# Homework 2

Ridwan Khan - rk667
Mit Patel - mdp170
Harsh Patel- hhp30

3/07/17

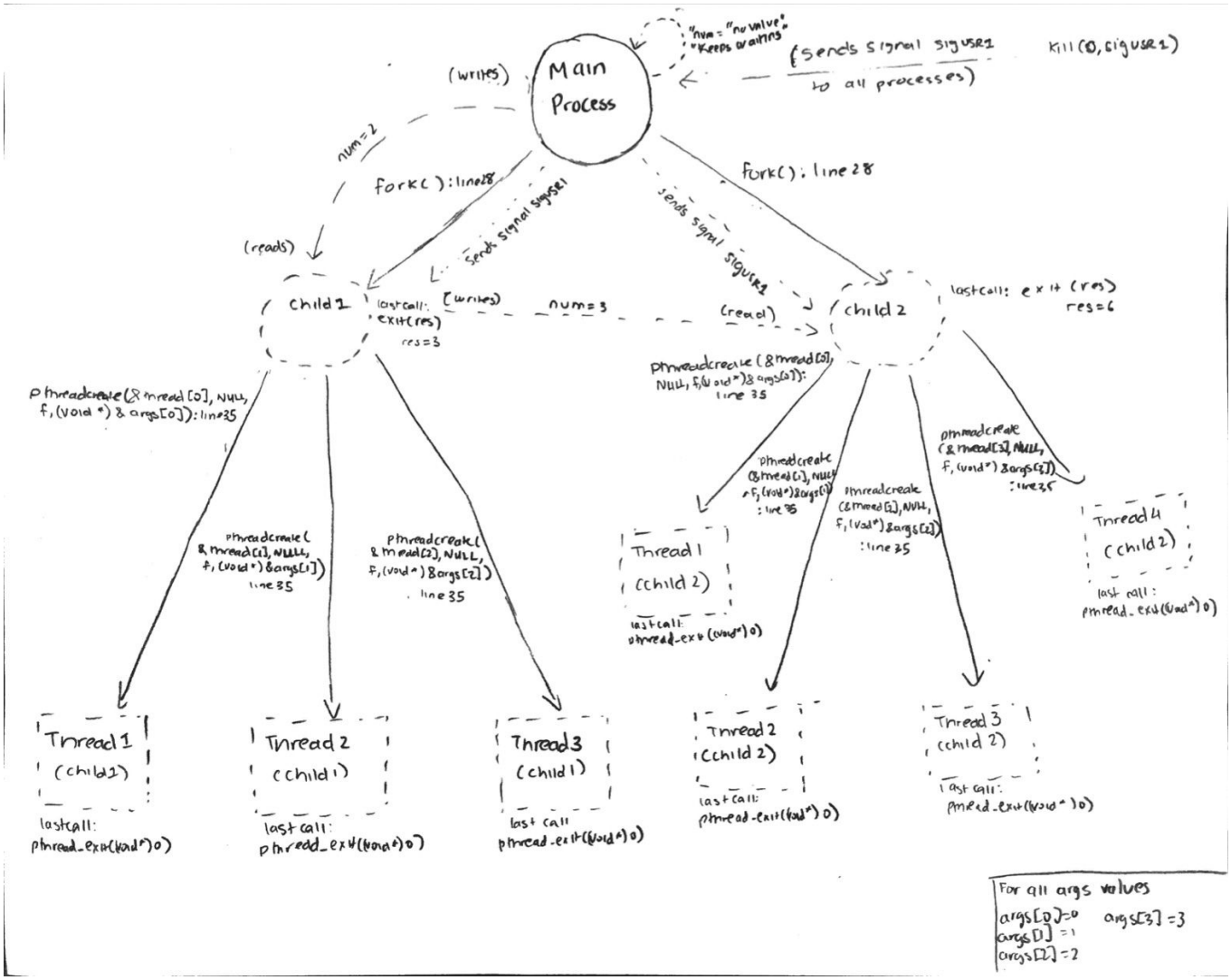
# Problem 1

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include <math.h>


void sieve(int myArray[]){
	//calculate sieve

	int number = myArray[0];

	int root = sqrt(myArray[0]);



	for (int i=2; i<=root; i++){

		if (myArray[i] == 1){
			for (int j = i*i; j<=(number+1); j= j+i){

				myArray[j] = 0;
			}
		}
	}

	//print prime numbers
	for (int i = 2; i<(number+1); i++){
		if (myArray[i] ==1){
			printf("Prime Number: %d\n", i);
		}
	}


