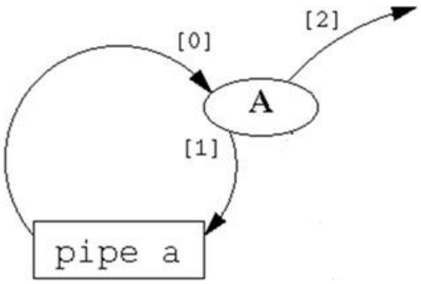
Operating Systems

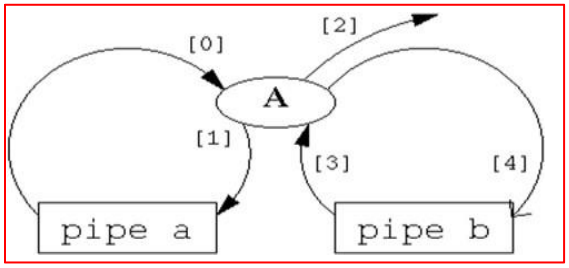
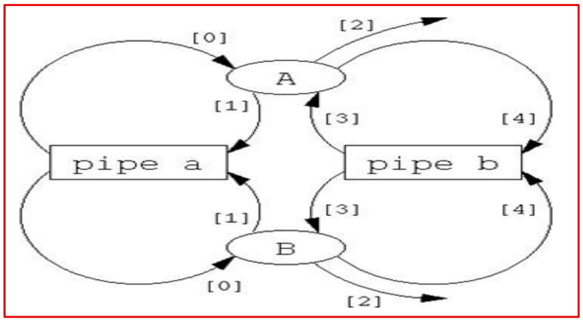
CS-2006

Question 1

Suppose the following code sections are executed without any error. Draw the diagrams showing the relationship between the file descriptors, pipes and processes (first one is given).

Code section	Diagram after the execution of the code section (A) is the name of parent process.

int fda[2], fdb[2];
pipe(fda);
dup2(fda[0], STDIN_FILENO);
dup2(fda[1], STDOUT_FILENO);
close(fda[0]);
close(fda[1]);
	

pipe(fdb);	
fork();	 [0] [2] A [1] [3] [4] pipe a pipe b [1] [3] [4] B [0] [2]

Note: In the diagram, [2] is Error fd, and the arrow shows that it is in its real state.

// dup2(int fd1, int fd2); closes fd2 and copies fd1 to fd2

// dup(int fd); copies fd to smallest available file descritop i.e., firstly 1 then 2 and so on…

Question 2

Assume that we have the following piece of code:

```c
int main() {
printf("Starting main\n");
int file_fd = open("test.txt", O_WRONLY | O_TRUNC | O_CREAT, 0666);
dup2(file_fd, STDOUT_FILENO);
pid_t child_pid = fork();
if (child_pid != 0) {
      wait(NULL);
      printf("In parent\n");
}
else {
      printf("In child\n");
}
printf("Ending main: %d\n", child_pid);
} // end of main()
```

What is the output of this program? You can assume that no syscalls fail and that *the child's PID is1234, and parent's PID is 4841.* Fill in the following table with your prediction of the output:

Standard out	test.txt
Starting main	*In child* *Ending main: 0* *In parent* *Ending main: 1234*

Next, assume we have altered the code as follows:

```c
int main() {
printf("Starting main\n");
int file_fd = open("test.txt", O_WRONLY | O_TRUNC | O_CREAT, 0666);
int new_fd = dup(STDOUT_FILENO);
dup2(file_fd, STDOUT_FILENO);
pid_t child_pid = fork();
if (child_pid != 0) {
      wait(NULL);
      printf("In parent\n");
}
else {
      dup2(new_fd, STDOUT_FILENO);
      printf("In child\n");
} // end of else block
printf("Ending main: %d\n", child_pid);
} // end of main()
```

What is the output this time? Fill in the following table with your prediction of the output:

Standard out	test.txt
Starting main *In child* *Ending main: 0*	*In parent* *Ending main: 1234*

Question 3

Consider the following program, executing under **normal conditions:**

```c
int main()
{
        int fd[2];
        pipe(fd);
        char buf[5];
        printf("FORK\n");
        if (fork()) {
                int rc = read(fd[0], buf, 4);
                switch (rc) {
                        case 4:
                                buf[4] = 0;
                                printf("%s\n", buf);
                                break;
                        case 0:
                                printf("READ-EOF\n");
                                break;
                        case -1:
                                printf("READ-ERROR\n");
                                break;
                }
        } // end of if block
        else {
                dup2(fd[1], STDOUT_FILENO);
                int rc = execv("/bin/echo",(char* []) {"echo", "ECHO", NULL});
                switch (rc) {
                case 0:
                        printf("EXEC-SUCCESS\n");
                        break;
                case -1:
                        printf("EXEC-ERROR\n");
                        break;
                }
        } // end of else block
        printf("DONE\n");
}
```

Which of the following options can be the possible output of the above code? **Select all possible outputs.**

❑	FORK ECHO EXEC-SUCCESS DONE DONE	❑	FORK FORK ECHO DONE DONE
❑	FORK READ-EOF DONE	☑	FORK ECHO DONE
❑	FORK FORK ECHO EXEC-SUCCESS DONE DONE	❑	FORK READ-ERROR EXEC-ERROR DONE DONE

Question 4

Consider the following C program, executing on Linux.

```
pid_t save_cpid;
int main()
 {
    pid_t cpid = fork();
    if (cpid < 0) {
       perror("fork"), exit(1);
    }
    save_cpid = cpid;
    if (cpid == 0) {
       printf("Child is %d and parent is %d\n", getpid(), getppid());
       save_cpid = save_cpid + 25;
       sleep(10);  // sleep for 10s
    }
    else {
       printf("Parent is %d and child is %d\n", getpid(), cpid);
       save_cpid = save_cpid - 25;
    }

    printf("%d sees save_cpid = %d\n", getpid(), save_cpid);
    if (cpid == 0) {
       exit(42);
    }
    int child_status;
    pid_t wpid = wait(&child_status);

    if (wpid == -1)
       printf("Could not wait for child yet.\n");
    else if (WIFEXITED(child_status))
       printf("Child %d terminated with exit status %d\n",
          wpid, WEXITSTATUS(child_status));  // part of else if
    else
       printf("Child %d terminated abnormally\n", wpid);
    return 0;
 }
```

When the above program was executed, lines 10 and 15 produced the following output:

```
                    Parent is 8386 and child is 8387
                    Child is 8387 and parent is 8386
```

(though not necessarily in this order).

(A): What output would have been produced by line 19?

```
        8386 sees save_cpid = 8362
        8387 sees save_cpid = 25
```

(in any order)

(B): Which, if any, of lines 27, 29 or 32 will be executed, and what would be output by the line(s)? If the answer cannot be uniquely determined in a concurrent environment, say why.

Line 29 will execute and output:

```
        Child 8387 terminated with exit status 42
```

The answer is unique in a concurrent environment – the fact that the child sleeps for 10s doesn't matter – wait() will block the parent process until the child exits on line 21. Conversely, it would work even if the child had already exited by the time the parent calls wait.

Question 5

Implementation of a UNIX Command:

The **/usr/bin/time** command can be used to determine how long a **Unix command** takes to execute as well as additional statistics about its execution. The command takes the name and command line arguments of an arbitrary program as input, and passes those command line arguments to the program it executes.

For instance, below are examples that show **how to time the execution of 'sleep 1.5' and 'echo Hello World'.**

```
$ /usr/bin/time sleep 1.5
  0.00user 0.00system 0:01.50elapsed

$ /usr/bin/time echo Hello World
  Hello World
  0.00user 0.00system 0:00.00elapsed
```

In the above example, the first command took 1.5 seconds to complete, as it is sleep 1.5, and in the second example it executed without any delay.

In this Question, you should implement a simplified version of time which only prints the real (wall clock) time elapsed during the execution of any command. **The output of your command shall be:**

```
$ ./time sleep 1.5
  Took 1 sec 502470 usec.

$ ./time echo Hello World
  Hello World
  Took 0 sec 960 usec.
```

Complete the implementation of time.c Program, carefully analyze the execution command of executable file of time.c, and implement your logic accordingly.

```
int main(int ac, char* av[]) {

    struct timeval start, end, diff;
    gettimeofday(&start, NULL);
    /* Complete this part, for simplicity no need to show error checking */
    if (    !fork()    ) {
        execvp(av[1], av + 1);
    }
    wait(NULL);
    gettimeofday(&end, NULL);
    timersub(&end, &start, &diff);
    printf("Took %ld sec %ld usec.\n", diff.tv_sec, diff.tv_usec);
    return 0;
} // end of main()
```

Question 6

Implementation of pipe Command:

You are asked to write a program (say prog.c) that forks and runs a sub-shell as a child process, and simply counts the number of output characters that the shell **printed on the standard output.** When the sub-shell terminates, the parent, simply reports the number of output characters produced by the sub-shell.

