**Problem 1**

**Original solution (parent thread waits for child to finish execution)**

//compile program with command line entry: gcc p1.c -o FILENAME -lpthread -lm

//run program with ./FILENAME NUMBER

// e.g. ./a.out 5

/\*

currently running program on Ubuntu for Windows with:

gcc p1.c -lpthread -lm

./a.out 10

\*/

//NOTE: This program can only calculate up to the 1474th Fibonacci number accurately

//Iteration 1: program with only two threads total.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <math.h>

/\*

currently running program on Ubuntu for Windows with:

gcc p1.c -lpthread -lm

./a.out

\*/

//NOTE: This program can only calculate up to the 78th Fibonacci number accurately

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <math.h>

void \*fibCalc(void \*fibStruct);

struct fibArrPtr{

int n;

unsigned long \*fibArr;

};

//This function is for the spawned POSIX thread to run.

// It calculates the Fibonacci sequence accurately up to the 78th term.

// Errors are introduced in the 79th term and persist onwards

// due to the computer representation of floating point numbers.

void \*fibCalc(void \*fibStruct)

{

//initialize second thread's variables from fibArrPtr struct

struct fibArrPtr \*fibAP = (struct fibArrPtr \*) fibStruct;

int n = fibAP->n;

unsigned long \*fibArr = fibAP->fibArr;

fibArr[0] = 0.0;

fibArr[1] = 1.0;

//calculate latter half of Fibonacci numbers and put into Fibonacci array shared with parent

for(int j = 2; j < n; j++)

{

fibArr[j] = fibArr[j-1] + fibArr[j-2];

}

//printf("Second half finished\n");

return NULL;

}

int main(int argc, char \*\*argv)

{

int n;

char \*input = calloc((size\_t) 8, (size\_t) sizeof(char));

printf("Enter a positive integer: ");

fgets(input, 6, stdin);

n = atoi(input); //Convert user input to integer for use in threads

//check validity of user input

if(input[0] < '0' || input[0] > '9' || n < 0 || n > 1474)

{

printf("Error: Input is not a valid number.\n");

exit(0);

}

//got and validated n for number of Fibonacci sequence terms

unsigned long \*fibArr = calloc((size\_t) n, (size\_t) sizeof (unsigned long)); //Create array to hold Fibonacci numbers

if(n == 0) //edge case

{

printf("%d\n", 0);

}

else if(n == 1) //edge case

{

printf("%d\t%d\n", 0, 1);

}

else

{

struct fibArrPtr fibAP; //Declare "fibonacci array pointer" structure

//Initialize structure members

fibAP.n = n;

fibAP.fibArr = fibArr;

struct fibArrPtr \*fibptr = calloc((size\_t) 1, (size\_t) sizeof(struct fibArrPtr));

fibptr = &fibAP;

pthread\_t splitter;

if(pthread\_create(&splitter, NULL, fibCalc, fibptr)) {

fprintf(stderr, "Error creating thread\n");

return 1;

}

else

{

/\* wait for the second thread to finish \*/

if(pthread\_join(splitter, NULL)) {

fprintf(stderr, "Error joining thread\n");

exit(0);

}

else{

printf("Fibonacci sequence up to term %d:\n", n);

//print Fibonacci sequence

for(int i = 0; i < n; i++)

{

if(i % 10 == 0)

{

printf("\n");

}

printf("%.0lu\t", fibArr[i]);

}

printf("\n");

}

}

}

return 0;

}

**Modified solution**

/\*

currently running program on Ubuntu for Windows with:

gcc p1.c -lpthread -lm

./a.out

\*/

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <math.h>

void \*fibCalc(void \*fibStruct);

struct fibArrPtr{

int n;

unsigned long \*fibArr;

unsigned long currentFib;

unsigned long prevFib;

unsigned long prevPrevFib;

pthread\_mutex\_t \*mutexArr;

};

pthread\_mutex\_t lock;

//This function is for the spawned POSIX thread to run.

// It calculates the Fibonacci sequence accurately up to the 78th term.