}


void reversePrime(int myArray[]){
	//find primes whose reverse is also a prime
```

```
	int n;
	int reverse;
	int number = myArray[0];

	for (int i = 2; i < (number+1); ++i)
	{
		if (myArray[i] ==1)
		{
			//to reverse an integer used the following source

//http://www.programmingsimplified.com/c/source-code/c-program-reverse-number
			n = i;
			reverse = 0;
			while (n != 0){
				reverse = reverse*10;
				reverse = reverse + n%10;
				n = n/10;
			}
			if ((reverse <= number) && myArray[reverse])
			{
				printf("Reverse Primes: %d\n", i);
			}
		}

	}


}


int main(int argc, char *argv[]){
	if (argc != 2){
		fprintf(stderr, "usage: a.out <integer value>\n");
		return -1;
	}

	int number = atoi(argv[1]);
	int myArray[number+1];
	myArray[0] = number;

	//initialize arrat values to 1 to be true
	for (int i = 1; i<(number+1); i++){
		myArray[i] = 1;
```

```
        }



        pthread_t tid;
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        //create the thread
        pthread_create(&tid, &attr, (void*)sieve, &myArray);
        //wait for thread to return
        pthread_join(tid,NULL);


        pthread_t tid2;
        pthread_attr_t attr2;
        pthread_attr_init(&attr2);

        //create the thread
        pthread_create(&tid2, &attr2, (void*)reversePrime, &myArray);
        //wait for thread to return
        pthread_join(tid2,NULL);



        return 0;

}
```

# Problem 2




Above image included in submission folder.
Line 54 will print “Final result1: 9”.
Line 56 will not be printed.

This is because the exit(res) from line 41 sends the exit status as res, which is the value 1+2 =3 for the first child process, and 1+2+3 =6 for the second child process. These values are the iteration of 0 to <num. Num was 2 and 3 respectively. Now when the parent process waits for the first process to exit, when i = 0 for the for loop on line 49, the WIFEXITED(pstatus) is true and WEXITSTATUS(pstatus) is 3, the exit value for first child process. The res value in the parent process is now res + 3, and res is 0 at this point since the res of the parent process hasn't been changed yet, it was only changed in the address space of the child process. So the res value in parent process is now 3. Next the, the value in res of child process is 6. Now when

the parent process waits for the second process to exit, when i = 1 for the for loop on line 49, the WIFEXITED(pstatus) is true and WEXITSTATUS(pstatus) is 6, the exit value for first child process. Therefore the res of parent process is 3 + 6 =9. Therefore line 54 prints “Final result1: 9”.

Now on line 55 the process tries to read from the pipe pd. However, the pipe is empty because it was last read by the second process on line 7 in the signal handler h and has not been written to. Therefore the process blocks on line 55 until something is written to it, however no other processes or threads are left to write, therefore it is infinitely blocked on line 55 and line 56 is not printed. To end the program we hit CTRL-C.

# Problem 3

Problem 3 Bank Simulation
i) What are semaphores and what is its usefulness?

Semaphore is a tag that indicates the status of a common scarce resource. A thread would need to check with a semaphore to see if it can use a specific resource. Semaphores are counters that allow only a certain amount of threads to run at once in the program. They regulate if a certain thread will have access to something, and when a thread using the resource ends, it allows another one that is waiting to start running. They are useful to synchronize the threads. This means that a certain amount of threads will be allowed to run on a resource all at once at the same time.

What type are the synchronization problems that a semaphore may resolve but the simple lock cannot?

A few examples of synchronization problems that can be solved using semaphores include the following. One example would be that of a producer-consumer. Let's say there is one process that is creating the information that will be used by another process. In this case if you were to use a mutex each time the producer produces one item it would have to wait for the consumer to consume the item before producing the next one. However, there's no need for a perfect lock-step because producer should be able to get ahead of the consumer.
In this case if we use a semaphore, we can cover both mutual exclusion as well as the aspect of condition synchronization. Meaning only one of them, either the consumer and the producer should access the information at once (mutual exclusion) however producer can keep producing till buffer is full (condition synchronization) and the consumer has to wait till the buffer has some information (condition synchronization).

Another example of using semaphores over mutexes would be the case where you have a shared database with multiple readers and writers. In this case using only a mutex would prevent readers/writers from accessing same information at the same time. However if you use semaphores you would be able to make sure that there are no active writers or readers when someone is writing (condition synchronization) and also that no one is reading when there are active writers (condition synchronization).

Overall the main advantage to using semaphores is that is also gives condition synchronization along with mutual exclusion.

ii) Provide the definition of a semaphore by Dijkstra. The implementation based on this definition leads to the issue of “busy waiting”. What is this problem and why is it undesirable?

In computer science, a semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multiprogramming operating system. By Dijkstra's definition we are given operations P and V that correspond to wait() and signal(). Definitions given are that letter P is for *passeren* meaning "to pass" and letter V for *vrijgeven* meaning "to release".

Busy waiting means a process simply spins (does nothing but continue to test its entry condition) while it is waiting to enter its critical section. This continues to use (waste) CPU cycles, which is inefficient. In this time, a different process that doesn't need to wait could have ran its process so it is inefficient whenever a process takes computing power just to busy wait.

iii)Provide an implementation of the semaphore that avoids the busy wait.

To avoid busy wait, a process that has to wait on a semaphore should be blocked and put into a queue. When the semaphore allows new processes to run, they will be taken out of the queue in a first in first out order. This prevents busy waiting since the spot is opened up to a new process.
Therefore implementation of semaphore would be:

```
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 one process P from S->listl
   wakeup(P);
  }
}
```

In which places is now the waiting restricted?

Waiting is now restricted whenever a process needs to wait for a semaphore. It will now go into a queue block where it will wait its turn until it reaches the front of the line and has the semaphore signal for another process.

iv)

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


int shared;
sem_t roomAvailable;
sem_t serviceAvailable;
int k =3;

void make_transaction()
{
	// makes the transaction

	sleep(2);
	printf("Customer %u Transaction Made! \n",(int) pthread_self());
	sem_post(&serviceAvailable);

}

void take_a_walk()
{
	// makes the customer come back and try again
	printf("Customer %u took a walk! \n",(int) pthread_self());
	sleep(2);
}
void return_home()
{
	//removes the customer
	printf("Customer %u Removed! \n",(int) pthread_self());
	sem_post(&roomAvailable);

}
void bank_client()
{

	while(1)
	{
```

```
                printf("Hi I am thread %u \n", (int) pthread_self());

                int currentvalue;


                sem_getvalue(&roomAvailable, &currentvalue);

                printf("Current number of seats available %d\n",currentvalue);


                if(currentvalue != 0)
                {
                        sem_wait(&roomAvailable); // blocks if no room is available

                        sem_wait(&serviceAvailable);
                        printf("Customer %u making transaction \n", (int) pthread_self());
                        make_transaction();
                        break;

                }

                else
                {
                        take_a_walk();

                }

        }

        return_home();
}

int main(int argc, char **argv)
{
        sem_init(&roomAvailable,0,k);
        sem_init(&serviceAvailable,0,1);


        int NUM_THREADS = 5; // Create 15 customers

        pthread_t threads[NUM_THREADS];
        int rc;
```

```
        void *status;


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

        {
                //create threads
                rc = pthread_create(&threads[i], NULL, (void *)bank_client, NULL);

        }



        for(int j =0; j<NUM_THREADS; j++)
        {
                //join threads
                int rc = pthread_join(threads[j], &status);

        }


        return 0;

}
```