Some part of the prog.c is already implemented for you, you can ignore most of the error checking to make your solution clear, **but you need to close all necessary file or pipes and check what read-write etc.,** returns. Also read/write one char at a time to make counting job easy!

```
/* Implementation of prog.c  */
/* There may be some extra lines given before while loop, you may leave them */
/* Only implement the required functionality on each blank block of lines */
/* Pipes and file descriptors must be set acc. To above req. statement */

int main(int ac, char* av[]) {
    int pfd[2];
    int childpid, numread, numwritem, count=0;
    char buf;
    /* Setup the pipe and child process below */
    pipe(pfd);
    childpid = fork();

    /* start child execution below */
    if (   childpid == 0   ) {
    /* child sets up its pipe below */
        dup2(pfd[1], STDOUT_FILENO);
        close(pfd[0]);
        close(pfd[1]);

        execl("/bin/bash","shell",NULL);
        perror("cannot start shell");
        return 1;
    } // end of if statement

    /* parent sets up the pipe below */
    dup2(pfd[0], STDIN_FILENO);
    close(pfd[0]);
    close(pfd[1]);
    ____________________________________;
    /* parent reads from pipe as long as there is data */
    /* counts the characters and puts them on standard output */
    while (  true   ) {
        numread = read(STDIN_FILENO, &buf, 1);
        if (  numread <= 0   ) break;
        count++;
        numwrite = write(STDOUT_FILENO, &buf, 1);
        if (  numwrite <= 0   ) break;
    }// end of while
    fprintf(stderr, Shell printed %d characters.\n, count);
    /* process terminates (1 important thing to do before this) */
    wait(NULL);
}// end of main()
```

Question 7

Exec and Unnamed Pipes:

As we know that there are two types of pipes, Unnamed and Named. Unnamed Pipes are used for communication between related processes. Now suppose we have two C programs given below:

Prog.c

```
int main()
{
      int pfd[2];
      pipe(pfd);
      pid_t id = fork();

      if (id == 0)
      execl("/home/aneeq/Desktop/OS/testExec",
                  "./testExec", NULL);

      wait(NULL); // reap the child

      // close the write end
      close(pfd[1]);
      char buf[17];

      read(pfd[0], buf, 17);
      printf("%s", buf);
}
```

testExec.c

```
int main()
{

      close(3);
      char buf[] = "Hello from Child";
      write(4, buf, sizeof(buf));

}
```

In the above example, we are communicating between two different programs using unnamed pipes, because they both are being executed by related processes (parent-child). But, if you notice we don't have the access of pipe variable **`int pfd[2]`** in testExec.c, and we have to manually put the descriptor. Examining this scenario answer the following Questions:

(A): Will both programs communicate effectively, i.e., Will **"Hello from Child"** be printed in Prog.c?

Yes, both will communicate bcz they have a child-parent relation and a call to exec Doesn't affect the file descriptors

(B): How can we access the variable **`int pfd[2]`** in testExec.c file? Make all necessary changes in the code for this implementation. After successful implementation we should be able to do

```
close(pfd[0]);
write(pfd[1], buf, sizeof(buf));
```

in testExec.c, rather than to use 3 & 4.

We will send the file descriptors as args with the exec call then on the other end we Will receive the args and make the array int pfd[2] with those values

(C): Modify the program as such that we take **No. of pipes** input from user. Set up all those pipes in Prog.c, and access all those pipes in testExec.c, also basic implementation of above program must stay same, i.e., closing write ends of all pipes in Prog.c and read ends of all pipes in testExec.c and finally read and write from all pipes as same as the above program.

We will send the file descriptors as args with the exec call same as above, but we will Utilize a for loop to make args to send over and on the receiving end we will form the Array by using a loop like the following:

```
char *str = (char *)malloc(Npipe * 2 + 1);
str[0] = Npipe + '0'; // No of pipes
int **pfd = (int **)malloc(Npipe);
for (int i = 0, j = 1; i < Npipe; i++, j += 2 ) {
      pfd[i] = (int *)malloc(2);
      pipe(pfd[i]);
      // now store these into str[j] and str[j+1]
}
pid_t id = fork();
char *arg[] = {"./testExec", str};
if (id == 0)
      execv("/home/aneeq/Desktop/OS/testExec", arg);
```

Question 8

Shells use the UNIX system call pipe( ) to connect a program's standard output to another program's standard input. Following is an incomplete example program that shows how redirection works. It implements a program "spipe.c", which starts two child processes whose standard input/outputs connected through a pipe.

In other words, `./spipe program1 program2` is equivalent to running `program1 | program2` in a shell that supports pipes:

```
$ ./spipe ls sort
  // displays the list in sorted way

$ ls | sort
  // displays the list in sorted way
```

For simplicity, spipe does not support passing on any arguments to thecommands it launches. For presentation purposes, error handling was omitted as well. Complete the program given:

```
/* Implementation of spipe.c */
/* Following function is already implemented for you: */
/* int launch_child(char* prgname, char* prgargv[], int stdout, int stdin,
                                                 int fdtoclose ) */
/* Explanation of the above stated function: */
/* Launches a child process, redirect its standard out to stdout, standard in to
   stdin, Also close ‘fdtoclose’ in child. Close any redirected file descriptors
   In parent before returning. Returns pid of child process */

int main(int ac, char* av[]) {

    char* firstprogargv[] = { av[1], NULL };
    char* secondprogargv[] = { av[2], NULL };
    /* Setup the pipe below */
    int pfd[2];
    pipe(pfd);

    /* launch first child */
    pid_t leftchild = launch_child(av[1], firstprogargv, pfd[1], 0, pfd[0]);
    /* reap the first child below use waitpid */
    waitpid(leftchild, NULL, 0);
    /* launch second child */
    pid_t rightchild = launch_child(av[2],secondprogargv, 1, pfd[0], pfd[1]);
    /* reap the first second below use waitpid */
    waitpid(rightchild, NULL, 0);
 }
```

Users observed the following behavior of “spipe”. For many combinations of programs, “spipe” appears to work as intended. For some (say prog2.c, implementation hidden) “it gets stuck”, then the ps command shows:

```
$ ps f
   PID TTY      STAT      TIME COMMAND
 24889 pts/9    Ss        0:00 -bash
 31462 pts/9    S         0:00 \_ ./spipe ./prog2 wc
 31463 pts/9    S         0:00 |     \_ ./prog2
 31490 pts/9    R+        0:00 \_ ps f
```

Based on this information, answer the following Questions:

(A) Does this apparent bug of “spipe” cause excessive CPU usage on the machine on which user runs it? Justify your answer!

No, all processes involved are in the BLOCKED state (Linux calls this 'S' for Sleeping as can be seen in the output of ps), they do not use any CPU.

(B) Why does 'spipe' not finish? Be precise!

Pipes have finite capacity; when a process attempts to write to a full pipe, it is blocked. Here, prog2 filled the pipe, then blocked when attempting further writes. The process that consumes data from the pipe (wc in this case) hasn't been started because the parent spipe program is waiting for prog2 to finish first. A deadlock results.

FYI, here's the prog2.c, which writes exactly 64KB + 1 bytes.

```
#include <stdio.h>
int main()
{
        int i;
        for (i = 0; i < 65537; i++)
                fputc('A', stdout);
        return 0;
}
```

(C) Suggest a way to fix spipe!

Don't wait for the first child until after both children have been forked; i.e. move the 'waitpid(leftchild, NULL, 0);' call after the second call to launch_child( ). This will allow the children to run concurrently so that wc can drain the pipe whenever prog2 writes into it.

Question 9

Suppose that the file "tmpdata.txt" contains "abcdefghijk". If the following code is executed correctly without generating any errors, then:

```
int main(int ac, char* av[]) {

    int fd;
    char buf[6] = "12345";
    fd = open("tmpdata.txt", O_RDONLY);
    fork();
    read(fd, buf, 2);
    read(fd, buf + 2, 2);
    printf("%d: %s\n", getpid(), buf);
} // end of main()
```

(A) Explain if the following two outputs are possible or not? Why/why not?
Suppose parent's pid is 7 while child's pid is 8.

7:abcd5 8:abef5	Not possible, since system file table entry and thusoffset is shared, they need to read different char!
7:abef5 8:cdgh5	Yes, possible, since system file table entry and thus offset is shared, each reads 1 or 2 char and advance theoffset. accordingly the other reads the next one.

(B) What could be the outputs if the lines 5 & 6 are exchanged? Write at least 3 possible outputs.

```
7:abcd5     7:a2bc5     7:abc45
8:abcd5     8:a2b45     7:a2bc5 ....
```

Question 10

Suppose the following C code is executed; assume appropriate wait( ) calls occur in the missing part of the code. Theexecutables progA, progB, progC and progD are installed in directories in the system path, and each simply prints a message indicating the pid of the process executing it and then terminates.

```
/* implementation of fork02.c */
int main(int ac, char* av[]) {

    char* argv1[] = {“progA”, NULL};
    char* argv2[] = {“progB”, NULL};
    char* env[] = {NULL};

    printf(“%d is entering fork02.\n”, getpid());
    int index = 0;
    pid_t pid = fork();
    pid = fork();
    if ( pid == 0 && index == 0) {
       index++;
       pid = fork();
       if ( pid == 0 ) {
            execve(“progA”, NULL, NULL);
       } // end of nested if

       if ( pid != 0 && index = 0 ) {
            pid = fork();
            execve(“progB”, NULL, NULL);
       } // end of nested if
    } // end of if
    /* Assume Necessary wait calls executed here */
    printf(“%d is exiting from fork02.\n”, getpid());
    return 0;
} // end of main()
```

(A) Which program is the original process executing when that original process terminates?

progB - it enters the second if statement and exec's.

