Mhammed Alhayek
Brian Monticello
Sean Olejar

Intro to Comp Systems HW 2

Problem 1:

```c
#include <stdio.h>           // for standard input/output
#include <math.h>            // for pow function
#include <stdbool.h>  // required to use bool in C

#include <pthread.h>  // for threads

#include <stdlib.h>          // for atoi() function



struct arg_struct {

        int arg1;            // integer n

        bool* arg2;          // boolean array of primes

};



/* Iterative function to reverse digits of num*/

// reference:
http://www.geeksforgeeks.org/write-a-c-program-to-reverse-digits-of-a-number/
int reverse_digits(int num){

        // initialize the reverse number

   int rev_num = 0;

   while(num > 0){

        // everytime we add a digit to rev_num, we move the previous digit left by
multiplying by 10
        rev_num = rev_num*10 + num%10;

        num = num/10;

   }

   return rev_num;

}
```

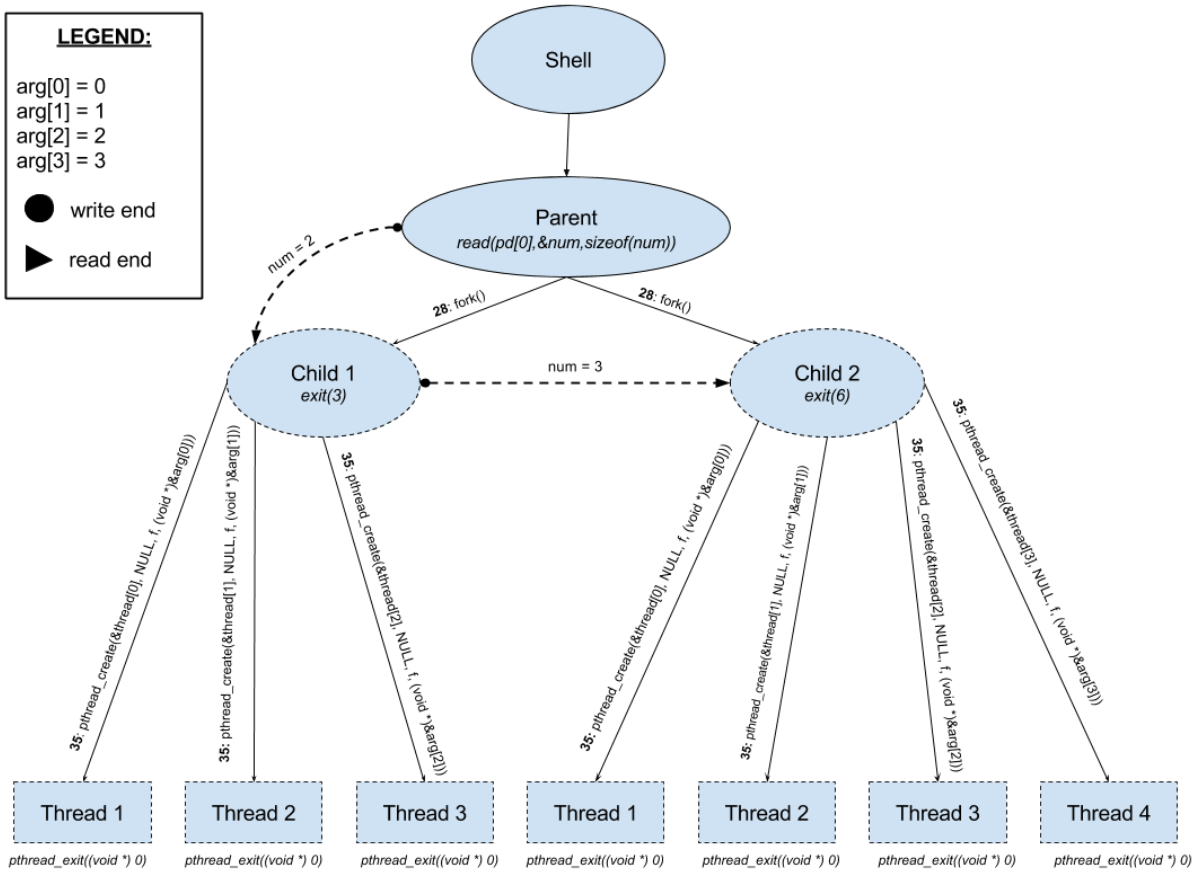
```c
void *rev_prime(void* param){
	// extracting the parameters and saving them in n and primes
	struct arg_struct *args = param;
	int n = args->arg1;
	bool* primes = args->arg2;

	for (int i = 10; i<=n; i++){
		int revi = reverse_digits(i);		// reverse of i

		// if the reverse of i is greater than n, we can't check it
		if (revi <= n){
			if (primes[i] && primes[revi]){
				// declare them as false after they are printed so they don't get printed twice
				primes[i] = false;
				primes[revi] = false;
				printf("%d and its reverse %d are primes.\n", i, revi);
			}
		}
	}

	pthread_exit(0);
}

void *all_primes(void *param){
	// extracting the parameters and saving them in n and primes
	struct arg_struct *args = param;
	int n = args->arg1;
	bool* primes = args->arg2;

	//initialize primes to true
```

```
        for (int i = 0; i<(n+1); i++){
                primes[i] = true;
        }

        int root = pow(n,0.5);        //rounded down to int
        for (int i = 2; i<=root; i++){
                if (primes[i]){
                        for (int j = i*i; j<=n; j+=i){
                                primes[j] = false;
                        }

                }
        }
        printf("The prime numbers not exceeding %d are:", n);
        for (int i = 0; i <=n; i++){
                if (primes[i]){
                        printf(" %d", i);
                }
        }

        printf("\n");
        pthread_exit(0);
}

int main(int argc, char *argv[]){
        // this program needs an extra argument for n, so argc = 2
        if (argc != 2) {
                fprintf(stderr,"usage: a.out <integer value>\n");
                return -1;
        }
```

```
        int x = atoi(argv[1]);        // the int entered by user
        // x must be a positive number
        if (x < 0) {
                fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
                return -1;
        }

        pthread_t tid[2];             // the thread identifier, need 2 for 2 threads
        pthread_attr_t attr;  // set of thread attributes

        // get the default attributes
        pthread_attr_init(&attr);

        // create a bool array for all the ints from 0 to n
        // array must go to n+1 so that primes[n] can be used
        bool primes[x+1];     // declaring this here so that it can be used by all threads

   // using struct to pass n and the boolean array to both functions all_primes() and
rev_prime()
   struct arg_struct func_args;
   func_args.arg1 = x;
   func_args.arg2 = primes;

        //create the thread
        pthread_create(&tid[0], &attr, all_primes, (void*) &func_args);

        // wait for the thread to exit
        pthread_join(tid[0],NULL);
```

```
        // create second thread
        pthread_create(&tid[1], &attr, rev_prime, (void*) &func_args);
        pthread_join(tid[1],NULL);


        return 0;
}
```

