Consider the following set of processes, with the length of the CPU burst given in milliseconds

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

The processes are assumed to have arrived in the order Pi, P2, P3, P4, P5, all at time 0. Write a C program to calculate the turnaround time of each process by incorporating SJF scheduling.

DESCRIPTION

If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her. However if some person is reading the file, then others may read it at the same time. Precisely in OS we call this situation as the readers-writers problem

Problem parameters:

One set of data is shared among a number of processes Once a writer is ready, it performs its write. Only one writer may write at a time If a process is writing, no other process can read it If at least one reader is reading, no other process can write Readers may not write and only read Solution when Reader has the Priority over Writer

Here priority means, no reader should wait if the share is currently opened for reading.

Three variables are used: mutex, wrt, readcnt to implement solution

semaphore mutex, wrt; // semaphore mutex is used to ensure mutual exclusion when readcnt is updated i.e. when any reader enters or exit from the critical section and semaphore wrt is used by both readers and writers int readcnt; // readcnt tells the number of processes performing read in the critical section, initially 0 Functions for sempahore :

– wait() : decrements the semaphore value.

– signal() : increments the semaphore value.

Writer process:

Writer requests the entry to critical section. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting. It exits the critical section.

Reader process:

Reader requests the entry to critical section. If allowed: it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the wrt semaphore to restrict the entry of writers if any reader is inside. It then, signals mutex as any

other reader is allowed to enter while others are already reading. After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “wrt” as now, writer can enter the critical section. If not allowed, it keeps on waiting. Thus, the semaphore ‘wrt‘ is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

Article contributed by Ekta Goel. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Boundary conditions:

Process rarely know in advance how many resources they will need.

The number of processes change as time progresses.

Resources once available can disappear.

The available assumes processes will return their resources within a reasonable time.

Processes may only get their resources after an arbitrarily long delay.

METHODOLOGY

```
#include <iostream>

#include <condition_variable>

#include<stdio.h>

#include <pthread.h>

#include <random.h>

#include <mutex.h>


{

int readers_count = 0;


int counter = 0;
```

```
const int X = 5;

std::mutex m;

std::condition_variable reader_cond;

std::condition_variable writer_cond;


void read() {

  std::mt19937 rng;

  rng.seed(std::random_device()());

  std::uniform_int_distribution<std::mt19937::result_type> dist(0, 20);

  std::unique_lock<std::mutex> lk(m, std::defer_lock);

  for(int i = 0; i < X; i++) {

    std::this_thread::sleep_for(std::chrono::milliseconds(dist(rng)));

    lk.lock();

    if(readers_count == -1) {

      reader_cond.wait(lk, []{ return readers_count != -1; });
```

```
    }

    readers_count++;

    lk.unlock();

    std::cout << "read value: " << counter << ", number of readers: " << readers_count <<
std::endl;

    lk.lock();

    readers_count--;

    if(readers_count == 0){

      writer_cond.notify_all();

    }

    lk.unlock();

  }

}

void write() {

  std::mt19937 rng;

  rng.seed(std::random_device()());
```

```
  std::uniform_int_distribution<std::mt19937::result_type> dist(0, 20);

  std::unique_lock<std::mutex> lk(m, std::defer_lock);

  for(int i = 0; i < X; i++) {

    std::this_thread::sleep_for(std::chrono::milliseconds(dist(rng)));

    lk.lock();

    if(readers_count > 0) {

      writer_cond.wait(lk, [](){ return readers_count == 0; });

    }

    readers_count = -1;

    lk.unlock();

    counter++;

    std::cout << "written value: " << counter << ", number of readers: " << readers_count <<
std::endl;

    lk.lock();

    readers_count = 0;

    reader_cond.notify_all();
```

```
    writer_cond.notify_all();
    lk.unlock();
  }
}

int main() {
  const int NUM_READERS = 5;
  const int NUM_WRITERS = 5;
  std:: cout << std::thread::hardware_concurrency() << std::endl;
  std::vector<std::thread> threads;
  for(int i = 0; i < NUM_READERS; i++) {
    threads.push_back(std::thread{read});
  }
  for(int i = 0; i < NUM_WRITERS; i++) {
    threads.push_back(std::thread{write});
```

```
  }

  for(int i = 0; i < NUM_READERS + NUM_WRITERS; i++) {

    threads[i].join();

  }

  return 0;

}
```