CSE321: Operating Systems

Topic: Process Synchronization

Prepared by:
Saad Bin Sohan

BRAC University

Email: sohan.academics@gmail.com
GitHub: https://github.com/saad-bin-sohan

Process Synchronization

Background

- Process can execute concurrently or in parallel

Concurrency:

→ Multiple processes appear to run simultaneously by rapidly (context) switching execution on a single CPU core

Parallelism:

→ Multiple processes run at the same time on multiple CPU cores (requires multi-core processors)

4. Synchronization Challenges:

→ when processes share memory/files, improper handling can cause data corruption/unpredictable program behavior.

eg: Reading and writing the same data without coordination

Race Condition

A race condition occurs when:

→ multiple processes manipulate the same shared resources (eg: a variable) concurrently

→ the final result depends on the sequence of process execution which makes it unpredictable

Solution of race problem:

i) Mutual Exclusion:

only one process can access the critical section at a time

ii) Synchronization: coordinate processes to ensure proper execution order

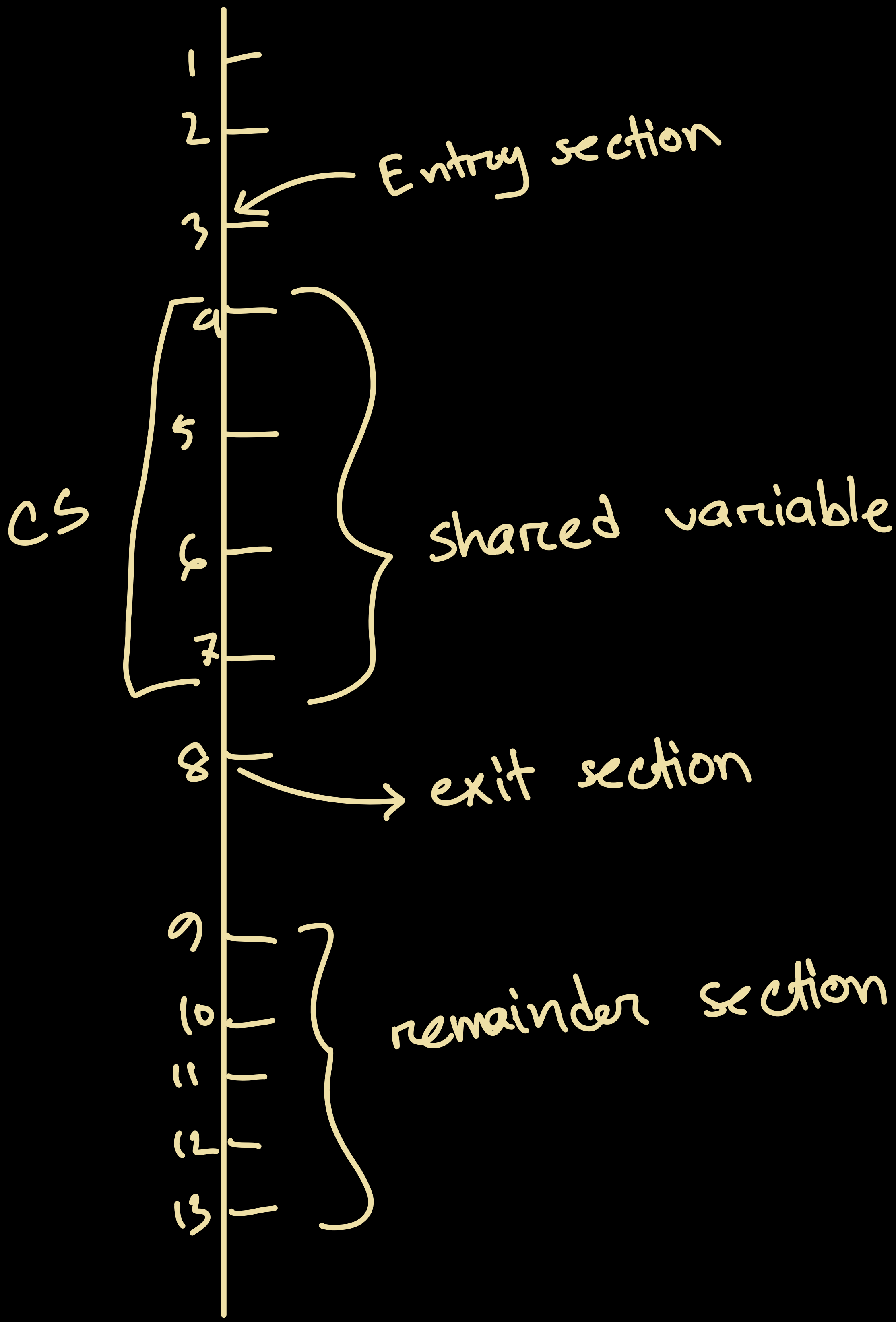
Critical Section

Critical Section: Part of a program where the processes access shared variables

→ the CS portion must execute atomically

$\underbrace{\text{atomically}}$

the entire sequence of instructions within that CS portion must be executed as a single, uninterrupted section


