CS 3214 Midterm

This is a closed-book, closed-internet, closed-cell phone and closed-computer exam. However, you may refer to one double sided page of prepared notes. Your exam should have 11 pages with 3 topics totaling 100 points. You have 75 minutes. Please write your answers in the space provided on the exam paper. If you unstaple your exam, please put your initials on all pages. You may use the back of pages if necessary, but please indicate if you do so we know where to look for your solution. You may ask us for additional pages of scratch paper. You must submit all sheets you use with your exam. However, we will not grade what you scribble on your scratch paper unless you indicate you want us to do so. Answers will be graded on correctness and clarity. The space in which to write answers to the questions is kept purposefully tight, requiring you to be concise. You will lose points if your solution is more complicated than necessary or if you provide extraneous, but incorrect information along with a correct solution.

Name (printed) _____

I accept the letter and the spirit of the Virginia Tech undergraduate honor code – I will not give and have not received aid on this exam.

(signed) _____

#	Problem	Points	Score
I	Processes and Threads	40	
II	Input/Output	20	
III	Synchronization	40	
	Total	100	

Part I: (Grader: David 1-(1), Luna 1-(2), Safdar and Ali 1-(3), Scott 1-(4)
Part II: (Grader: Safdar 2-(1), Safdar 2-(2)
Part III: (Grader: Ali 3-(1), Elmer 3-(2)

I. Processes (40 points)

Question I.1 (10 points)

A process has 3 threads, T1, T2, and T3, and its code does not contain any `exec*()` commands. T1 opens 3 files and T2 creates a pipe. After these actions, T1 executes a `fork()` command and T2 executes a `fork()` whose child immediately calls `execvp()` to execute a single-threaded program. T1 closes the three files and then terminates. Note that all processes use Pthreads.

(a) (2 points) How many different processes are running? Why?

Answer: 3 processes.
Explanation: the original process does two forks each of which creates a new process. The other action do not create/terminate processes

(b) (2 points) How many different programs are being executed? Why?

Answer: 2 programs.
Explanation: the original process and its threads along with the first child are executing the same program. The second thread does an execvp() to execute a new/different program.

(c) (3 points) How many open file descriptors are there? Why?

Answer: 21 file descriptors.
Explanation: the original process has 8 file descriptors when it forks its two child processes (3 for STDIN/OUT/ERR) 3 for the files and two for the pipes. These 8 file descriptors are inherited by each child. The original process then closes the 3 file descriptors for the files leaving 5 open file descriptors.

(d) (2 points) Would any of the above answers change if the `execvp()` call was not made? If so, explain how.

Answer: The answer to (b) would change
Explanation: Without the execvp() call all processes would be executing the same program (the original program of the parent).

(e) (1 point) Thread T3 makes a system call that causes all threads in its process to terminate. Which system call could it be?

Answer: exit()
Explanation: the exit() call terminates all activity in the process.