(B) How many processes terminate in each of the relevant programs?

Program	# of processes that terminate there
fork02	2
progA	2
progB	4

Question 11

Consider the following example programs. List all possible outputs.

// included in all programs
#include <unistd.h>
#include <sys/wait.h>
// W(A) means write(1, "A", sizeof "A")
#define W(x) write(1, #x, sizeof #x)
	Possible Outputs:

int main() {
 	
    W(A);
    fork();
    W(B);
    fork();
    W(C);
} // end of main()
	ABBCCCC
	ABCBCCC
	ABCCBCC

int main() {
 	
    W(A);
    int child = fork();
    W(B);
    if(child)
        wait(NULL);
    W(C);
} // end of main()
	ABCBC
	ABBCC

int main() {
 	
    W(A);
    if(!fork() && !fork())
        W(B);
    else {
        W(A);
        exit(0);
    } // end of else
    W(C);
    wait(NULL);
} // end of main()
	ABAAC
	AABAC
	AAABC
	ABCAA
	AABCA

Question 12

What output do the following 2 programs produce & why?

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

int counter;

static void * thread_func(void * _tn) {
   int i;
   for (i = 0; i < 100000; i++)
        counter++;

   return NULL;
} // end of function

int main()
{
   int i, N = 5;
   pthread_t t[N];

   for (i = 0; i < N; i++)
       pthread_create(&t[i], NULL,
                 thread_func, NULL);

   for (i = 0; i < N; i++)
       pthread_join(t[i], NULL);

   printf("%d\n", counter);
   return 0;
}
```

Output:

The output is not deterministic - 5 threads are accessing a shared but unprotected variable concurrently. A blatant race condition.

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

int counter;

static void * thread_func(void * _tn) {
   int i;
   for (i = 0; i < 100000; i++)
        counter++;

   return NULL;
} // end of function

int main()
{
   int i, N = 5;
   pthread_t t[N];

   for (i = 0; i < N; i++) {
       pthread_create(&t[i], NULL,
                 thread_func, NULL);

       pthread_join(t[i], NULL);
   } // end of for loop

   printf("%d\n", counter);
   return 0;
}
```

Output:

The output is 500000. In this case, the 5 thread do not execute concurrently, but in sequence. Each thread is started only after the previous one has exited (and was joined by the main thread.)

Question 13

Consider the following 2 programs and predict their output.

```
# define size 6
int main()
{
    int pfd[2];
    pipe(pfd);

    if(!fork())
    {
```

```
# define size 6
int main()
{
    int pfd[2];
    pipe(pfd);

    if(!fork())
    {
        close(pfd[0]);
        char buf[size] = "Ooops";
        write(pfd[1], buf,sizeof(buf));
        write(pfd[1], buf, 3);
    } // end of if
    else{
        close(pfd[1]);
        char buf[size * 2];
        read(pfd[0], buf, size + 1);
        printf("%s\n", buf);
        char buf2[size];
        read(pfd[0], buf2, 3);
        printf("%s", buf2);
    } // end of else
} // end of main()
```

Output:

```
Ooops
oo
```

```
        close(pfd[0]);
        char buf[size] = "Ooops";
        write(pfd[1], buf,sizeof(buf));
        write(pfd[1], buf, 3);
    } // end of if
    else{
        close(pfd[1]);
        char buf[size * 2];
        read(pfd[0], buf, size + 5);
        printf("%s", buf);
    } // end of else
} // end of main()
```

Output:

```
Ooops
```

Question 14

Consider the following code snippets, identify No. of Zombie & Orphan processes, when control of parent process reaches the sleep( ).

```
int main()
{
   if(!fork())
   {
        printf("Hello from child 1");
        wait(NULL);
        exit(0);
   }
   else
   {
        if(!fork())
        {
            wait(NULL);
            exit(0);
        }
   }
   sleep(10);
}
```

```
int main()
{
   if(!fork())
   {
        if(!fork())
        {
            exit(0); // exit-1
        }
        exit(0); // exit-2
   }
   else
   {
        printf("Hello from parent");
   }
   sleep(10);
}
```

No. of Zombies: **2**	No. of Zombies: **Max 2, Min 1**
No. of Orphans: **Max 2, Min 0**	No. of Orphans: **Max 2, Min 0**

Question 15

Dividing the Work:

Consider the following incomplete program **(Max.c)** which finds the maximum value in an array, but NOT with a single process. It takes multiples processes and divides the array between them, and each finds the maximum in their array and writes to pipe.

Complete the code given below:

```
/* Implementation of Max.c */

int max(int *arr, int start, int end) {
    int maxVal = arr[start];
    for (int i = start + 1; i < end; i++)
        if(arr[i] > maxVal)  maxVal = arr[i];

    return maxVal;
} // end of function

int main() {
    srand(time(0));
    int size, Np; // No. of processes
    printf(“Enter size of array: “);
    scanf(“%d”, &size);

    int *arr = (int *)malloc(sizeof(int) * size);
    printf(“Array: ”);
    for (int i = 0; i < end; i++) {
        arr[i] = rand() % 100 + 1;
        printf(“%d ”, arr[i]);
    } // end of for loop

    printf(“\nEnter No. of processes: ”);
    scanf(“%d”, &Np);

    int sizePerProcess = size / Np, start = 0;

    /* set up the pipe below */
    int pfd[2];
    pipe(pfd);

    for (int i = 0; i < Np; i++) {
        if (  !fork()  ) { // child process should execute
            close(pfd[0]);
34:             int m = max(arr, start, start + sizePerProcess);
35:             write(pfd[1], &m, sizeof(m));
36:             exit(0);
37:         } // end of if
38:         start += sizePerProcess;
39:     } // end of for loop
40:     close(pfd[1]);  // close pipe for parent
41:     /* reap all the zombie processes below */
42:     for (int i = 0; i < Np; i++)
43:         wait(NULL);
44:
45:     int m, p=1;
46:     read(pfd[0], &m, sizeof(m)); // read first value written to pipe
47:
48:     for (int i = 0; i < Np - 1; i++) {
49:         int newM;
50:         read(pfd[0], &newM, sizeof(newM));
51:         if (newM > m) {
52:             m = newM;
53:             p = i + 2;
54:         }
55:     } // end of for loop
56:     printf("\nMax of array: %d found by Process No. %d", m, p);
57: } // end of main()
```

Now as you can analyze, how we divided the work between the processes. Answer the following Questions by analyzing the completed code:

(A) Is this calculation of max, running in Parallel or Serial?

It is running in Parallel.

(B) If answer to above is serial, then convert to Parallel and vice versa.

To convert to Serial don't call wait in separate loop, rather call wait after line 38 inside loop.

(C) Convert the above code, by using threads instead of processes.

Convert it Yourself!!, just have to use pthread_create instead of fork and pthread_join instead of wait

Question 16

Suppose the following C program is executed, and the executable alpha and beta exist in the current directory

```
void rptNewChild(pid_t pid, char* src)
{
	if (pid != 0) {
		printf("%d forked %d at %s\n", getpid(), pid, src);
	}
}
```

```
void chill(pid_t pid)
{
        pid_t cpid;
        printf("%d is chilling\n", pid);
        while ((cpid = wait(NULL)) > 0) {
                printf("%d waited for %d\n", pid, cpid);
        }
}


int main() {
        char* av[] = { NULL };
        char* env[] = { NULL };

        printf("%d is the original process.\n", getpid());
        pid_t pid = fork();  // F1 fork
        rptNewChild(pid, "F1");

        if (pid != 0) {
                pid = fork();
                rptNewChild(pid, "F2");
                if (pid == 0) {
                        pid = fork();
                        rptNewChild(pid, "F3");
                        chill(getpid());
                        execve("./alpha", av, env);
                }
        }
        else {
                chill(getpid());
                execve("./beta", av, env);
                pid = fork();
                rptNewChild(pid, "F4");
        }
        chill(getpid());

        return 0;
}
```

Alpha.c	Beta.c

int main(int argc, char** argv) {
 	
    printf("%d running alpha\n", getpid());
    sleep(5);
    return 0;
}
	int main(int argc, char** argv) {
	
	    printf("%d running beta\n", getpid());
	    sleep(5);
	    return 0;
	}

When the program was executed, the various calls to `rptNewChild( )` produced the following output (and possibly more that is not shown here):

```
20317 is the original process.
```

```
20317 forked 20318 at F1

20317 forked 20319 at F2

20319 forked 20320 at F3
```

Based on this information answer the following Questions:

Questions	Answers
State the PIDs of the processes, if any, that will eventually execute the program alpha.	**20319, 20320**
State the PIDs of the processes, if any, that will eventually execute the program beta.	**20318**
Will a child be forked at the `fork( )` call labelled F4? **<u>If so,</u>** state the PID of parent that will fork that child, and assume the child's PID is 20321. **<u>If not,</u>** explain why no child will be forked there.	**No. Any process that enters that else clause will execve() before reaching that fork() call, and calls to execve() do not return unless they fail. Given that the programs alpha and betaare said to be in the current directory, there is no reason for the execve() calls to fail.**
State the PIDs of the processes, if any, that will eventually call `chill( )`.	**20317, 20318, 20319, 20320**
Determine which processes will eventually produce output from `printf( )` call inside while loop in function `chill( )`. For each of those processes, write output that would be produced.	**20317:** **20317 waited for 20318** **20317 waited for 20319** **20319:** **20317 waited for 20320**
Is it possible to determine which process will terminate last? **<u>If so,</u>** state the PID of that process & explain why it cannot terminate until all the others have done so. **<u>If not,</u>** list two different orders in which the processes could terminate, showing a different process terminating last.	**Yes. 20319 must wait for 20320 to terminate, and 20317 must wait for 20318 and 20319 to both terminate; therefore, 20317 must be the last to terminate.**

Question 17

Consider the following Programs and Process Trees.