// Errors are introduced in the 79th term and persist onwards

// due to the computer representation of floating point numbers.

void \*fibCalc(void \*fibStruct)

{

//initialize second thread's variables from fibArrPtr struct

struct fibArrPtr \*fibAP = (struct fibArrPtr \*) fibStruct;

int n = fibAP->n;

unsigned long \*fibArr = fibAP->fibArr;

fibArr[0] = 0;

fibArr[1] = 1;

for(int j = 2; j < n; j++)

{

//printf("not there\n");

pthread\_mutex\_init(&fibAP->mutexArr[j], NULL);

pthread\_mutex\_lock(&fibAP->mutexArr[j]);

//("got here\n");

fibArr[j] = fibArr[j-1] + fibArr[j-2];

pthread\_mutex\_unlock(&fibAP->mutexArr[j]);

}

//printf("Second half finished\n");

return NULL;

}

int main(int argc, char \*\*argv)

{

int n;

char \*input = calloc((size\_t) 8, (size\_t) sizeof(char));

printf("Enter a positive integer: ");

fgets(input, 6, stdin);

n = atoi(input); //Convert user input to integer for use in threads

unsigned long currentFib = 0;

//check validity of user input

if(input[0] < '0' || input[0] > '9' || n < 0)

{

printf("Error: Input is not a valid number.\n");

exit(0);

}

//got and validated n for number of Fibonacci sequence terms

unsigned long \*fibArr = calloc((size\_t) n, (size\_t) sizeof (unsigned long)); //Create array to hold Fibonacci numbers

pthread\_mutex\_t \*mutexArr = calloc((size\_t) n, (size\_t) sizeof(pthread\_mutex\_t)); //Create array of mutex locks

if(n == 0) //edge case

{

printf("%d\n", 0);

}

else if(n == 1) //edge case

{

printf("%d\t%d\n", 0, 1);

}

else

{

struct fibArrPtr fibAP; //Declare "fibonacci array pointer" structure

//Initialize structure members

fibAP.n = n;

fibAP.fibArr = fibArr;

fibAP.currentFib = currentFib;

fibAP.fibArr[0] = 0;

fibAP.fibArr[1] = 1;

fibAP.mutexArr = mutexArr;

struct fibArrPtr \*fibptr = calloc((size\_t) 1, (size\_t) sizeof(struct fibArrPtr));

fibptr = &fibAP;

//initialize mutex lock

if (pthread\_mutex\_init(&lock, NULL) != 0)

{

printf("\n mutex init failed\n");

return 1;

}

pthread\_t splitter;

if(pthread\_create(&splitter, NULL, fibCalc, fibptr))

{

fprintf(stderr, "Error creating thread\n");

return 1;

}

else

{

int currentIndex = 0;

int i = 0;

while(1)

{

//print Fibonacci sequence

if(pthread\_mutex\_trylock(&fibAP.mutexArr[currentIndex]) == 0)

{

if(i % 10 == 0)

{

printf("\n");

}

printf("%lu\t", fibAP.fibArr[currentIndex]);

currentIndex++;

i++;

if(currentIndex == n)

{

break;

}

pthread\_mutex\_unlock(&fibAP.mutexArr[currentIndex]);

}

}

printf("\n");

}

}

pthread\_mutex\_destroy(&lock);

return 0;

}

**Problem 2**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <string.h>

//First Thread, exists to print out the prime values

void \*myThreadFun(void \*vargp){

int \*n = (int\*) vargp;

int number = \*n;

int j,i;

int primes[number+1];

printf("PRIME Thread\n");

//populate an array with integers up to the inputted number

for(i = 2; i<=number; i++)

primes[i] = i;

//While loop to test if a number is prime and set all composite numbers equal to zero

i = 2;

while ((i\*i) <= number)

{

if (primes[i] != 0)

{

for(j=2; j<number; j++)

{

if (primes[i]\*j > number)

break;

else

//makes composite elements 0

primes[primes[i]\*j]=0;

}

}

i++;

}

//prints the prime numbers

for(i = 2; i<=number; i++)

{

//prints all nonzero array elements

if (primes[i]!=0)

printf("%d\n",primes[i]);

}

printf("\n\n");

return NULL;

}

//second thread prints the subset of the first array that have prime swaps

void \*inPrint(void \*vargp){

int \*n = (int\*) vargp;

int number = \*n;

int j,i;

int primes[number+1];

int swap[number+1];

printf("Inverse Thread\n");