Problem 2:



I. Without a mutex lock on the variable res in the function f, there is a race condition because 3 threads attempt to access the value at the same time. This causes the actual output of this program to be non-deterministic. For the sake of simplicity, assume that the race condition doesn't occur. In this situation, Line 54 prints "Final result1: 9"

Line 56 prints nothing. This is because on line 55, there is a read() which causes the parent process to block. This block occurs because nothing is ever written to the file descriptor to be read in! This is why the parent process is in a "steady state"

Problem 3:

I. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal(). There are two types of semaphores, counting semaphores and binary semaphores. The value of a counting semaphore can range over an unrestricted domain. The value of a binary semaphore can only be 0 or 1. Binary semaphores are useful for providing mutual exclusion. Counting semaphores are useful for controlling access to limited resources. For example, the semaphore can be initialized to the number of resources available. Anytime a process uses the resource, it performs a wait() operation on the semaphore to decrement its count. Anytime a process releases a resource, it performs a signal() operation on the semaphore to increment its count. When the semaphore count reaches 0, any process that wishes to use that resource has to block until the count is greater than 0. Semaphores can also be used to synchronize processes. The synchronization problems that a semaphore may resolve but the simple lock cannot is a semaphore can cause a system to block while a simple lock causes a system to busy wait. Busy waiting wastes CPU cycles.