Program A	Program B	Program C	Program D

`int main() {`
 	
`   if (fork())`
`       fork();`
`   sleep(1000);`
`}`
	`int main() {`
	
	`   if (!fork())`
	`       fork();`
	`   sleep(1000);`
	`}`
		`int main() {`
		
		`   if (fork())`
		`      if(fork())`
		`         fork();`
		`   sleep(1000);`
		`}`
			`int main() {`
			
			`   if (fork())`
			`       fork();`
			`   else`
			`       fork();`
			`   sleep(1000);`
			`}`

```
Tree 1                   | Tree 2                 | Tree 3                 | Tree 4
PID    CMD               | PID    CMD             | PID    CMD             | PID    CMD
10252  -bash             |                        | 10252  -bash           |
10316  \_ ./ft           | 10252  -bash           | 10836  \_ ./ft         | 10252  -bash
10317      \_ ./ft       | 10304  \_ ./ft         | 10837      \_ ./ft     | 10694  \_ ./ft
10319      |   \_ ./ft   | 10305      \_ ./ft     | 10838      \_ ./ft     | 10695      \_ ./ft
10318      \_ ./ft       | 10306      \_ ./ft     | 10839      \_ ./ft     | 10696          \_ ./ft
```

Each of the four Programs (A through D) produced one of the four process tree diagrams (1 through 4) when started in the background. Which Program produced which tree?

Programs	A	B	C	D
Trees	2	4	3	1

Question 18

Consider the following program

```
const int READ_END = 0;
const int WRITE_END = 1;

static pid_t run_process(char* exe, char* arg1, int std_in, int std_out)
{
     pid_t  child = fork();
     if (child == 0) {     // fork child process
          char* argv[] = { exe, arg1, NULL };
          if (std_in != -1)     // redirect stdin
               dup2(std_in, STDIN_FILENO);
          if (std_out != -1)    // redirect stdout
               dup2(std_out, STDOUT_FILENO);
          execvp(exe, argv);
     }
     waitpid(child, NULL, 0);  //  wait  for  child
     return child;
}

int main(int ac, char* av[])
{
     int fd[2];
     pipe2(fd, O_CLOEXEC);
      // same as pipe, just closes the pipe when process calls exec

      run_process("cat", av[1], -1, fd[WRITE_END]);
     close(fd[WRITE_END]); // close parent's write end

      run_process("wc", "-m", fd[READ_END], -1);
     close(fd[READ_END]);  // close parent's read end
}
```

This program is complied with `gcc -Wall -o piping piping.c.` When run like so:

```
$ ./piping piping.c
909
```

The program outputs a result, <u>the number of characters in the source file.</u>

However, when run like so:

```
$ ./piping  /home/aneeq/Desktop/OS/words
```

<u>The program appears not to finish, it gets **"stuck"**</u>

(A) Explain why this program finished when run in first way but didn't finish when run the second way.

The first time, the amount of data the cat program wrote into the pipe was smaller than the pipe's internal buffer, allowing the write to complete and cat to exit. Thus, the piping program would return from its waitpid call and fork and wait for the second process running wc.

However, when faced with a large file, cat was unable to write the full amount of data into the pipe since it exceeded the size of the pipe buffer. At this point, the process entered the BLOCKED state. Since the piping program waited for the first child process before spawning the second second, it remained in the BLOCKED state as well, effectively causing deadlock.

(B) Repair the program so that it completes successfully, i.e.,

```
$ ./piping  /home/aneeq/Desktop/OS/words
4953680
```

<u>Note:</u> You should not write a program that performs the functionality of counting the number of characters in a file, rather you should make the minimum amount of changes to the given program to make it complete and perform its functionality in the intended way. The repaired program must send its data through a pipe. This can be accomplished by rearranging the system calls made by the program without adding new ones or removing any.

The solution is that remove the waitpid call inside the function, and catch the id of child in main process; that the function returns. After both function calls, write waitpid calls with child ids catched from function calls. Now both processes are running concurrently and pipe is being drained as it is being filled.

Question 19

Reading from file Twice:

Your friend tells you that they can open a file once but read it twice. Although you are skeptical, you decide to give it a try. Without utilizing another call to open, finish the following `double_read()` function. This function should read the specified file twice, storing the appended result as a duplicated, null-terminated string in the given buffer ‘buffer’. Note: You must actually `read( )` the file contents twice – do not just copy the data after reading the file once!

In the following, assume that all system calls succeed and that the buffer is big enough to hold the result. Read the file in a maximum of CHUNK_SIZE bytes at a time. While you do not necessarily need to use every line here, you are also not allowed to add semicolons to existing lines.

```
# define CHUNK_SIZE 1024

void double_read(char* filename, char* buffer) {
    int fd = open(filename, O_RDONLY);
    int offset = 0;
    int bytesread;

    for (int i = 0; i < 2; i++) {
        while(true) {
             bytesread = read(fd, buffer + offset, CHUNK_SIZE);
             if (  bytesread <= 0  ) break;
             offset += bytesread;
        } // end of while
        lseek(fd, 0, SEEK_SET);
    } // end of for
    *(buffer + offset) = ‘\0’;
    close(fd);
} // end of function
```

Question 20

Indicate the output produced by running the following code.

```
void* helper(void* args) {
    int *fd = (int *)args;
    printf(“Printing from thread helper\n”);
    dup2(*fd, STDOUT_FILENO);
    return NULL;
}

int main() {
    printf(“Starting main\n”);
    int file_fd = open(“test.txt”, O_WRONLY | O_CREAT | O_TRUNC, 0666);
    int* temp_fd = (int *)malloc(sizeof(int));
    *temp_fd = dup(STDOUT_FILENO);
13:     pthread_t thread;
14:     dup2(file_fd, STDOUT_FILENO);
15:     pthread_create(&thread, NULL, &helper, temp_fd);
16:     pthread_join(thread, NULL);
17:     printf(“Ending main\n”);
18:     return 0;
19: }
```

Standard out	test.txt
Starting main Ending main	Printing from thread helper

Now, suppose we interchange line # 14 and 15. **Now predict the output.**

Standard out	test.txt
Starting main Printing from thread helper	Ending main

Question 21

Provide **3 possible outputs** of the following code. Assume all the calls i.e., pthread_create( ) and pthread_join( ) successfully execute.

```
void subtask(void* args_) {
      int idx = (int)args_;
      printf("%d ", idx);
      return;
}

int main(void) {
      pthread_t threads[6];
      for (int i = 1; i <= 6; i++)
      {
            pthread_create(threads[i - 1], NULL, subtask, (void*)i);
            if (i % 3 == 0) {
                  for (int j = i - 2; j <= i; j++)
```

```
                pthread_join(threads[j - 1], NULL);
        }
    }
    return 0;
}
```

Output No. 1 (Any possible)	123456
Output No. 2 (Any possible)	231654
Output No. 3 (Any possible)	312546

Question 22

To test the functionality of multithreading you write the following code:

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

#define WORMS 8

typedef struct {
    pthread_t tid;
    char* msg;
} pthread_args;

static void* spawn_worm(void* arg) {
    pthread_args* args = (pthread_args*)arg;
    char msg[100];
    // copies formatted string to MSG
    sprintf(msg, "Worm #%ld", args->tid);
    args->msg = msg;
}

int main() {
    pthread_args args;
    int s;
    for (int i = 0; i < WORMS; i++) {
        s = pthread_create(&args.tid, NULL, &spawn_worm, &args);
        if (s != 0) {
            return 1;
        }
    }
    for (int i = 0; i < WORMS; i++) {
        s = pthread_join(args.tid, NULL);
        if (s == 0) {
            printf("%s\n", args.msg);
        }
    }
    return 0;
}
```

You run the program assuming the following:

- Newly spawned threads start at ID 1 and increment by 1 for each additional thread.
- Your machine specification is good enough to not fail any memory allocations.

You expect that this program prints out all of the worm numbers from **1 to 8** in sequential order like:

