.

When a philosopher gets hungry he tries to pick up the two forks that are closest to him. A philosopher may pick up only one fork at a time.



One cannot pick up a fork if it is already in the hand of neighbour.

Hungry philosopher has both his forks at the same time, he eats without releasing his forks.

When he has finished eating he puts down both of his forks and starts thinking again.

Solution

* Represent each fork/chopstick with a semaphore.
* Philosopher tries to grab a fork/chopstick by executing wait() on that semaphore.

He releases his fork/chopsticks by executing the signal() op. on the appropriate semaphores.

Thus the shared data are

semaphore chopstick [5];

where all the elements of chopstick are intialized to 1.

wait operation — wants the semaphore

releases the semaphore.

Thus the shared data.

semaphore chopstick [5];

where all the elements of chopstick are initialized to 1.

1	1	1	1	1

semaphore-type ~~array~~ arr

Structure of philosophers : [from 1 to 0]

elements of the chopstick = 0

```
do {
wait (chopstick [i]); one of the two chopsticks
wait (chopstick [(i+1)%5]);
....
//eat
signal (chopstick [i]);
signal ([(i+1) %5]);

//think
} while (TRUE);
```

All of the philosophers gets hungry at the same time.
All of the philosophers grabs their chopstick (left) at the same time
cannot pick up both the forks at the same time

All of the elements of the chopstick array will be equal to zero.
When each philosopher tries to grab his right chopstick he will be delayed forever.

Possible remedies :

i) Allow at most 4 philosophers to be sitting simultaneously at the table.

ii) Allow a philosopher to pickup his chopsticks only if both chopsticks are available
└ critical section এ pick করবে।

iii) Use an assymetric solution.
Odd philosopher – pick up first his left chopstick then right chopstick.
Even philosopher – picks up right chopstick and left chopstick.

Questions Δ

Stuck in the while loop.

P_1	P_2
while ($s_1 == s_2$);	while($s_1 == s_2$);
Critical Section	critical section
$s_1 = s_2$	s_2 = not(s_1);

which of the properties achieved?

Mutual exclusion: One process can execute in its critical section at one time. ✓

Progress: when a process wants to enter its critical section, a process which is executing in its remainder section should not involve in allowing/disallowing the first process from entering its critical section.

~~P1 entering again in its critical section depends on P2~~

So progress is not satisfied

✗

Questions 2

3 concurrent processes. $s0 = 1$
3 binary semaphores. $s1 = 0$
$s2 = 0$

~~Process P0~~

while(true){ the loop will continue until a semaphore is free
wait(s0); — checking the semaphore
print(0);
release(s1);
release(s2); } setting the semaphores as free.
}

s1 set to 1
s2 set to 1

0 = held
1 = free

value of s0 being set to 0 from 1 so code is executing

~~Process P1~~

wait(s1); → check s1 (held or free)
release(s0);

s0 set to 1 from 0 again

$P_0 = S_0$
$P_1 = S_1$
$P_2 = S_2$

and s1 = 0 (since executing)

~~Process P2~~

~~wait(s2);~~
release(s0);

How many times will process P0 print 0?