# Problem 4

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include <math.h>
#include <semaphore.h>
#include <stdbool.h>

sem_t numberOfTeacher;
sem_t numberOfChildren;
sem_t numberOfParent;

int currentTeacherValue;
int currentChildrenValue;
int currentParentValue;
int R = 3;
int currentTeachers;
int currentChildren;
int NUM_CHILD = 8;
int NUM_TEACHER = 3;
int NUM_PARENT = 3;

void teacher_enter()
{
        printf("Teacher %u Entered \n", (int) pthread_self());
        //sem_wait(&numberOfTeacher);
}
void child_enter(){
        printf("Child %u Entered \n", (int) pthread_self());
        sem_wait(&numberOfChildren);
}
void parent_enter(){
        printf("Parent %u Entered \n", (int) pthread_self());
        sem_wait(&numberOfParent);
}
void teach(){
        printf("Teacher %u is teaching \n",(int) pthread_self());
        sleep(2);
}
```

```
void learn(){
	printf("Child %u is learning \n", (int) pthread_self());
	sleep(2);

}
void verify_compliance(){
	printf("Parent %u checking compliance \n", (int) pthread_self());

	sem_getvalue(&numberOfChildren, &currentChildrenValue);
	sem_getvalue(&numberOfTeacher, &currentTeacherValue);

	currentTeachers=NUM_TEACHER-currentTeacherValue;
	currentChildren=NUM_CHILD-currentChildrenValue;

	printf("Number of children: %d \n", currentChildren);
	printf("Number of teachers: %d \n", currentTeachers);

	if(ceil((double)currentChildren / (double)R) <= currentTeachers)
	{
		printf("Regulation is complied with\n");
	}

	else
	{
		printf("Regulation NOT COMPLIED WITH \n");
	}

}
bool teacher_exit()
{

	printf("Teacher %u trying to leave \n", (int) pthread_self());

	sem_getvalue(&numberOfChildren, &currentChildrenValue);
	sem_getvalue(&numberOfTeacher, &currentTeacherValue);

	currentTeachers=NUM_TEACHER-currentTeacherValue;
	currentChildren=NUM_CHILD-currentChildrenValue;

	printf("Number of children: %d \n", currentChildren);
	printf("Number of teachers: %d \n", currentTeachers);

	if(ceil((double)currentChildren / (double)R) < currentTeachers)
```

```
        {
                printf("Teacher %u is allowed to leave\n", (int) pthread_self());
                sem_post(&numberOfTeacher);
                return true;
        }

        else
        {
                printf("Teacher %u goes back to classroom \n", (int) pthread_self());
                return false;
        }

}
void child_exit(){
        printf("Child %u is leaving \n",(int) pthread_self());
        sem_post(&numberOfChildren);
}
void parent_exit(){
        printf("Parent %u is exiting\n", (int) pthread_self());
        sem_post(&numberOfParent);
}


void Teacher() {
        printf("Teacher %u created\n", (int) pthread_self());
        sem_wait(&numberOfTeacher);
        for (;;)
        {
                sem_getvalue(&numberOfChildren, &currentChildrenValue);
                sem_getvalue(&numberOfTeacher, &currentTeacherValue);

                currentTeachers=NUM_TEACHER-currentTeacherValue;
                currentChildren=NUM_CHILD-currentChildrenValue;


                teacher_enter();
                sem_getvalue(&numberOfChildren, &currentChildrenValue);
                sem_getvalue(&numberOfTeacher, &currentTeacherValue);

                currentTeachers=NUM_TEACHER-currentTeacherValue;
                currentChildren=NUM_CHILD-currentChildrenValue;

                printf("Number of children from teacher func: %d \n", currentChildren);
```

```
		printf("Number of teachers from teacher func: %d \n", currentTeachers);

		//critical section
		teach();

		if (teacher_exit())
		{	printf("Teacher %u goes home\n", (int) pthread_self());
			break;
		}

		else
			continue;
	}
}

void Child(){
printf("Child %u created\n", (int) pthread_self());
	for (;;)
	{
		sem_getvalue(&numberOfChildren, &currentChildrenValue);
		sem_getvalue(&numberOfTeacher, &currentTeacherValue);

		currentTeachers=NUM_TEACHER-currentTeacherValue;
		currentChildren=NUM_CHILD-currentChildrenValue;
		//printf("before entering %d\n",currentChildren );


		child_enter();
		sem_getvalue(&numberOfChildren, &currentChildrenValue);
		sem_getvalue(&numberOfTeacher, &currentTeacherValue);

		currentTeachers=NUM_TEACHER-currentTeacherValue;
		currentChildren=NUM_CHILD-currentChildrenValue;
		printf("Number of children from children func: %d \n", currentChildren);
		printf("Number of teachers from children func: %d \n", currentTeachers);
		//critical section
		learn();
		child_exit();
		printf("Child %u goes home\n", (int) pthread_self());
		break;
	}
}
```