II. Dijkstra defined a semaphore as a non-negative integer that supports two operations: P() and V(). P stood for “prolaag” which is dutch for try. V stood for “verlaag” which is dutch for decrease. Busy waiting is when a process repeatedly checks if a condition is true, so the CPU can’t be used until the condition is true. It is undesirable because it doesn’t optimize use of the CPU. For example if a single CPU is shared among many processes, busy waiting causes many CPU cycles to be wasted because a different process could have used those cycles productively.

III. An implementation of the semaphore that avoids the busy wait is as follows:

```
typedef struct {
        int value;
        struct process *list;
} semaphore;

wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}

signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
```

```
        }
    }
```

The way this implementation works is now when a process executes wait() and finds the semaphore to be less than 0, it gets added to a waiting list and blocks itself rather than busy wait. When a process executes signal() and finds the semaphore to be less than 0, it wakes up a process from the waiting list. Whenever the semaphore is less than 0, that means there are processes waiting on the list. The waiting is now restricted to the critical sections of application programs as opposed to the entry section. This is good because the critical sections are short and rarely occupied

IV.

```c
#include <stdio.h>           // for standard input/output
#include <stdbool.h>  // required to use bool in C
#include <pthread.h>  // for threads
#include <stdlib.h>          // for atoi() function
#include <semaphore.h>       // for semaphores
#include <unistd.h>          // for sleep function


sem_t K;              // global semaphore variable for seats available -- counting semaphore
sem_t csdesk;  // global semaphore variable for customer service desk -- binary semaphore
char* bank_name = "Bank of America";        // assignment requests a shared variable, so all my customers will share the bank name


// assignment wants shared variable, but there really isnt much to share
// the below two variables will be shared just to varify that customers get serviced in the order that they sat down
int seat_number = 0;
int serv_number = 0;




void make_transaction(){


        // decrement the service desk availability

        // if the value is 0, it will block until the desk is available

        sem_wait(&csdesk);


        // at this point, the customer is going to the cs desk. We can free up his seat
```

```
        sem_post(&K);         // make another seat available


        serv_number++;

        printf("Customer %u is being serviced!\n", (int) pthread_self());

        printf("Customer %u is the %d person to get serviced.\n",(int) pthread_self(),
serv_number);
        sleep(3);             // 3 seconds to perform a transaction for this simulation


        // sem_post will increment the csdesk semaphore.

        // if csdesk is less than 0 after the increment, there is a thread blocking
waiting for the desk
        // that blocking thread will be unblocked

        sem_post(&csdesk);

}


void take_a_walk(){

        printf("Customer %u is taking a walk.\n", (int) pthread_self());

        sleep(3);      // taking a walk will be simulated by a 3 second pause


}


void return_home(){

        printf("Customer %u is going home.\n", (int) pthread_self());

        pthread_exit(0);      // customer leaves, exit his thread

}



void bank_client(){

        int seats_available;

        printf("Customer %u arrived at %s.\n", (int) pthread_self(),bank_name);

        while(1) {

                sem_getvalue(&K, &seats_available);
```

```
                printf("Currently there are %d seats available for Customer %u.\n",
seats_available, (int) pthread_self());


                if (seats_available > 0){     /* if seats available */

                        sem_wait(&K);         // occupy a seat, this will decrement K

                        printf("Customer %u got a seat!\n", (int) pthread_self());

                        seat_number++;

                        printf("Customer %u is the %d person to get a seat.\n", (int)
pthread_self(), seat_number);
                        make_transaction();   // attempt to make a transaction, will block
until it can


                        // -- question for group --

                        /*

                        We can make the seat available after the customer is done being
serviced
                        OR we can make the seat available as soon as the customer goes to
the desk.
                        Technically, the seat is available as soon as the customer is being
serviced,
                        so that is what I implemented. If we want to change that to the
first option,
                        we simply move "sem_post(&K)" from the make_transaction() function
to here.
                        */


                        break;

                }

                else{

                        // attempt to do this with a wait instead

                        take_a_walk();

                }

        }


        return_home();

}
```

```
int main(int argc, char* argv[]){
        // this program needs an extra argument for K, the number of seats
        // program also needs an extra argument for number of customers
        if (argc != 3) {
                fprintf(stderr,"usage: a.out <integer value> <integer value>\n");
                return -1;
        }


        int x = atoi(argv[1]);        // the int entered by user for K
        int custs = atoi(argv[2]);    // the int entered by user for customers
        // x and custs must be a positive number
        if (x < 0) {
                fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
                return -1;
        }

        if (custs < 0) {
                fprintf(stderr,"%d must be >= 0\n",custs);
                return -1;
        }
        sem_init(&K, 0, x);           // initialize semaphore for seats available
        sem_init(&csdesk,0,1);        // intialize semaphore for the service desk
        pthread_t tid[custs];         // the thread identifier, need one for each cust
        pthread_attr_t attr;          // set of thread attributes

        // get the default attributes
        pthread_attr_init(&attr);

        for (int i = 0; i <custs; i++){
```

```
            // create the thread
            pthread_create(&tid[i], &attr, (void*) bank_client, NULL);
            // each time a thread is created, a customer arrives at the bank.
            // customers will arrive once every second.
            sleep(1);
        }

        for (int i =0; i< custs; i++){
            pthread_join(tid[i],NULL);
        }

        sem_destroy(&K);
        sem_destroy(&csdesk);
        return 0;
}
```