1
2
Entry section
3
4
5
CS
6
shared variable
7
8
exit section
9
10
remainder section
11
12
13

Structure of a process:

i) Entry section

ii) Critical section (CS)

iii) Exit Section: code to release execution of critical section.

iv) Remainder Section (RS): Critical section এর পরের code গুলো

Critical Section problem arises when:

→ multiple processes share resources

→ these processes need to coordinate their access to ensure correctness

Requirements for a solution:

i) Mutual Exclusion: only one process

can execute it's CS at a time

ii) Progress:

if no process is in its CS and there are processes waiting to enter, one of the waiting processes must eventually be allowed to enter

iii) Bounded waiting:

limitation

on how many times other processes

can enter their CS after a process has requested to enter its own (prevents starvation)

solutions to the CS problem

i) Preemptive vs non-pre-emptive kernels:

↓ process will be preempted

↓ process cannot be preempted

ii) Peterson's Solution (software based solution)

iii) Hardware-based solution

→ Test and set instruction

→ Compare and swap instruction

iv) Mutex locks

v) Semaphores

Peterson's solution for Critical Section Problem

→ software (code) based solution

→ restricted to two processes that share a critical section

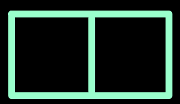
→ it uses two shared variables ('flag', 'turn') to achieve mutual exclusion, progress, and bounded waiting

turn: indicates which process's turn is it to enter the critical section

int number

flag: an array, where flag[i] element indicates if process P_i is ready and waiting to enter the CS.

array) ↙
boolean value (T/F)



2 length array

→ when P_1 works on its CS, P_2 can't be allowed to access its CS

→ humble algorithm

i, j process এর নিজস্ব variable (যেটা থাকবে value)

i → process এর identity

Algorithm →

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

turn = ~~0~~ 1

flag = [~~F~~ T , ~~F~~ ~~T~~ F]

P_0 | P_1

$i=0$, $j=1$ | $i=1$, $j=0$

$f[1]=T$

$t=0$

while F, T

$f[0]=T$

$t=1$

while T, T

cs1

cs2

cs3

stuck in while loop

cs4

$f[1]=F$

R1

while F, T

cs 1

cs 2

R2

cs3

cs4

f[0] = F

RS 1

RS 2

Hardware Based Solution for CS Problem

→ For 2 processes, we will create a shared memory location that both the processes will have access.

→ basis is "locking"

lock = memory location

→ "lock" denotes, if a process is accessing its critical section, then it is locked.

If neither process is accessing their CS, then its unlocked.

→ 2 different ways of locking:

1) test_and_set()

2) compare_and_swap()

test_and_set()

→ পুরো test_and_set() function টা এবং do while পুরো code টা atomically execute হতে (whole func is treated as a single unit and executed entirely)

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

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

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Algorithm→

```
boolean test_and_set(boolean *target){

    boolean *rv= *target
    *target= true
     return rv;

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

Compare_and_Swap()

→ compares to see if the CS is free or locked

→ locks it if free and runs own CS, then frees it.

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

```c
int compare_and_swap(int *value, int
                expected, int new-value) {

    int temp = *value;

    if (*value == expected)

        *value = new_value;

    return temp;
}
```

```c
do{

  while (compare_and_swap(&lock, 0, 1) != 0);

    // critical section

    lock = 0

    // remainder section

  } while (true);
```

Mutex Locks

→ has 2 functions:

i) acquire ( )

ii) release( )

<u>algorithm</u>:

```
acquire()
  while (!available);
  // CS
  available = false
  }

release() {
     available = true;
    }
```

Semaphore

→ Semaphose S is an integer variable which is shared with all the processes

→ accessed only through 2 atomic functions:

i) wait() → semaphore এর decrement by 1

ii) signal() → semaphore এর increment by 1

→ when one process modifies S, no other process can modify that S

```
do {
    wait(s);

    // CS

    Signal(s);

    // RS

}
```

```
wait(s){
    while (s<=0);
    s--;
}
```

```
signal (s){
    s++
}
```

2 types of semaphores:

i) Binary semaphore:

→ allows only one process to access CS

→ initial $s = 1$ $(0 \leq s \leq 1)$

→ similar to Mutex

ii) Counting Semaphores:

→ value of S can have a finite range

→ allows a finite number of processes to run their CS at a time

→ that number = number of

resources

i.e: a RAM R_1 with 3 resources (cores?) an 6 processes. how do you imply semaphore?

$\Rightarrow$ $S = 3$

lets say P_1 wait(S) $S = 2$

P_2 wait (S) $S = 1$

P_5 wait(S) $S = 0$

P_3 wait (S) → semaphore এর access পাবে না until one of processes release it

P_2 signal (S) $S = 1$

now P_3 can run its CS

⋮

and so on