```
| OS2006@FAST ~$ gcc silkworm.c -o silkworm -lpthread
| OS2006@FAST ~$ ./ silkworm
| Worm 1
| Worm 2
| Worm 3
| Worm 4
| Worm 5
| Worm 6
| Worm 7
| Worm 8
| OS2006@FAST ~$
```

However, upon compiling and running this program, you noticed that your output seems **"bugged";** you get the following output:

```
| OS2006@FAST ~$ gcc silkworm.c -o silkworm -lpthread
| OS2006@FAST ~$ ./ silkworm
|
| OS2006@FAST ~$
```

Identify which lines of code **(5 lines)** should be replaced and their replacements such that the program will work as expected. You are **NOT** allowed to delete or add lines.

Old line	New Replaced Line
`char msg[100];`	`char* msg = (char*) malloc(100 * sizeof(char));`
`pthread_args args;`	`pthread_t tid[WORMS];`
`s = pthread_create(&args.tid, NULL, &spawn_worm, &args);`	`s = pthread_create(&tid[i], NULL, &spawn_worm, &tid[i]);`
`s = pthread_join(args.tid, NULL);`	`s = pthread_join(tid[i], &worm_msg);`

This program has 2 issues:

Memory is allocated on the stack for the string, which is why we see an empty line being outputted.

Each thread reuses the same tid, so the first pthread_join will block until tid 8 finishes. The remainder of the pthread_join will fail.

Question 23

Parallel File Copy:

You are making a backup copy of your entire hard drive, but you find that copying a single file at a time takes too long. You decide to put your OS knowledge to good use, and copy multiple files in parallel.

Your program takes in a non-negative integer n and two lists of strings, both of length n. The first list, **in_files,** contains a list of source files; the second list, **out_files,** contains the corresponding destination file names. Write a program to copy each source file to the corresponding destination. You must create a separate process to copy each file. The files should all be copied concurrently.

The main (parent) process must wait for all the files to be copied, and all child processes to exit, before exiting.

Fill in the blanks below to implement your parallel file copy program. You may assume calls to read and write never error. Do NOT assume they read/write the full amount.

NOTE: You should only use **AT MOST** one line per blank. Extra lines are NOT allowed. Your solution does not need to use all the blanks. You may use loops, conditional statements, and other control structures in your solution as you see fit.

```
#define BUFSIZE 2048
void copy_file(char* src, char* dst) {
    char buf[BUFSIZE];

    int srcfd = open(src, O_RDONLY);
    int dstfd = open(dst, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    int chunk, total;

    while ( (chunk = read(srcfd, buf, BUFSIZE)) > 0 ) {

       total = 0;
       while ( total < chunk ) {
             total += write(dstfd, &buf[total], chunk - total);
       }

    } // end of while

    close(srcfd);
    close(dstfd);
} // end of copy_file()

void copy(int n, char** in_files, char** out_files) {
    pid_t children[n];
    int i;

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

       pid_t child = fork();

       if ( child > 0 )
             children[i] = child;
       else if ( child == 0 ) {
             copy_file(infiles[i], outfiles[i]);
             exit(0);
       }
       else
             perror(“fork failed”);
    } // end of for loop

    for (int i = 0; i < n; i++)
    {
         waitpid(children[i], NULL, 0);
    }


} // end of copy()

int main() {
    // Example usage
    int n = 2;
    char* in_files[] = {“test1.txt”, “test2.txt”};
    char* out_files[] = {“copy1.txt”, “copy2.txt”};
    // test1.txt should be copied to copy1.txt
    // test2.txt should be copied to copy2.txt
    copy(n, in_files, out_files);
    return 0;
} // end of main()
```

Question 24

Santa’s List

Santa is digitizing his accounting department and would like to track naughty and nice children on his computer. He has a file for each person in the form of `childX.txt (e.g. child0.txt)` that contains string records representing the good and bad deeds someone has done all year.

For example,

child0.txt

GOOD attended live lecture and asks questions.

GOOD finished assignments in a timely manner.

BAD didn't study for midterm.

child1.txt

BAD didn't attend any discussion.

BAD didn't help his teammates during projects.

child2.txt

GOOD answered a lot of questions on Piazza.

BAD didn't fill out course evaluations.

Fill in the code below that determines whether a child has performed more good deeds than bad deeds. As there are many children, **create a new thread for each file for concurrency.** When all threads are done counting all files, the program should write the filenames with positive karma to nice.txt and others to naughty.txt. Separate each filename by a newline with a trailing newline at the end of the file.

```
typedef struct thread_data {
    int tid;
    pthread_t thread;
    char filename[128];
    int karma;
} thread_data_t;

void* count_good_bad(void* threadarg) {
    thread_data_t* tdata = (thread_data_t*)threadarg;
    FILE* fp = fopen(tdata->filename, “r”);
    char* line = NULL;
    size_t len = 0;
    ssize_t read;
    while ((read = getline(&line, &len, fp)) != -1) {
       if (strncmp(line, “GOOD”, 4) == 0)
             tdata->karma++;
       else if (strncmp(line, “BAD”, 3) == 0)
             tdata->karma--;
    } // end of while
    pthread_exit(NULL);
}

int main(int argc, char* argv[]) {
    int num_children = atoi(argv[1]);
    thread_data_t thread_data_array[num_children];
    for (int i=0; i<num_children; i++) {
       thread_data_array[i].tid = I;
       sprint(thread_data_array[i].filename, “child%d.txt”, i);
       thread_data_array[i].karma = 0;
       pthread_create(&thread_data_array[i].thread, NULL, count_good_bad,
                      (void*)&thread_data_array[i]);
    }

    for (int i=0; i<num_children; i++)
       pthread_join(thread_data_array[i].thread, NULL);

    FILE* fp_nice = fopen(“nice.txt”, “w”);
    FILE* fp_naughty = fopen(“naughty.txt”, “w”);

    for (int i=0; i < num_children; i++) {
      if ( thread_data_array[i].karma >= 0 )
          fprintf(fp_nice, “%s\n”, thread_data_array[i].filename);
      else
          fprintf(fp_naughty, “%s\n”, thread_data_array[i].filename);
    }
    fclose(fp_nice);
    fclose(fp_naughty);
    return 0;
}
```

With the above example, the files would look like:

nice.txt

child0.txt
child2.txt

naughty.txt

child1.txt

Also, answer the following Questions:

(A) What would happen if we did not call pthread_join( ) ?

The program would write the results of tabulation before all threads complete. However, the process would not wait for all threads to complete before exiting as we do not call pthread_exit from the main thread.

(B) How would you implement this using processes rather than threads?

Instead of calling pthread_create, you would call fork for each filename. exec is not acceptable since the original process would get replaced and never writes the results. You would then use pipes to communicate results back to the main process which would write the results to nice.txt and naughty.txt. Other less desirable alternatives include sockets, files, or shared memory. For the latter two options, a wait would need to be used.

Question 25

Spell Finder

Harry has a big test coming up at LogPorts Castle, and he hasn’t read the textbook yet! He needs to be able to perform a specific spell, but he doesn’t know where it is in the textbook.

Harry has asked you, a computer magic major at LogPorts, to write a multiprocessing program for him that can rapidly find the spell in the textbook. Your solution must create num_processes processes, and each process should search an equal portion of the textbook using the `search_for_spell()` function.
You can assume that num_processes is a power of 2, and that textbook_size is a multiple of num_processes.

NOTE: For each blank, you can use as many lines as you need to solve the problem. You may be able to leave some blanks empty. You can assume that all syscalls will not error. **Your solution must run all processes in parallel, A solution in which processes are running serially in NOT allowed.**

```
/* This function will read the textbook file into array `arr`. You can assume that the
textbook will be exactly `textbook_size` characters long. Assume that this is already
implemented. */

void read_textbook_into_array(char *arr, int textbook_size);

/* This function will search for Harry's spell in array `arr`, and print the spell if
found. The function will only search for the spell in the subarray between `start_index`
(inclusive) and `end_index` (exclusive). Assume that this is already implemented. */

search_for_spell(char *arr, int start_index, int end_index);
```

```
void process_array (int num_processes, int textbook_size) {
    char* arr = malloc(sizeof(char) * textbook_size);
    read_textbook_into_array(arr, textbook_size);

    int start_index = 0;
    int end_index = textbook_size;

    for (int i = 0; i < num_processes; i++) {
       pid_t pid = fork();
       if (pid == 0) {

          start_index = (num_processes - i - 1) * textbook_size / num_processes;
          break;

       }
       else {
          end_index -= textbook_size / num_processes;

       }
    }
    search_for_spell(arr, start_index, end_index);
}
```

Also, answer the following Questions:

(A) Why this program is not memory-efficient with respect to the size of textbook?

The first thing that the function does is allocate space for the textbook in memory and read the entire textbook into the heap. However, each time the process is forked, because there is no copy-on-write mechanism, the entire address space (including the in-memory copy of the textbook) will be copied into the child process's address space. As a result, there will be num_processes copies of the textbook in-memory by the time the program completes.

(B) Neville, a computer magic minor, recommends that you use multithreading instead. Justify his recommendation.

Multithreading would allow each of the threads to share the same copy in-memory copy of the textbook.
Context switching between threads is faster than context switching between processes, leading to better performance.

Question 26

In the below program, several threads are created. Which of the following statements **(select all possible)** in foo always print the same memory address when evaluated by different threads in same process?