```
void Parent(){
printf("Parent %u created\n", (int) pthread_self());
	for (;;)
	{
		sem_getvalue(&numberOfChildren, &currentChildrenValue);
		sem_getvalue(&numberOfTeacher, &currentTeacherValue);
		sem_getvalue(&numberOfParent, &currentParentValue);

		parent_enter();
		//critical seciton
		verify_compliance();
		parent_exit();
		printf("Parent %u goes home\n", (int) pthread_self());
		break;
	}
}

int main(int argc, char *argv[]){

	void * status;

	pthread_t childThreads[NUM_CHILD];
	pthread_t teacherThreads[NUM_TEACHER];
	pthread_t parentThreads[NUM_PARENT];

	sem_init(&numberOfTeacher,0,NUM_TEACHER);
	sem_init(&numberOfChildren,0,NUM_CHILD);
	sem_init(&numberOfParent,0,NUM_PARENT);

	for (int i = 0; i < NUM_TEACHER; ++i)
	{
		pthread_create(&teacherThreads[i], NULL, (void *)Teacher, NULL);

	}
sleep(1);
	for (int i = 0; i < NUM_CHILD; ++i)
	{

		pthread_create(&childThreads[i], NULL, (void *)Child, NULL);

	}

sleep(1);
```

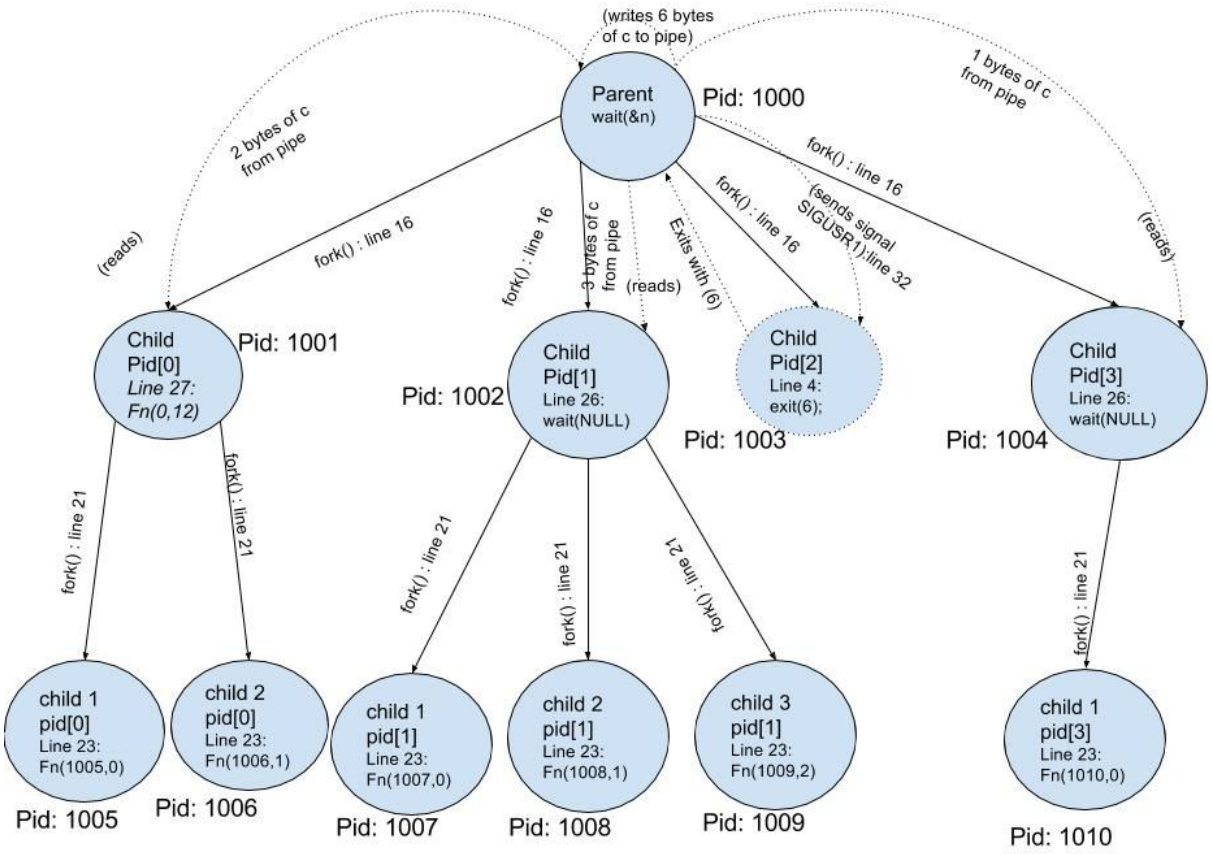
```
	for (int i = 0; i < NUM_PARENT; ++i)
	{
		pthread_create(&parentThreads[i], NULL, (void *)Parent, NULL);