//generates the prime array the same way as before

for(i = 2; i<=number; i++)

primes[i] = i;

i = 2;

while ((i\*i) <= number)

{

if (primes[i] != 0)

{

for(j=2; j<number; j++)

{

if (primes[i]\*j > number)

break;

else

//makes composite elements 0

primes[primes[i]\*j]=0;

}

}

i++;

}

//swaps the digits of all the two didgit prime numbers

for(i = 2; i<=number; i++)

{

swap[i]=primes[i];

if (primes[i]>=10){

int twoD = swap[i];

int tens = (twoD/10)%10;

int ones = twoD%10;

twoD = (ones\*10)+tens;

swap[i] = twoD;

}

}

//new loop to test for composite numbers and set their values to zero

i = 2;

while(i<number){

for(j=2; j<=swap[i]/2; ++j)

{

// condition for nonprime number

if(swap[i]%j==0)

{

swap[i]=0;

break;

}

}

i++;

}

//clears the values in the prime array so it can be reused

memset(primes, 0, number\*sizeof(int));

//swap the numbers back again

for(i = 2; i<=number; i++)

{

primes[i]=swap[i];

if (swap[i]>=10){

int twoD = primes[i];

int tens = (twoD/10)%10;

int ones = twoD%10;

twoD = (ones\*10)+tens;

primes[i] = twoD;

}

}

//if the number is not 0 and greater than ten we print it as part of this subset

for(i = 2; i<=number; i++)

{

if (primes[i]!=0 && primes[i]>=10)

printf("%d\n",primes[i]);

}

printf("\n\n");

return NULL;

}

int main()

{

printf("Enter the number\n");

int i,j;

int\* number;

scanf("%d",number);

printf("\n");

pthread\_t tid,tid2;//two thread ids

pthread\_create(&tid, NULL, myThreadFun, (void\*)number); //main thread

pthread\_create(&tid2, NULL, inPrint, (void\*)number); //secondary thread

pthread\_join(tid, NULL); //thread 1 finishes

pthread\_join(tid2, NULL);//thread two finishes

return 0;

}

**Problem 3**

Dekker’s Algorithm satisfies all three requirements of the critical section problem. Mutual exclusion is the first requirement, this ensures once a processor has set it’s flag to true, then the next process has to wait for the first process to update the value of its turn before continuing. The second requirement is progress, this is checked by following the flag and turn variables. If a process wishes to enter a critical section, it sets its flag variable to true, the value of turn is set once the processor leaves the critical section. If the process decides to enter the same critical section again then it repeats the same process of entering a critical section. The last requirement is bounded waiting, this requirement is observed by the turn variable. If two separate processes decide to enter a critical section, they set their flag to true but only the thread whose turn it is is able to enter the critical section, the other processor must wait. The waiting processor would have to wait indefinitely while the first processor entered and exited the critical section if it wasn’t for block waiting. Dekker’s algorithm ensures that the processes turn turns and the first process cannot enter and exit the critical section indefinitely, but instead but let the second processor enter upon its exit.

**Problem 4**

1. Spinlocks are not appropriate for single-processor systems because they require busy waiting. Using a spinlock on a single-processor system would waste time and resources. Consider a system that utilizes time slice scheduling, where two processes A and B, both utilizing spinlocks, need to run. If a thread in process A acquires the critical section and the process uses up its time slice before finishing the critical section, then process B runs and gets to the spinlock to wait for the critical section, process B then needs to wait until it uses up its time slice so that process A may finish the critical section and set the condition for process B to break out of the spinlock. This situation has a lot of wasted cycles. In the worst case, the spinlock could cause a deadlock on the processor and there exist no other processors to break the deadlock. However, spinlocks are often used on multiprocessor systems because a concurrently running process could break the deadlock on another process by breaking a thread that is in a spinlock out of the lock loop.
2. Disabling interrupts is not sufficient for a multi-processor system because it does not guarantee mutual exclusion from multiple processes on multiple processors. This means that code running on other processors could access a resource that should be protected in a critical section even though the processor that is supposed to have exclusive access to the critical section is running that section. The processor that is first running the critical section can only provide mutual exclusion from other processes on the same processor.