```
int global;

void* foo(void* arg) {

    printf("%p\n", &foo);
    printf("%p\n", &global);
    printf("%p\n", &arg);
    printf("%p\n", arg);
    return NULL:

}

int main() {
    void* hmem = malloc(1);
    for (int i=0; i<3; i++) {
       pthread_t pid;
       pthread_create(&pid, NULL, foo, hmem);
    }
    return 0;
}
```

□	printf("%p\n", &foo);	□	printf("%p\n", &global);
□	printf("%p\n", arg);	□	printf("%p\n", &arg);
□	None of the above		

Question 27

Beaver Bother Logs

You are writing a program for the beavers of Beaver Bother that will automatically sort log entries into separate log files based on a simple protocol: the first character of each log entry will be a digit representing the index of the destination log file. For example, the entry **2 - Beaver Bother will be saved to log2.txt,** while the **entry 0 - Bothering Beavers will be saved to log0.txt.** Log entries will be collected from standard input. You've written a function, **get_entry**, to get one log entry from **stdin and save it to a null-terminated buffer.**

IMPORTANT: Even though each entry is null-terminated, the null terminator should not be written to the log file.

You also decide that the main process of the program should spawn a child process for each log file (to handle file IO), and you plan to use pipes to forward log entries to each child process. Fill in the missing code below to finish your program!

NOTE: You should only use one line per blank. Extra lines are NOT allowed.

```
#define BUFFER_LEN 1024

int get_entry(FILE * infile, char* buf, size_t buflen) {
   size_t index = 0;
   char ch = fgetc(infile);
   int log_index = ch - '0';
   buf[0] = '\0';
   while ((index < buflen - 2) && (ch != '\n')) {
        buf[index++] = ch;
        buf[index] = '\0';
        ch = fgetc(infile);
    }
   buf[index++] = '\n';
   buf[index] = '\0';
   return log_index;
}

typedef struct pipe_fds {
    int fds[2];
} pipe_fds;

int main(int argc, char** argv) {
    int num_logs = atoi(argv[1]);
    pipe_fds log_fds[num_logs];
    pid_t pid;
    int log_index;
    for (log_index = 0; log_index < num_logs; log_index++) {
        pipe(log_fds[log_index].fds);
        if ((pid = fork()) == 0)
            break;
    }
    if (pid != 0) {
       char str[BUFFER_LEN];
       while (1) {
           log_index = get_entry(stdin, str, BUFFER_LEN);
           size_t bytes_written;
           size_t total = 0;
           size_t str_length = strlen(str);
           int wfd = log_fds[log_index].fds[1];
           while (bytes_written = write(wfd, &str[total], str_length-total)){
               total += bytes_written;
           }
        }
     } // end of if
    else {
       char filename[BUFFER_LEN];
       sprint(filename, “log%d.txt”, log_index);
       int file_fd = open(filename, O_RDWR | O_CREAT, 0666);
       char read_buf[BUFFER_LEN];
       size_t bytes_read;
       int rfd = log_fds[log_index].fds[0];

       while (bytes_read = read(rfd, read_buf, BUFFER_LEN)) {
            size_t bytes_written;
            size_t total = 0;
            while (bytes_written = write(file_fd, &read_buf[total],
                                        bytesread-total)) {
                  total += bytes_written;
            }
       }
   } // end of else
   return 0;
} // end of main
```

Question 28

In the following code snippet, **<u>Assume at least two individual calls to fork( ) succeed.</u>**

```
#include <sys/types.h>
#include <unistd.h>

int main() {
      for (int i = 0; i < 3; i++) {
            fork();
      }
      return 0;
}
```

How many new processes could be created?			
□	0	□	1
☑	2	☑	3
☑	4	☑	5
☑	6	☑	7
□	8		

How many times could fork() be called?			
□	0	□	1
□	2	□	3
☑	4	☑	5
☑	6	☑	7
□	8		

Note: Select all possible options.

Question 29

In the following code snippets assume all calls to `read( )`, `write( )` and `fork( )` succeed. Suppose that `montymole.txt` was empty before running these blocks of code. All codes are separate from each other. **You have to select all possible options that can be content of `montymole.txt` after each code block execution.**

```
int main(int argc, char** argv) {
      int fd1 = open("montymole.txt",
                              O_WRONLY);
      int fd2 = open("montymole.txt",
                              O_WRONLY);
      write(fd1, "mole", 4);
      write(fd2, "whack", 5);
      write(fd2, "mole", 4);
      write(fd1, "mole", 4);
      write(fd1, "mole", 4);
      close(fd1);
      close(fd2);
}
```

☑	whacmolemole	□	whacmolek
□	whackmolemole	□	Nothing
□	molewhackmolemolemole		

```
void new_thread(void* arg) {
      int* fd = (int*)arg;
      write(*fd, "whackwhackmole", 14);
}

int main(int argc, char** argv) {
      int fd1 = open("montymole.txt",
                                O_WRONLY);
      pthread monty_mole;
      pthread_create(&monty_mole, NULL,
                new_thread, (void*)&fd1);

      write(fd1, "mole", 4);
      close(fd1);
}
```

□	molewhack	□	whackwhackmole
☑	whackwhackmolemole	☑	molewhackwhackmole
□	molewhackwhackwhack		

```
void fork_p(pid_t pid, int fd1) {
      if (pid == 0) {
            write(fd1, "whack", 5);
      }
      else {
            write(fd1, "mole", 4);
      }
}

int main(int argc, char** argv) {
      int fd1 = open("montymole.txt",
                        O_WRONLY);
      fork_p(fork(), fd1);
      write(fd1, "mole", 4);
      close(fd1);
}
```

❑	molekmole	❑	whacmmole
❑	whackmole	❑	molemolee
❑	whackmolemole		

None of the above is correct

Question 30

Consider the following piece of code, which can be executed by one or more threads:

```
x = x + 1;
y = x + y;
```

Assumptions:

- Before any thread starts running, both x and y are initialized to 1. After that x and y are only updated by the threads executing the above code segment.
- A thread can be preempted at any point during its execution.
- Each of the two instructions is atomic.

(A). Assume two copies of the above piece of code run concurrently in two threads. Write down all Possible values after both threads finish, one per column
(Number of outputs might be smaller than the number of columns.)

x	3	3						
y	6	7						

(B). Same question as above, but now assume that three copies of the code segment in three threads.

x	4	4	4	4				
y	13	12	11	10				

(C). Assume 10 copies of the same program run concurrently. What is the maximum value of y?

111

More generally let n be the number of concurrent threads. Then, y = n^2 + n + 1

Maximum y occurs when all x = x + 1 executes first, and all y = x + y execute last:

x = x + 1 (x = 2)
...
x = x + 1 (x = n + 1)
y = x + y (y = (n + 1) + 1)
y = x + y (y = 2*(n+1) + 1)
...
 y = x + y (y = n*(n+1) + 1 = n^2 + n + 1)

(D). Assume 10 copies of the same program run concurrently. What is the minimum value of y?

66

Minimum y occurs when an x = x + 1 executes first, and then its corresponding y = x + y executes after it

x = x + 1 (x = 2)
y = x + y (y = 3)
x = x + 1 (x = 3)
y = x + y (y = 6)
x = x + 1 (x = 4)
y = x + y (y = 10)
x = x + 1 (x = 5)
y = x + y (y = 15)
...
x = x + 1 (x = n + 1)
y = x + y
(x = n + 1; y = 1 + 2 + 3 + 4 + 5 + ... + n+1 = (n+1)*(n+2)/2)

Question 31

Consider the following code, assume all fork() and other calls succeed.

```
01: void* rem(void *args) {
02:  printf("Blue: %d\n", *((int*) args));
03:  exit(0);
04: }
05: void* ram(void *args) {
06:  printf("Pink: %d\n", ((int*) args)[0]);
07:  return NULL;
08: }
09: int main(void) {
10:        pid_t pid; pthread_t pthread; int status; //declaring vars
11:
12:       int fd = open("emilia.txt", O_CREAT|O_TRUNC|O_WRONLY, 0666);
13:       int *subaru = (int*) calloc(1, sizeof(int));
14:       printf("Original: %d\n", *subaru);
15:
16:       if(pid = fork()) {
17:            *subaru = 1337;
18:            pid = fork();
19:       }
20:       if(!pid) {
21:            pthread_create(&pthread, NULL, ram, (void*) subaru);
22:       } else {
23:            for(int i = 0; i < 2; i++)
24:                 waitpid(-1, &status, 0);
25:            pthread_create(&pthread, NULL, rem, (void*) subaru);
26:       }
27:       pthread_join(pthread, NULL);
28:       if(*subaru == 1337)
29:            dup2(fd, fileno(stdout));
30:       printf("All done!\n");
31:       return 0;
32: }
```

(A). Including the original process, how many processes are created?

3 Processes

(B). Including the original thread, how many threads are created?

6 Threads

(C). Provide all possible outputs in standard output. If there are multiple possibilities, write all of them.

