Worksheet - Semaphore - Peterson’s Algorithm - FCFS

```
/* CPSC 457 (Winter 2019)
 * Week 6 - 1
 * Sina Keshvadi
 */
```

==

Exmaple 01 - Crate a simple thread

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> //Header file for sleep(). man 3 sleep for details.
#include <pthread.h>

void *myThreadFun(void *vargp)
{       sleep(1);
        printf("Hi, I am new thread! \n");
        return NULL; }

int main()
{       pthread_t thread_id;
        printf("Before Thread\n");
        pthread_create(&thread_id, NULL, myThreadFun, NULL);
        pthread_join(thread_id, NULL);
        printf("After Thread\n");
        exit(0); }
```

Compile by --> gcc -lpthread 01_thread.c -o thread | Run by --> ./thread

==

Sample semaphore- A semaphore is a variable or abstract data type that is used for controlling access, by multiple processes, to a common resource in a parallel programming or a multi user environment.

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

using namespace std;
mutex mtx;             // mutex for critical section
condition_variable cv; // condition variable for critical section
bool ready = false;         // Tell threads to run
int current = 0;            // current count

/* Prints the thread id / max number of threads */
void print_num(int num, int max) {

  unique_lock<mutex> lck(mtx);
  while(num != current || !ready)
        { cv.wait(lck); }
  current++;
  cout << "Thread: ";
  cout << num + 1 << " / " << max;
  cout << " current count is: ";
  cout << current << endl;

  /* Notify next threads to check if it is their turn */
  cv.notify_all();
}

/* Changes ready to true, and begins the threads printing */
void run(){
  unique_lock<mutex> lck(mtx);
  ready = true;
  cv.notify_all();
}

int main (){
  int threadnum = 15;
  thread threads[15];
  /* spawn threadnum threads */
  for (int id = 0; id < threadnum; id++)
    threads[id] = thread(print_num, id, threadnum);

  cout << "\nRunning " << threadnum;
  cout << " in parallel: \n" << endl;

  run(); // Allows threads to run

  /* Merge all threads to the main thread */
  for(int id = 0; id < threadnum; id++)
    threads[id].join();

  cout << "\nCompleted semaphore example!\n";
  cout << endl;

  return 0;
}
```

==

Peterson's Algorithm

Peterson's Algorithm is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication (Wikipedia).

The idea is that first a thread expresses its desire to acquire lock and sets flag[self] = 1 and then gives the other thread a chance to acquire the lock. If the thread desires to acquire the lock, then, it gets the lock and then passes the chance to the 1st thread. If it does not desire to get the lock then the while loop breaks and the 1st thread gets the chance.

```
#include <stdio.h>
#include <pthread.h>

int flag[2];
int turn;
const int MAX = 1e9;
int ans = 0;

void lock_init()
{       // Initialize lock by reseting the desire of both the threads to acquire the locks. And, giving turn to one of them.
        flag[0] = flag[1] = 0;
        turn = 0; }

// Executed before entering critical section
void lock(int self)
{
        // Set flag[self] = 1 saying you want to acquire lock
        flag[self] = 1;

        // But, first give the other thread the chance to acquire lock
        turn = 1-self;

        // Wait until the other thread looses the desire to acquire lock or it is your turn to get the lock.
        while (flag[1-self]==1 && turn==1-self) ;
}

// Executed after leaving critical section
void unlock(int self)
```

```
{
        flag[self] = 0;
}

// A Sample function run by two threads created in main()
void* func(void *s)
{
        int i = 0;
        int self = (int *)s;
        printf("Thread Entered: %d\n", self);

        lock(self);

        // Critical section (Only one thread
        // can enter here at a time)
        for (i=0; i<MAX; i++)
                ans++;

        unlock(self);
}

// Driver code
int main()
{
        // Initialized the lock then fork 2 threads
        pthread_t p1, p2;
        lock_init();

        // Create two threads (both run func)
        pthread_create(&p1, NULL, func, (void*)0);
        pthread_create(&p2, NULL, func, (void*)1);

        // Wait for the threads to end.
        pthread_join(p1, NULL);
        pthread_join(p2, NULL);

        printf("Actual Count: %d | Expected Count: %d\n", ans, MAX*2);
        return 0;
}

Compile by --> gcc -lpthread peterson.c -o peterson | Run by --> ./peterson
=============================================================================
First Come First Service Scheduling - C++ program for implementation of FCFS
scheduling

#include<iostream>
using namespace std;

// Function to find the waiting time for all processes
void findWaitingTime(int processes[], int n,
                          int bt[], int wt[])
{
    // waiting time for first process is 0
    wt[0] = 0;

    // calculating waiting time
    for (int  i = 1; i < n ; i++ )
        wt[i] =  bt[i-1] + wt[i-1] ;
}

// Function to calculate turn around time
void findTurnAroundTime( int processes[], int n,
                  int bt[], int wt[], int tat[])
{
    // calculating turnaround time by adding
    // bt[i] + wt[i]
    for (int  i = 0; i < n ; i++)
        tat[i] = bt[i] + wt[i];
}

//Function to calculate average time
void findavgTime( int processes[], int n, int bt[])
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;

    //Function to find waiting time of all processes
    findWaitingTime(processes, n, bt, wt);

    //Function to find turn around time for all processes
    findTurnAroundTime(processes, n, bt, wt, tat);

    //Display processes along with all details
    cout << "Processes  "<< " Burst time  "
         << " Waiting time  " << " Turn around time\n";

    // Calculate total waiting time and total turn
    // around time
    for (int  i=0; i<n; i++)
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        cout << "   " << i+1 << "\t\t" << bt[i] <<"\t    "
            << wt[i] <<"\t\t  " << tat[i] <<endl;
    }

    cout << "Average waiting time = "
         << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "
         << (float)total_tat / (float)n;
}

// Driver code
int main()
{
    //process id's
    int processes[] = { 1, 2, 3};
    int n = sizeof processes / sizeof processes[0];

    //Burst time of all processes
    int  burst_time[] = {24, 3, 3};

    findavgTime(processes, n,  burst_time);
    return 0;
}
=====================================================================
```