Problem 4:

```c
#include <stdio.h>           // for standard input/output
#include <stdbool.h>         // required to use bool in C
#include <pthread.h>         // for threads
#include <stdlib.h>          // for atoi() function
#include <semaphore.h>       // for semaphores
#include <unistd.h>          // for sleep function


//gcc -std=c99 hw2q4.c -pthread


// need a semaphore for number of children
// semaphore for number of teachers


sem_t t;       // global semaphore for number of teachers
sem_t c;       // global semaphore for number of children
float R;       // child to teacher ratio required, current ratio must stay less than this
float curr_R;  // current child to teacher ratio
sem_t r_mutex; // mutex lock for shared curr_R
sem_t t_mutex; /* mutex lock for teachers so no two teachers try to leave at the same
time */



int inf = 100000000;


void teacher_enter(){

        sem_post(&t);  // increment the number of teachers, if another was waiting to
leave, it will be released
        printf("Teacher %u just arrived.\n", (int) pthread_self());

}


void teach(){
```

```
        printf("Teacher %u is teaching.\n", (int) pthread_self());
        sleep(10);     // teachers teach for 10 seconds
}

void teacher_exit(){

        int curr_teach;              // current number of teachers
        sem_getvalue(&t, &curr_teach);
        int curr_child;              // current number of children
        sem_getvalue(&c, &curr_child);

        float temp_R;

        if ( (curr_teach - 1) != 0 ){
                temp_R = curr_child/(curr_teach-1);
        }
        else{
                if (curr_child == 0){
                        temp_R = 0;
                }
                else{
                        temp_R = inf;
                }
        }

        // if when the teacher leaves the ratio will still be met, the teacher can leave
freely
        if (temp_R <= R){
                sem_wait(&t); // decrement number of teachers
                printf("Teacher %u just left.\n", (int) pthread_self());
```

```
        }

        // else, teacher can't leave until another joins

        else {

                printf("Teacher %u wants to leave but can't. Will go back to work and then
try again later.\n", (int) pthread_self());
                teach();        // teacher is teaching again

                teacher_exit();        // teacher attempts to leave again

        }

}


void go_home(){

        printf("%u is going home now.\n", (int) pthread_self());

        pthread_exit(0);       // person leaves

}


void child_enter(){

        sem_post(&c);  // increment the number of children

        printf("Child %u just arrived.\n", (int) pthread_self());

}


void child_exit(){

        sem_wait(&c); // decrement number of children

        printf("Child %u just left.\n", (int) pthread_self());

}


void learn(){

        printf("Child %u is learning.\n", (int) pthread_self());

        sleep(10);     // children learn for 10 seconds

}


void parent_enter(){
```

```
        printf("Parent %u just arrived.\n", (int) pthread_self());
}

void verify_compliance(){

        int curr_teach;              // current number of teachers
        sem_getvalue(&t, &curr_teach);
        int curr_child;              // current number of children
        sem_getvalue(&c, &curr_child);

        if ( curr_teach!= 0 ){
                curr_R = curr_child/curr_teach;
        }
        else{
                if (curr_child == 0){
                        curr_R = 0;
                }
                else{
                        curr_R == inf;
                }
        }

        sem_post(&r_mutex);   // unlock the resources

        if (curr_R <= R){     // if the compliance is verified
                printf("Parent %u has verified compliance!\n", (int) pthread_self());
        }
        else{
                printf("Parent %u is unsatisfied because the regulation is not met!\n",
(int) pthread_self());
        }
```

```
}

void parent_exit(){
        printf("Parent %u just left.\n", (int) pthread_self());
}



void Teacher(){
        for (;;) {
                teacher_enter();
                // ... critical section ... //
                teach();
                sem_wait(&t_mutex);   // locking so that only one teacher can leave at a
time
                teacher_exit();
                sem_post(&t_mutex);
                go_home();
        }
}

void Child(){
        for (;;) {
                child_enter();
                // ... critical section ... //
                learn();
                child_exit();
                go_home();
        }
}

void Parent(){
```

```
        for (;;) {
                parent_enter();
                // ... critical section ... //
                verify_compliance();
                parent_exit();
                go_home();
        }
}

int main(int argc, char ** argv){
 //first arg = # of teach
 //2nd arg = # of children
 sem_init(&c, 0, 0);          // initialize semaphore for children to 0
 sem_init(&t, 0, 0);          // initialize semaphore for teachers to 0
 sem_init(&r_mutex, 0, 1);           // initialize semaphore for teachers to 0
 sem_init(&t_mutex, 0, 1);           // initialize semaphore for teachers to 0

 R = 3.0;      // arbitrary number for ratio required

 int count_child = 9;
 int count_teacher = 5;
 int count_parent = 4;

 //threads instantiated here
 pthread_t child_tid[count_child];
 pthread_t teacher_tid[count_teacher];
 pthread_t parent_tid[count_parent];

 pthread_attr_t attr;
 pthread_attr_init(&attr); // get the default attributes
```