```
Original: 0        Original: 0        Original: 0
Pink: 1337         Pink: 0            Pink: 0
Pink: 0            Pink: 1337         All done!
All done!          All done!          Pink: 1337
Blue: 1337         Blue: 1337         Blue: 1337
```

(D). Provide all possible contents of emilia.txt. If there are multiple possibilities, write all of them.

All done!

(E). Suppose we deleted line 28, how would the contents of emilia.txt change (if they do)?

All done!
All done!

(F). What if, in addition to doing the change in part (E), we also move line 12 between lines 19 and 20? What would emilia.txt look like then?

All done! - or - a blank file

Question 32

Consider the following C program.

```
int  thread_count = 0;
void* thread_start(void* arg) {
      thread_count++;
      if (thread_count == 3) {
            char* argv[] = { "/bin/ls",  NULL };
            execv(*argv, argv);
      }
      printf("Thread:  %d\n", thread_count);
      return  NULL;
}

int  main(int  argc, char* argv[]) {
      int  i;
      for (i = 0; i < 10; i++) {
            pthread_t* thread = malloc(sizeof(pthread_t));
            pthread_create(thread, NULL, &thread_start, NULL);
            pthread_join(*thread, NULL);
      }
      return  0;
}
```

(A). When you run the program, what will be the output.

Thread: 1
Thread: 2
(contents of current directory)

(B). If we removed the line `pthread_join(*thread, NULL),` could the output change? If so, explicitly describe the way(s) it might differ.

Because the threads are created but aren't waited for, they will execute in a nondeterministic order. So, we could get 0 or more printings of:
Thread: {1…10}
(contents of current directory zero or one times)

Question 33

Consider the following two C programs

```
int i = 100;
char* buf;

void process() {
    pid_t pid;
    int status, rd;
    int j = 1;

    buf = strcpy(malloc(100), " boring ");
    pid = fork();

    if (pid != 0) {
       printf("P Parent : j = %d \n", j);
       i = 162;
       j = 2;

       printf("P Parent : j = %d \n", j);
       printf("P Parent : %s\n ", buf);
       j = 4;
       wait(&status);
    }
    else {
       printf("P Child : i = %d, j = %d\n", i, j);
       printf("P Child : i = %d, j = %d\n", i, j);

       j = 3;
       strcpy(buf, " cool ");
       exit(0);
       }
}
```

```
int i = 100;
char* buf;

void* tfun(void* noarg) {
    int j = 0;
    printf("T Child : i = %d, j = %d\n", i, j);
    printf("T Child : i = %d, j = %d\n", i, j);

    j = 3;
    strcpy(buf, " cool ");
    pthread_exit(NULL);

}

void thread() {
    pthread_t tid;
    int j = 1;
    buf = strcpy(malloc(100), " boring ");

    pthread_create(&tid, NULL, tfun, NULL);
    printf("T Parent : j = %d \n", j);
    i = 162;
    j = 2;

    printf("T Parent : j = %d \n", j);
    printf("T Parent : %s\n ", buf);
    j = 4;
    pthread_join(tid, NULL);

}
```

The above two programs are nearly identical process based and thread based programs. For each question, answer YES or NO and provide a short, crisp explanation. Assume all functions will succeed.

(A). Can the output of the process-based program ever include (meaning they appear in this order, possibly with other output interleaved) the following? Why?

```
P Parent: j = 1
P Child: i = 100, j = 1
P Parent: j = 2
```

YES. The child process may get switched in after the parent prints its initial value of its stack variable j and the parent later switch back in. The child obtained a copy of the parent's stack and heap at the fork and it has changed neither when it first prints.

(B). Can the output of the thread-based program ever include the following? Why?

```
T Parent: j = 1
T Child: i = 100, j = 1
T Parent: j = 2
```

NO. The child thread has its own j on its stack, and this j only have value 0 when the child thread prints.

(C). Can the output of the process-based program ever include the following? Why?

```
P Parent: j = 1
P Parent: j = 2
P Child: i = 100, j = 1
```

YES. The child process may get switched in after the parent process executes its if block. The child obtained a copy of the parent's stack and heap at the fork and it has changed neither when it first prints.

(D). Can the output of the thread-based program ever include the following? Why?

```
T Parent: j = 1
T Parent: j = 2
T Parent: i = 100, j = 1
```

NO. If the child is switched in after the parent reaches after pthread_create, the value of shared variable i must be 162. And also j in the child thread should be 0.

(E). Can the output of the process-based program ever include the following? Why?

```
P Parent: cool
```

NO. The parent process never changes the value of *buf from what was set using malloc. The child has its own copy of the heap.

(F). Can the output of the process-based program ever include the following? Why?

```
T Parent: cool
```

YES. The child thread could get switched in and execute the strcpy command before the parent resumes to execute line to print the value of buf.

Question 34

Consider the following C code

```
#define NUM_THREADS 3

void* thread_function(void* arg) {
	int* arg = (int*)arg;
	printf("Thread %d: Running\n", *arg);

	if (!fork()) {
```

```
            printf("Where am I, who called me here /: did you %d\n", getppid());
            printf("Well then will you reap me OR I should make more threads :)\n");
            if (*arg < 1)
                  exit(0);
      }
      else
      {
            sleep(2); // until child completes its execution
            printf("What is my ID /:) is It %d\n", getpid());
      }

      pthread_exit(NULL);
}

int main() {
      pthread_t threads[NUM_THREADS];
      int thread_args[NUM_THREADS];
      int i, status;

      for (i = 0; i < NUM_THREADS; i++) {
            thread_args[i] = i + 1;
            status = pthread_create(&threads[i], NULL, thread_function,
                                                       (void*)&thread_args[i]);
            wait(NULL);       // can I reap that child of yourssss
      }

      pthread_exit(NULL);

      printf("Byee\n");

      return 0;
}
```

No. of processes created?	3
No. of threads created?	3
No. of zombie processes just before the program exits (Main program)?	3
Will printf("Byee\n") execute?	NO
Suppose if we remove sleep() then will the printf() statements always execute in child?	YES
No. of zombie processes if wait() is removed from for loop and written in place of sleep()?	0
Is there a possibility that this program terminates without showing any output, given that all syscalls i.e., pthread_create, etc succeed?	NO
Write pid of all processes that have one or more children. Given that starting (main) process has a pid of 8342 and any child of a process has (pidOFparent + 1) as it's pid, which surely increments as no of childs increase. No two processes have same pid, in case of clash give next pid available.	8342

Question 35

Consider the following C code.

```
#define NUM_THREADS 3

void* thread_function(void* arg) {
    int* arg = (int*)arg;
    printf("Thread %d: Running\n", *arg);

    sleep(10);

    printf("Here we are from process %d\n", getpid());

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    int i, status;

    for (i = 0; i < NUM_THREADS; i++) {
        thread_args[i] = i + 1;
        status = pthread_create(&threads[i], NULL, thread_function,
                                               (void*)&thread_args[i]);
    }
    // suppose this fork call executes before
    // any thread has completed its sleep(10)
    if (!fork())
    {
        printf("Do I have your threads, hahaha\n");
        printf("Can I join your threads or MINE??\n");
        for (i = 0; i < NUM_THREADS; i++) {
            if (pthread_join(threads[i], NULL))
                printf("No i Can't join..%d", i);
            else
                printf("Yes I can join..%d", i);
        }
        pthread_exit(NULL); // what will this do? hmmm...
    }
    else
    {
        wait(NULL);
        for (i = 0; i < NUM_THREADS; i++) {
            if (pthread_join(threads[i], NULL))
                printf("No i myself Can't join..%d", i);
            else
                printf("Yes I mself can join..%d", i);
        }
    }
    printf("Bye\n");
    pthread_exit(NULL);