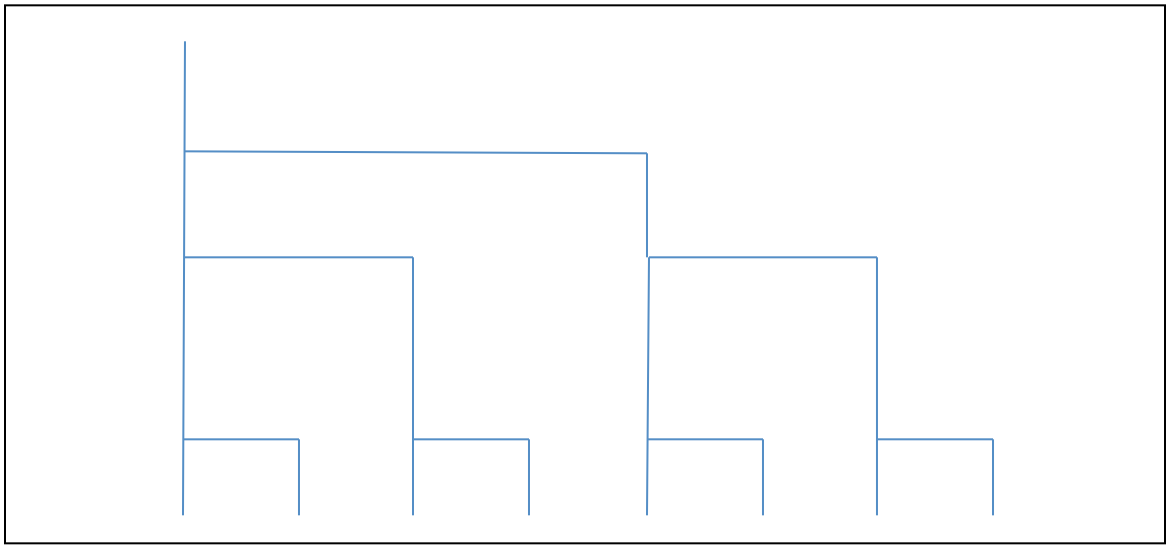
Question I.2 (10 points)

How many times does the following program print "Hello World"? Draw a simple tree diagram to show the parent-child hierarchy of the spawned processes.

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

int main() {
int i;
for (i = 0; i < 3; i++)
    fork();
printf("Hello World\n");
return 0;
}
```

Answer: "Hello World" is printed 8 times.



Question I.3 (10 points)

Circle the valid state transition for process A in each of the following cases.

(2 points) A context switch is performed because process A was "picked" by the OS scheduler for execution.

(a) locked → Ready
(b) Blocked → Running
(c) Ready → Running
(d) Running → Ready
(e) None of the above

(2 points) Process A performs the `exit()` system call.

(a) Running → Blocked
(b) Running → Ready
(c) Running → Running
(d) Ready → Ready
(e) None of the above

(2 points) Process A performs an `exec*()` system call.

(a) Running → Blocked
(b) Running → Running
(c) Running → Ready
(d) Both (b) and (c)
(e) None of the above

(2 points) Process A receives SIGSTOP.

(a) Running → Blocked
(b) Running → Stopped
(c) Running → Ready
(d) Blocked → Ready
(e) None of the above

(2 points) Process A performs the `pause()` system call.

(a) Running → Blocked
(b) Running → Stopped
(c) Running → Ready
(d) Blocked → Ready
(e) None of the above

Question I.4 (10 points)

Consider the following example program. List all legal outputs this program may produce when executed on a Unix system. The output consists of strings made up of multiple letters.

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

// W(A) means write(1, "A", sizeof "A")
#define W(x) write(1, #x, sizeof #x)

int main()
{
   W(A);
   int child = fork();
   W(B);
   if (child)
          wait(NULL);
   W(C);
}
```

Answer: Two possible answers: ABBCC and ABCBC

II. I/O (20 points)

Question II.1 (15 points)

For a certain shell the command A || B means that the standard output of A should be connected to the standard input of B and the standard input of A should be connected to the standard output of B. Write the C code of a parent process that would create the above configuration.

ANSWER:

```
#define WRITE  1
#define READ   0

int pipe1[2], pipe2[2];
pipe(pipe1);
pipe(pipe2);

if(fork() == 0) {
    dup2(pipe1[WRITE], STDOUT);
    dup2(pipe2[READ], STDIN);
    close(pipe1[READ]);
    close(pipe2[WRITE]);
    exec(A);
}
if(fork() == 0) {
    dup2(pipe1[READ], STDIN);
    dup2(pipe2[WRITE], STDOUT);
    close(pipe1[WRITE]);
    close(pipe2[READ]);
    exec(B);
}

close(pipe1[READ]);
close(pipe1[WRITE]);
close(pipe2[READ]);
close(pipe2[WRITE]);
```

Question II.2 (5 points)

(a) (2 points) Circle the BEST answer. A file descriptor is an example of an abstraction that is . . .