	}


	for (int i = 0; i < NUM_TEACHER; ++i)
	{
		int rc = pthread_join(teacherThreads[i], &status);

	}

	for (int i = 0; i < NUM_PARENT; ++i)
	{
		int rc = pthread_join(parentThreads[i], &status);

	}

	for (int i = 0; i < NUM_CHILD; ++i)
	{
		int rc = pthread_join(childThreads[i], &status);

	}

	return 0;
}
```

# Problem 5

a)

1. The call kill(pid,SIGSTOP) calls a specific process “pid” and is used for debugging. When we debug the program, the operating system sends SIGSTOP to stop the program, for instance if it reaches a breakpoint. However, when SIGSTOP is sent to the operating system it stops the process for later resumption. The call signal(SIGINT, handler) sends the SIGINT signal number, which means the signal sent when someone presses CTRL-C in the terminal while application is running in the foreground, to the “handler” that is defined in the C program. In this case it performs an “exit(6)” and “Fn(0,-1)”.
2. The cnt in the call n = read(fd,buf,cnt) signifies the byte size of what is being read from the pipe. The values variable n can have after its return from read is the number of bytes that have been read from the pipe.
3. The call wp = wait(&status) will make the parent process wait till the child process has completed and wp will be the pid of the child process that has terminated and the status will be a flag for the completion of the child process. If wp == 1011 and WIFSIGNALED(status) ==1 and WTERMSIG(status) =3 that means process 1011 was terminated by a signal, rather than completion, and this was done by process 1000 by calling SIGQUIT.
4. If a process calls read() in an empty pipe when write end remains open for writing then it will block until data is available to read.

b)




The process tree is created based on the bytes written to the pipe. The parent process initially creates the 4 children processes. Then the 4 processes wait on line 18 to read from the pipe, but nothing is written to the pipe yet. The parent process sends the signal SIGUSR1 to the pid[i-2] where i =4, therefore pid[2] is sent the signal. It will use the handler defined at line 3, which makes the process exit with 6 (exit(6)). So this process is now exited. The parent process mean while was waiting for a process to exit, and now that pid[2] has exited, the parent gets the value returned from exit and that status is now in n. The parent process writes to the pipe with 6 bytes (WEXITSTATUS(n) = 6 ) of character c. Then on line 18, the first child pid[0] reads 2 bytes, so n=2 and then pid[0] forks 2 children processes. There are still 4 bytes left in the pipe. Then pid[1] reads 3 bytes from the pipe, n=3 and pid[1] forks 3 children processes. There is 1 byte left in the pipe. The pid[3] (pid[2] has exited) now wants to read 5 bytes from the pipe, but only 1 is available, so n=1 and pid[3] forks 1 child process. None of these processes, only exception being pid[2], return or exit therefore are in this steady state. To solve a potential race condition on line 18 assuming three process arrive at point marked /*A*/ we can have mutex that must be locked before entering this critical condition and is unlocked upon leaving this critical condition. Using a mutex, only one process can enter this critical section at a time.