On a single processor system, disabling interrupts could work because the processor can only run one process at a time. There are no other concurrent processes running on the system, so the process running the critical section will have mutually exclusive access to the critical section. On this kind of system, interrupts are the only way a process can lose control of a critical section.

On a multi-core system, we can make this approach work by imposing the requirements that you can choose which processor an interrupt goes to. When you implement a synchronization primitive, you must implement two functions: lock and unlock. In the unlock function, an interrupt should be sent to signal that the lock is ready to be unlocked before unlocking. A disadvantage of this strategy is that the wrong lock may be unlocked if the interrupt is sent to the wrong processor.

**Problem 5**

1. Describe how the compare\_and\_swap() instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.

The Compare and swap function can provide mutual exclusion by comparing a value and setting based on the comparison for instance switching a lock between the value of true and false (0,1); Calling compare and swap will swap if the value entered matches the value of lock currently it returns true otherwise it is false

Pseudo-Code:

for implementation of compare and swap

int Compare\_And\_swap(int\* value, int old,int new){

if \*p != old {

return 0 (false);

}

\*p = new

return 1 (true);

}

1. Consider how to implement a mutex lock using an atomic hardware instruction. Assume that the following structure defining the mutex lock is available:

typedef struct {

int available;

}

lock;

where (available == 0) indicates the lock is available; a value of 1 indicates the lock is unavailable.

Using this struct, illustrate how the following functions may be implemented using the test\_and\_set() and compare\_and\_swap() instructions.

• void acquire(lock \*mutex)

• void release(lock \*mutex)

Be sure to include any initialization that may be necessary.

int my\_pthread\_mutex\_init(my\_pthread\_mutex\_t\* mutext){

//Return error code if cannot init mutex

if (mutex == NULL) {

return EINVAL;

}

mutex->available = 0;

return 0; // return if successful

}

int my\_pthread\_mutex\_Acquire(my\_pthread\_mutex\_t\* mutex){

if (mutex == NULL) return EINVAL;

int ret = mutex->available;

\* mutex->available = 1;

return ret;

}

int my\_pthread\_mutex\_Release(my\_pthread\_mutex\_t\* mutex){

{

\*mutex->available = 0;

return 0;

}

//------------------------OR--------------------------------\\

int my\_pthread\_mutex\_Acquire(my\_pthread\_mutex\_t\* mutex){

if (mutex == NULL) return EINVAL;

\_\_sync\_lock\_test\_and\_set ( mutex->available,1);

}

int my\_pthread\_mutex\_Release(my\_pthread\_mutex\_t\* mutex){

{

\_\_sync\_lock\_test\_and\_set ( mutex->available,0);

}

c) Show how to implement the wait() and signal() semaphore operations in multiprocessor

environments using test and set() instruction.

int wait()

{

return my\_pthread\_mutex\_Acquire(&lock);

}

int signal()

{

my\_pthread\_mutex\_Release(&lock);

return 0;

}

do {

while(wait()); // do nothing

// critical section

my\_pthread\_mutex\_Release(&lock);

// remainder section

} while(TRUE);

**Problem 6**

monitor procMon{

int n; //upper limit of processes

bool check = true; //bool to lock out extra processes

int totSum = 0; //int to keep track of sum of numbers processes input

void fileAcc(int x){

while(check){ //as long as check is true the process accesses the file

printf("Proccess has accessed the file. \n");

totSum+=x; //add x to the summation of process numbers

if(totSum> n){ //if the summation is greater than n check is set to false to lock other processes past n

check = false;

wait(pid); //puts the current process to sleep if check becomes false

}

}

}

void end(int x){//function to reset the lock after the process ends

totSum -= x;

check = true;