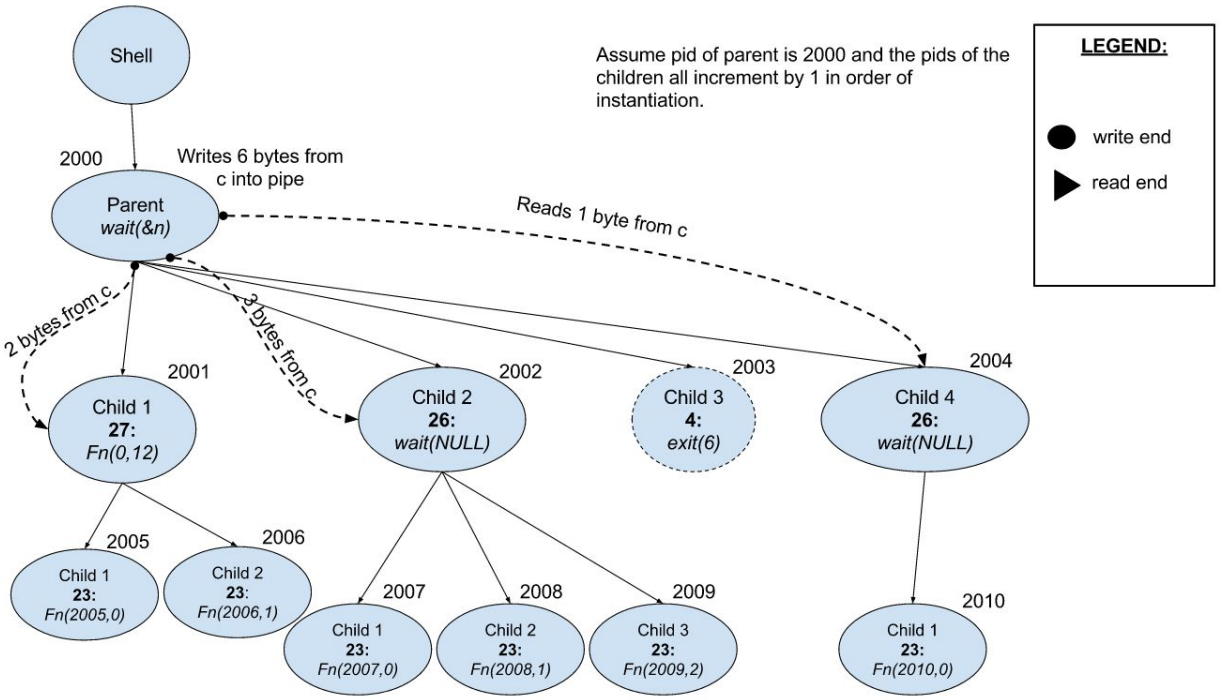
```
//creating all of the threads here
for(int i = 0;i<count_teacher;i++){
  pthread_create(&teacher_tid[i],&attr,(void*) Teacher, NULL);
}

sleep(1);

for(int i = 0;i<count_child;i++){
  pthread_create(&child_tid[i],&attr,(void *) Child, NULL);
}

for(int i = 0; i<count_parent;i++){
      pthread_create(&parent_tid[i],&attr,(void*) Parent, NULL);
  sleep(3);
}

//joining the thread here
for(int i = 0;i<count_teacher;i++){
  pthread_join(teacher_tid[i],NULL);
}

for(int i = 0;i<count_child;i++){
  pthread_join(child_tid[i], NULL);
}

for(int i = 0; i<count_parent;i++){
      pthread_join(parent_tid[i], NULL);
}
```

```
 sem_destroy(&c);
 sem_destroy(&t);
 sem_destroy(&r_mutex);
 sem_destroy(&t_mutex);

 return 0;
}
```