    return 0;
}
```

Answer the following Questions

A) Write any three (3) possible outputs of the code, provided parent process has 8342 and child has 8343 as their pid respectively. **Only consider printf statements in the function.**

Thread 1: Running Thread 2: Running Thread 3: Running Here we are from process 8342 Here we are from process 8342 Here we are from process 8342	Thread 1: Running Thread 2: Running Here we are from process 8342 Here we are from process 8342 Thread 3: Running Here we are from process 8342	Thread 1: Running Here we are from process 8342 Thread 3: Running Thread 2: Running Here we are from process 8342 Here we are from process 8342

B) Give answers of following

Total No. of threads created?	3
How many time printf("Bye\n") will be executed?	1
How many calls to pthread_join can produce error?	3
Suppose before fork call thread with threads[0] has completed it's execution. Now How many threads can be at max?	3
Suppose there is a fork call just before for loop written as fork(); How many possible (Max) threads can be created?	6
Considering the above scenario, can there be any case that program terminates and all threads have not completed their execution?	NO
Suppose in the original scenario, if all pthread_join calls fail, both in parent and child, can there be a possibility that program show only Bye as output?	NO

Question 36

Scheduling

All timeslice-based algorithms have a timeslice of one unit. The currently running thread is not in the ready Queue while it is running. An arriving thread is run at the beginning of its arrival time, if the scheduling policy allows it.

Fill in ALL blanks in EACH table – each blank has an unambiguous answer.

For the missing schedulers, the possibilities are **SRTF, RR, and Priority.** *Priority is a preemptive scheduler.*

Hint: Fill in the entry time (below) for Thread C first!

Entry Times	
A	1
B	2
C	***5***
D	8

Priorities	
A	3
B	4
C	5
D	6

↓**Current Time**	**Currently Scheduled Process**		
Scheduler →	**FIFO**	***SRTF***	***Priority***
1	A	A	A
2	A	A	B
3	A	A	B
4	B	*B*	*B*
5	B	*C*	*C*
6	B	*B*	*A*
7	C	*B*	*A*
8	D	*D*	*D*
9	D	*D*	*D*
10	D	*D*	*D*
Avg. Turnaround Time	3.5	*3.25*	*3.5*

Question 37

Consider the following table of processes and their associated arrival and running times.

Process ID	Arrival Time	CPU Running Time
Process A	0	2
Process B	1	6
Process C	4	1
Process D	7	4
Process E	8	3

Show the scheduling order for these processes under 3 policies: First Come First Serve **(FCFS),** Shortest-Remaining-Time-First **(SRTF),** Round-Robin **(RR) with time slice quantum = 1.** Assume that context switch overhead is 0, that new processes are available for scheduling as soon as they arrive,and that new processes are added to the **head** of the queue **except for FCFS, where they are added to the tail.**

Time Slot	FCFS	SRTF	RR
0	*A*	*A*	*A*
1	*A*	*A*	*B*
2	*B*	*B*	*A*
3	*B*	*B*	*B*
4	*B*	*C*	*C*
5	*B*	*B*	*B*
6	*B*	*B*	*B*
7	*B*	*B*	*D*
8	*C*	*B*	*E*
9	*D*	*E*	*B*
10	*D*	*E*	*D*
11	*D*	*E*	*E*
12	*D*	*D*	*B*
13	*E*	*D*	*D*
14	*E*	*D*	*E*
15	*E*	*D*	*D*

Question 38

Suppose we have the following set of processes, with their respective CPU burst times and priorities (here, processes with a higher priority value have higher priority).

Process ID	Arrival Time	CPU Burst (running) Time	Priority
A	0	3	4
B	1	1	2
C	3	5	5
D	6	2	6
E	8	2	1

For each of the three scheduling algorithms, fill in the the table below with the processes that are running on the CPU at each time tick. Assume the following:

- The time quantum for RR is 1 clock tick and when RR quantum expires, the currently running thread is added to the end of the ready list before any newly arriving threads.
- *Assume that context switch overhead is 0, and that new processes are available for scheduling as soon as they arrive.*

Time	RR	SRTF	Priority
0	***A***	***A***	***A***
1	***A***	***B***	***A***
2	***B***	***A***	***A***
3	***A***	***A***	***C***
4	***C***	***C***	***C***
5	***C***	***C***	***C***
6	***C***	***D***	***D***
7	***D***	***D***	***D***
8	***C***	***E***	***C***
9	***D***	***E***	***C***
10	***E***	***C***	***B***
11	***C***	***C***	***E***
12	***E***	***C***	***E***
Average Wait Time	***(1+1+4+2+3)/5=2.2***	***(1+0+5+0+0)/5=1.2***	***(0+9+2+0+3)/5=2.8***

Question 39

CPU scheduling.

Consider the following **single-threaded** processes,arrival times, and CPU processing requirements:

Process ID (PID)	Arrival Time	Processing Time
1	0	6
2	2	4
3	3	5
4	6	2

A) For each scheduling algorithm, fill in the table with the ID of theprocess that is running on the CPU. Each row corresponds to a time unit.

- For time slice-based algorithms, assume one unit time slice.
- When a process arrives it is immediately eligible for scheduling, e.g.,process 2 that arrives at time 2 can be scheduled during time unit 2.
- If a process is preempted, it is added at the **tail of the ready queue.**

Time	FIFO	RR	SJF
0	1	1	1
1	1	1	1
2	1	1	1
3	1	2	1
4	1	1	1
5	1	3	1
6	2	2	4
7	2	1	4
8	2	3	2
9	2	4	2
10	3	2	2
11	3	1	2
12	3	3	3
13	3	4	3
14	3	2	3
15	4	3	3
16	4	3	3

B) Calculate the response times of individual processes for each of the scheduling algorithms. The response time is defined as the time a process takes tocomplete after it arrives.

	PID 1	PID 2	PID 3	PID 4
FIFO	6	8	12	11
RR	12	13	14	8
SJF	6	10	14	2

C) Consider same processes and arrival times, but assume now a processor with **two** CPUs. Assume CPU 0 is busy for the first two time units. For each scheduling algorithm, fill in the table with the ID of the process that is running on each CPU.

- For any non-time slice-based algorithm, assume that once a process startsrunning on a CPU, it keeps running on the same CPU till the end.
- If both CPUs are free, assume CPU 0 is allocated first.

Time	CPU #	FIFO	RR	SJF
0	**0**	X	X	X
	1	1	1	1
1	**0**	X	X	X
	1	1	1	1
2	**0**	2	1	2
	1	1	2	1
3	**0**	2	1	2
	1	1	2	1
4	**0**	2	3	2
	1	1	1	1
5	**0**	2	2	2
	1	1	3	1
6	**0**	3	1	4
	1	4	2	3
7	**0**	3	3	4
	1	4	4	3
8	**0**	3	3	
	1		4	3
9	**0**	3	3	
	1			3
10	**0**	3		
	1			3

Question 40

Advanced Bike-share Scheduling

In this problem, we want you to apply what you have learned about scheduling algorithms to scheduling resources in a new domain – bike-share scheduling. Ford GoBikes are now in Islamabad and they need **your** help in figuring out how to schedule the resources in their system.

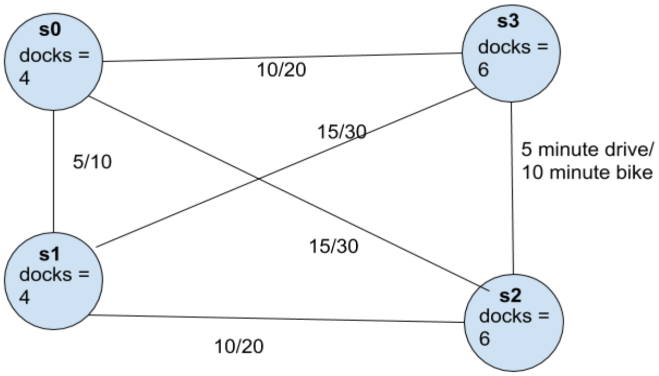
In a bike-share system, the **bikes** are parked in **stations** each with some number of **docks** for bikes. A potential **rider** walks up to a station, checks out a bike from a dock, rides to a destination station, returns the bike to an empty dock, and walks off. In order to avoid blocking riders, the bike-share operator periodically rebalances bikes by driving around in a **van** and adding or removing bikes from stations depending on demand.

Important Rebalancer Assumptions:

- Van has infinite capacity and infinite bikes and starts at s0 at 7:05
- Once the van arrives at a station, it stays there for 5 minutes to perform the rebalancing
- Multi-level priority uses the demand - (context switch time/10) to determine dynamic priority
- Pre-emptive algorithms pick up or drop off one bike on every visit; otherwise full rebalance to optimal state on every visit
- the optimal state is 50% full, every station starts in optimal state
- the demand can be represented by abs(n_bikes - n_empty_slots)/docks, and
- the context switch time is the drive time between stations + unloading/reloading time

Trips by riders

Start station	Start time	End station	End time
s1	7:00	s2	7:20
s3	7:05	s0	7:25
s3	7:10	s3	7:30
s3	7:20	s2	7:30



A) Fill out the table in the given on next page with the location of the van in each time slot, assuming each of the classic scheduling algorithms applied to this new context. Use **T** if the van is in transit. **We have filled in Round-Robin for you.**

Time	FCFS	Round-robin	Most imbalanced first (non- preempt)	Most imbalanced first (preempt)	Multi-level priority
7:06	**T**	**T**	**T**	**T**	**T**
7:11	**s1**	**s1**	**s1**	**S1**	**s1**
7:16	**T**	**T**	**T**	**T**	**T**
7:21	**T**	**T**	**T**	**T**	**T**
7:26	**T**	**T**	**T**	**T**	**T**
7:31	**s3**	**s3**	**s3**	**s3**	**s3**
7:36	**T**	**T**	**T**	**T**	**T**
7:41	**s2**	**s2**	**s2**	**s2**	**s2**
7:46	**T**	**T**	**T**	**T**	**T**
7:51	**T**	**T**	**T**	**T**	**s3**
7:56	**T**	**T**	**T**	**T**	**T**
8:01	**s0**	**s0**	**s0**	**s0**	**s2**
8:06		**T**		**T / T**	**T**
8:11		**T**		**T / T**	**T**
8:16		**s3**		**T / s3**	**T**
8:21		**T**		**s2 / T**	**s0**
8:26		**s2**		**T / s2**	
8:31				**s3 /**	

B) Briefly, in two to three sentences, explain what is common among the algorithms that work well in this scenario.

Non pre-emptive algorithms work much better than pre-emptive algorithms.

C) Briefly, in two to three sentences, explain; what are the characteristics of the problem that makes these algorithms particularly suitable.

Due to the large context switch time.