}

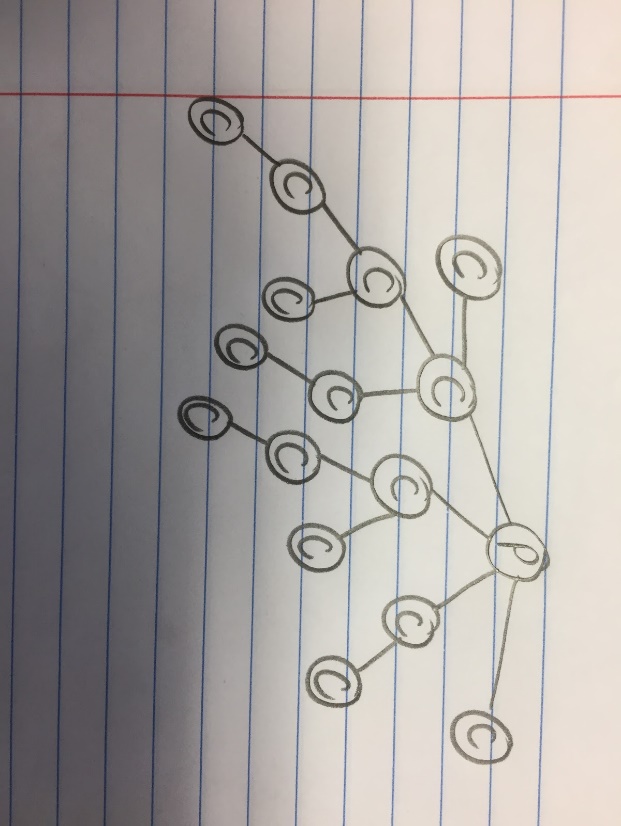
}

1. No matter what the signaling process does, either continuing its own execution or transferring control to the signalled process, the monitor should work about the same.

However, if it switches control to the signalled process the risk is run of waking the signalled process before there is enough memory on the stack to run it. Or a new process might access the file system before the current one exits. This renders the monitor somewhat obsolete as mutual exclusion isn't guaranteed.

**Problem 7**

1. This is the final process tree because after each command fork() is called a new child is made for each existing processor. This forms the total of 16 processes in the final drawing, all in steady state. This diagram shows the tree if none of the children have kill() called to them.
2. At each tree node, the PC increases at line stating “fork” for each node because that is when each processor is forked. The exact line states: “pid[i] = fork();”. Calling the fork() creates a new process with a new identification number. If it is a parent the process id would be 1, if it is a child then the process id would be 0.



Version 2:

1. If the kill() signal is called on the child then that child is removed from the tree and any child after is also killed. The killed child’s children get killed because if there exists a child whose parent got killed, then the child’s process value will be greater than zero, causing it to be killed anyway.
2. This time the kill() signal is pulled out of the first if statement, but all of the tasks up to “pid[i] = fork()” are unchanged. When the kill() signal is called, the child’s pid from the first fork is saved and then the child is killed. Then the parent calls gn1() and never returns causing the for loop to be abandoned. At this point the child will block and the parent will terminate it.

**Problem 8**

**Part 1:**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

#include <sys/types.h>

#include <sys/syscall.h>

#include <semaphore.h>

void employee();

void clearner();

enum{false = 0,true = 1 };

sem\_t in\_office;

int main(int argc, char const \*argv[])

{

sem\_init(&in\_office,0,1);

size\_t size = 2;

pthread\_t t[size];

for (int i = 0; i < size; i++)

{

if(i%2==0)

{

pthread\_create(&t[i],NULL,(void\*)\*employee,NULL);

}

else

{

pthread\_create(&t[i],NULL,(void\*)\*clearner,NULL);

}

}

for (int i = 0; i < size; i++)

{

pthread\_join(t[i],NULL);

}

sem\_destroy(&in\_office);

return 0;

}

void working\_window()

{

printf("Working on window -%ld-\n",syscall(SYS\_gettid));

}

void clean\_office()

{

printf("cleaning window -%ld-\n",syscall(SYS\_gettid));

}

void break\_time()

{

printf("Taking a Break -%ld-\n",syscall(SYS\_gettid));

sleep(5);

pthread\_exit(NULL);

}

void employee()

{

while(true==1)

{

sem\_wait(&in\_office);

working\_window();

sem\_post(&in\_office);

break\_time();

}

}

void clearner()

{

while(true==1)

{

sem\_wait(&in\_office);

clean\_office();

sem\_post(&in\_office);

break\_time();

}

}

**Part 2:**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

#include <sys/types.h>

#include <sys/syscall.h>

#include <semaphore.h>

#define \_GNU\_SOURCE

void employee();

void clearner();

enum{false = 0,true = 1 };

sem\_t in\_office;

sem\_t canWork;

int main(int argc, char const \*argv[])