A provided by modern disks
B provided by the standard C library
C provided by the hardware's privileged mode
D provided by Unix
E provided by the linker

(b) (3 points) Circle ALL correct options. When will a `pipe()` call fail?

A Too many descriptors are active.
B The system file table is full.
C The fields buffer is an invalid areas of the process's address space.
D The first descriptor is used as the read end of the file.
E The second file descriptor is used as the write end.

Note: Answer C is also correct. No points were deducted for not selecting this answer because the wording of the question is slightly ambiguous.

III. Synchronization (40 points)

Question III.1 (20 points)

The following code is a famous bug in a widely used open source SQL server, MySQL.

```
Thread 1:
if (thd->proc_info) {
  ...
  fputs(thd->proc_info, ...);
  ...
}

Thread 2:
thd->proc_info = NULL;
```

(a) (5 points) Explain the problem in the above code.

ANSWER: the code contains an atomicity problem because the test of thd->proc_info and its use in the fputs statement are not in a critical region. After the test the value of thd->proc_info could be changed to NULL by Thread 2.

(b) (5 points) Does the following code fix the problem? Explain your answer on the top of the next page.

```
pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;

Thread 1:
if (thd->proc_info) {
...
pthread_mutex_lock(&proc_info_lock);
fputs(thd->proc_info, ...);
pthread_mutex_unlock(&proc_info_lock);
...
}

Thread 2:
pthread_mutex_lock(&proc_info_lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&proc_info_lock);
```

ANSWER: the code still contains an atomicity problem because the test of thd->proc_info and its use in the fputs statement are not in a critical region. After the test the value of thd->proc_info could be changed to NULL by Thread 2 despite the use of the locks.

(c) (10 points) Write code to solve the problem in the code from part (a).

```
pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;

Thread 1:
pthread_mutex_lock(&proc_info_lock);  // move the locking here
if (thd->proc_info) {
...
fputs(thd->proc_info, ...);
pthread_mutex_unlock(&proc_info_lock);
...
}

Thread 2:
pthread_mutex_lock(&proc_info_lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&proc_info_lock);
```

Question III.22 (20 points)

(a) (5 points) An application has two concurrent activities. State and briefly explain two different requirements that would lead you to use two processes rather than two threads.

ANSWER: Security – to prevent the activities from sharing data
Different application code – if each activity is executing a distinct program

(b) (5 points) "The **volatile** keyword indicates that a value may change between different accesses, even if it does not appear to be modified." [Wikipedia] Explain why volatile is needed in the following example?

```
int coin_flip;
volatile bool coin_flip_done;

static void * thread1(void *_) {
    coin_flip = rand() % 2;
    coin_flip_done = true;
    printf("Thread 1: flipped coin %d\n", coin_flip);
    return NULL;
}

static void * thread2(void *_) {
      while (!coin_flip_done)
        continue;
    printf("Thread 2: flipped coin %d\n", coin_flip);
    return NULL;
}
```

ANSWER: We need to prevent the compiler optimization that would put coin_flip_done in a register and never see the change made by the other thread.

(c) (4 points) State whether the following statements are True or False.

Question	True	False
A thread **cannot** acquire more than one lock.		X
A mutex **can** be locked more than once.	X	
A signal handler **cannot** lock a mutex or signal semaphore.		X
A monitor **can** have more than one condition variable.		X

NOTE: it is OK if the answer to the second part is False if it included a comment interpreting the second part as meaning "at one time".

(d) (6 points) Two threads are using the following approach to implement a lock acquisition protocol. What problem may arise?

```
top:
  lock(L1);
    if (trylock(L2) == -1) {
          unlock(L1);
          goto top;
    }
```

Note: the `trylock()` routine will acquire the lock (if it is available) or return -1.

ANSWER: This could result in some thread, T, never being able to acquire the lock when multiple threads are contending for the lock. Every time T does the trylock(L2) operation some other thread has already managed to acquire L2.