Problem 5:

a)

1. In UNIX, the kill command, kill (pid, SIGSTOP), sends a signal to each process specified by pid. The default signal is SIGTERM (terminate the process), however, in this command, SIGSTOP pauses the process, and restricts it from being trapped. This is default if signal not provided to the kill command. Additionally, signal (SIGINT, handler) in UNIX accepts two arguments: (1) the signal number, and (2) a pointer to the signal handler function. The signal handler function usually returns void and accepts the signal integer argument that represents a signal number that has been sent. SIGINT is the signal that is being sent to the application when it is running and someone presses CTRL-C. The default handler of this signal will quietly terminate your program.
2. In the call n = read (fd, buf, cnt), the argument cnt specifies the number of bytes to be transferred. This function returns with the number of bytes that were transferred. Therefore, if it doesn't return the number specified, it may return with a number less than cnt. A return value of zero bytes implies end of file, and a return value of -1 indicates an error.
3. The wp = wait(&status) call suspends execution of the current process until a child has changed state, meaning wait for any child process to terminate. The information about the process (that can be checked with some predefined MACRO), containing the encoded value of both the exit status of the process as well as the reason for exiting, gets stored in the argument *status* passed to wait(). Finally, the wait() system call returns the value of the process id of the exited child process. WIFSIGNALED(status) returns true if status was returned for a child that terminated abnormally due to a signal raised that it didn't catch that caused it to exit. In this case, since WIFSIGNALED(status) returned true, we know that process 1011 terminated abnormally. Therefore, since WTERMSIG(status)==3, we also know that the numeric value of the signal that was raised by the child process was 3, which is the SIGQUIT signal. This means that process 1000 must have made a call that sent the SIGQUIT signal to process 1011 in order for it to terminate abnormally.
4. When the write end of a "reader" remains open and it calls read() after the "writer" closes its pipe, the "reader" will not handle the EOF since there is still another "writer" left to write to the pipe.



A. The race condition occurs around line 18. If no synchronization (such as a mutex) is put around the read, all of the children processes will block at the read, then the order in which they are unblocked to read from the pipe will be non-deterministic. This is a problem because since only 6 bytes were written to the pipe in total, and 10 bytes are attempting to be read in total, at least one of these processes is guaranteed to read less bytes than it expects. In order to fix this, we should put a mutex around the read(), so that the read is always visited in the same order every-time. This guarantees that the bytes received remains the same every time you run it. The first 2 children always receive the correct amount of bytes and the last child is always receiving the less-than-expected amount of bytes.