{

int N = 10;

size\_t size = 2\*N;

sem\_init(&in\_office,0,N);

sem\_init(&canWork,0,1);

pthread\_t t[size];

for (int i = 0; i < size; i++)

{

if(i!=10)

{

pthread\_create(&t[i],NULL,(void\*)\*employee,NULL);

}

else

{

pthread\_create(&t[i],NULL,(void\*)\*clearner,NULL);

}

}

for (int i = 0; i < size; i++)

{

pthread\_join(t[i],NULL);

}

sem\_destroy(&in\_office);

sem\_destroy(&canWork);

return 0;

}

void working\_window()

{

printf("Working on window -%ld-\n",syscall(SYS\_gettid));

pthread\_yield();

}

void clean\_office()

{

printf("cleaning window -%ld-\n",syscall(SYS\_gettid));

pthread\_yield();

}

void break\_time()

{

printf("Taking a Break -%ld-\n",syscall(SYS\_gettid));

pthread\_exit(NULL);

}

void employee()

{

while(true==1)

{

sem\_trywait(&canWork);

sem\_wait(&in\_office);

working\_window();

sem\_post(&in\_office);

sem\_post(&canWork);

break\_time();

}

}

void clearner()

{

while(true==1)

{

sem\_wait(&canWork);

clean\_office();

sem\_post(&canWork);

break\_time();

}

}

**Part 3:**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

#include <sys/types.h>

#include <sys/syscall.h>

#include <semaphore.h>

#define \_GNU\_SOURCE

void employee();

void clearner();

enum{false = 0,true = 1 };

sem\_t waits;

sem\_t in\_office\_C;

sem\_t in\_office\_E;

static int cleaning = 1;

int main(int argc, char const \*argv[])

{

int N = 10;

size\_t size = N;

sem\_init(&waits,0,1);

sem\_init(&in\_office\_E,0,5);

sem\_init(&in\_office\_C,0,5);

pthread\_t t[size];

for (int i = 0; i < size; i++)

{

if(i%2==0)

{

pthread\_create(&t[i],NULL,(void\*)\*employee,NULL);

}

else

{

pthread\_create(&t[i],NULL,(void\*)\*clearner,NULL);

}

}

for (int i = 0; i < size; i++)

{

pthread\_join(t[i],NULL);

}

sem\_destroy(&in\_office\_E);

sem\_destroy(&in\_office\_C);

sem\_destroy(&waits);

return 0;

}

void working\_window()

{

printf("Working on window -%ld-\n",syscall(SYS\_gettid));

pthread\_yield(NULL);

}

void clean\_office()

{

printf("cleaning window -%ld-\n",syscall(SYS\_gettid));

pthread\_yield(NULL);

}

void break\_time()

{

int value;

sem\_wait(&waits);

sem\_getvalue(&in\_office\_E,&value);

if(value == 5)

cleaning = -1;

else

cleaning = 1;

sem\_post(&waits);

pthread\_exit(NULL);

}

void employee()

{

while(true==1)

{

int value;

sem\_wait(&waits);

sem\_getvalue(&in\_office\_C,&value);

if(value == 5)

cleaning = 1;

else

cleaning = -1;

sem\_post(&waits);

while(cleaning<0)

{

}

sem\_wait(&in\_office\_E);

working\_window();

printf("Taking a Break -%ld-\n",syscall(SYS\_gettid));

sem\_post(&in\_office\_E);

break\_time();

}

}

void clearner()

{

while(true==1)

{

int value;

sem\_wait(&waits);

sem\_getvalue(&in\_office\_E,&value);

if(value == 5)

cleaning = -1;

else

cleaning = 1;

sem\_post(&waits);

while(cleaning>0)

{

}

sem\_wait(&in\_office\_C);

clean\_office();

printf("Taking a Break -%ld-\n",syscall(SYS\_gettid));

sem\_post(&in\_office\_C);

break\_time();

}

}

**Problem 9**

**Part 1:**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

#include <sys/types.h>

#include <sys/syscall.h>

#include <semaphore.h>

#define \_GNU\_SOURCE

void Teacher();

void Child();

void Parent();

sem\_t TeacherC;

sem\_t ParentC;

sem\_t ChildC;

int main(int argc, char const \*argv[])

{

int N = 10;

int R = 4;

int T = R/2;

int P = N-R-T;

size\_t size = N;

sem\_init(&TeacherC,T,0);

sem\_init(&ChildC,R,0);

sem\_init(&ParentC,0,P);

pthread\_t t[size];

int i = 0;

pthread\_create(&t[0],NULL,(void\*)\*Child,NULL);

pthread\_create(&t[1],NULL,(void\*)\*Child,NULL);

pthread\_create(&t[2],NULL,(void\*)\*Child,NULL);

pthread\_create(&t[3],NULL,(void\*)\*Teacher,NULL);

pthread\_create(&t[4],NULL,(void\*)\*Parent,NULL);

pthread\_create(&t[5],NULL,(void\*)\*Teacher,NULL);

pthread\_create(&t[6],NULL,(void\*)\*Child,NULL);

pthread\_create(&t[7],NULL,(void\*)\*Parent,NULL);

pthread\_create(&t[8],NULL,(void\*)\*Parent,NULL);

pthread\_create(&t[9],NULL,(void\*)\*Parent,NULL);

for (int i = 0; i < size; i++)

{

pthread\_join(t[i],NULL);

}

sem\_destroy(&TeacherC);

sem\_destroy(&ChildC);

sem\_destroy(&ParentC);

return 0;

}

void go\_home()

{

printf("I am leaving -%ld-\n",syscall(SYS\_gettid));

pthread\_exit(NULL);

}

//-------------------------------\\

void teacher\_enter()

{

sem\_post(&TeacherC);

}

void teach()

{

printf("I am Teaching -%ld-\n",syscall(SYS\_gettid));

sleep(5);

}

void teacher\_exit()

{

int Cvalue,Tvalue;

sem\_getvalue(&ChildC,&Cvalue);

sem\_getvalue(&TeacherC,&Tvalue);

if(Cvalue/2 < Tvalue)

{

sem\_wait(&TeacherC);

go\_home();

}

else

{

return;

}

}

//-------------------------------\\

void child\_enter()

{

sem\_post(&ChildC);

}

void learn()

{

printf("I am learning -%ld-\n",syscall(SYS\_gettid));

sleep(5);

}

void child\_exit()

{

sem\_wait(&ChildC);

go\_home();

}

//--------------------------------\\

void parent\_enter()

{

sem\_post(&ParentC);

}

void verify\_compliance()

{

int Cvalue,Tvalue;

printf("I am adult doing adult things -%ld-\n",syscall(SYS\_gettid));

sem\_getvalue(&ChildC,&Cvalue);

sem\_getvalue(&TeacherC,&Tvalue);

if(Cvalue/2 >= Tvalue)

{

printf("Regulation is Met\n");

}

sleep(5);

}

void parent\_exit()

{

sem\_post(&ParentC);

go\_home();

}

//-------------------------------\\

void Teacher()

{

while(1==1)

{

teacher\_enter();

teach();

teacher\_exit();

}

}

void Child()

{

while(1==1)

{

child\_enter();

learn();

child\_exit();

}

}

void Parent()

{

while(1==1)

{

parent\_enter();

verify\_compliance();

parent\_exit();

}

}

**Part 2:**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

#include <sys/types.h>

#include <sys/syscall.h>

#include <semaphore.h>

#define \_GNU\_SOURCE

void Teacher();

void Child();

void Parent();

sem\_t TeacherC;

sem\_t ParentC;

sem\_t ChildC;

int main(int argc, char const \*argv[])

{

int N = 10;

int R = 5;

int T = R/2;

int P = N-R-T;

size\_t size = N;

sem\_init(&TeacherC,T,0);

sem\_init(&ChildC,R,0);

sem\_init(&ParentC,0,P);

pthread\_t t[size];

int i = 0;

for (int i = 0; i < size; i++)

{

if(i%3==0)

{

pthread\_create(&t[i],NULL,(void\*)\*Teacher,NULL);

}

else if(i%3==1)

{

pthread\_create(&t[i],NULL,(void\*)\*Parent,NULL);

}

else

{

pthread\_create(&t[i],NULL,(void\*)\*Child,NULL);

}

}

for (int i = 0; i < size; i++)

{

pthread\_join(t[i],NULL);

}

sem\_destroy(&TeacherC);

sem\_destroy(&ChildC);

sem\_destroy(&ParentC);

return 0;

}

void go\_home()

{

printf("I am leaving -%ld-\n",syscall(SYS\_gettid));

pthread\_exit(NULL);

}

//-------------------------------\\

void teacher\_enter()

{

sem\_post(&TeacherC);

}

void teach()

{

printf("I am Teaching -%ld-\n",syscall(SYS\_gettid));

sleep(5);

}

void teacher\_exit()

{

int Cvalue,Tvalue;

sem\_getvalue(&ChildC,&Cvalue);

sem\_getvalue(&TeacherC,&Tvalue);

if(Cvalue/2 < Tvalue)

{

sem\_wait(&TeacherC);

go\_home();

}

else

{

return;

}

}

//-------------------------------\\

void child\_enter()

{

sem\_post(&ChildC);

}

void learn()

{

printf("I am learning -%ld-\n",syscall(SYS\_gettid));

sleep(5);

}

void child\_exit()

{

sem\_wait(&ChildC);

go\_home();

}

//--------------------------------\\

void parent\_enter()

{

sem\_post(&ParentC);

}

void verify\_compliance()

{

int Cvalue,Tvalue;

pthread\_t o;

printf("I am adult doing adult things -%ld-\n",syscall(SYS\_gettid));

sem\_getvalue(&ChildC,&Cvalue);

sem\_getvalue(&TeacherC,&Tvalue);

if(Cvalue/2 >= Tvalue)

{

printf("Regulation is Met\n");

}

else

{

printf("Fetching A Teacher to meet Regulation\n");

pthread\_create(&o,NULL,(void\*)\*Teacher,NULL);

}

sleep(5);

}

void parent\_exit()

{

sem\_post(&ParentC);

go\_home();

}

//-------------------------------\\

void Teacher()

{

while(1==1)

{

teacher\_enter();

teach();

teacher\_exit();

}

}

void Child()

{

while(1==1)

{

child\_enter();

learn();

child\_exit();

}

}

void Parent()

{

while(1==1)

{

parent\_enter();

verify\_compliance();

parent\_exit();

}

}

**Problem 10**

1. Beginning with the two functions declared in the beginning of the program, void h() reads a number from the pipe and stores it in the variable num. void \*f() receives arg[k], which has the kth k value from the for loop in main() that deals with arg[]. This function adds k to res, so res changes cumulatively by k each time in the arg[] loop in main(). Specifically, res is initially 0, then res = 0+0 = 0, then res = 0+1 = 1, then res = 1+2 = 3.

The main function of the program begins by initializing integers, a void pointer, a semaphore and a pipe. It also sets up the parent process to listen for when one of its children terminate to run function h. The program then enters a loop in which it forks a process in each iteration.

In each child process, num is incremented from 2 to 3, then written to the pipe. If num is ever greater than 3, then num is not written to the pipe and the child becomes a zombie until the parent process either waits on it or sends a kill signal.

Additionally, each child process creates threads that run function f. This uses the arg[] array to add to the child’s copy of res. After the threads are done running, the child terminates them in the following for loop. Since num should be less than 4 anyways in the child process, the child then writes its copy of num to the pipe and exits, passing its value of res to the parent.

In the parent process, the parent first writes its own value of num to the pipe, which is 2. This always happens before the last child writes its own value of num to the pipe, so the parent’s value of num is always overwritten by one of the child’s value of num in the pipe. The parent waits until all of the child processes have terminated before reading from the pipe, so the final result of num is always 3.

The last for loop that modifies the value of res runs two times because at that point in the code, the parent has not yet read from the pipe, so the value of num is unchanged and is still 2. In the loop, the parent waits on a child until that child terminates normally. After a child terminates normally, the parent takes the argument that the child passed to exit(), which is the child’s copy of res, and adds that to its own copy of res. This occurs once more in the loop, so the final value of res is 6.

1. See image below
2. See image below
3. See image below
4. Line 54 prints “Final result1: 6” and line 56 prints “Final result2: 3”

