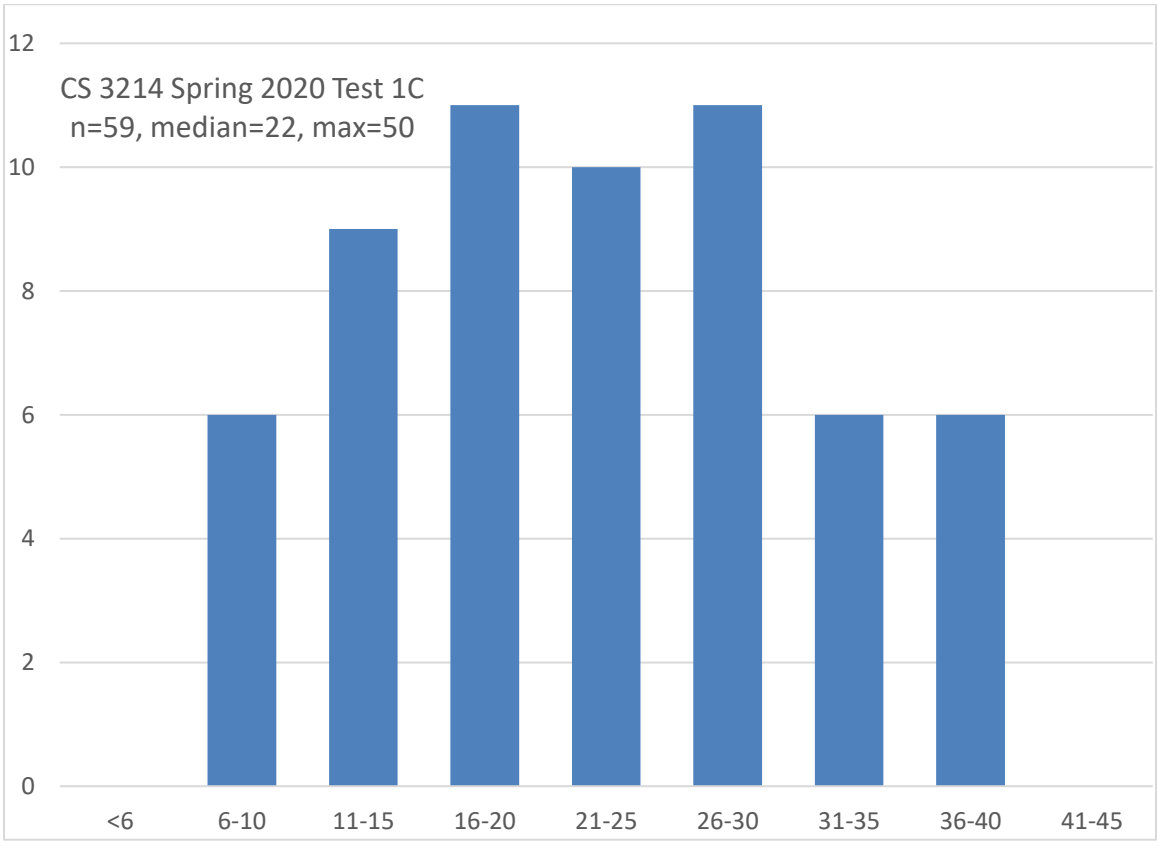
CS 3214 Test 1C Solution

59 students took this test. Statistics are shown below. For questions contact me (gback@vt.edu). The note sheet should be attached to your test. You may bring it to the final exam.

	Q1	Q2	Q3	Total
MAX	12	12	20	39
MIN	0	1	1	7
AVG	6.93	3.81	11.88	22.63
STD	4.26	3.38	4.09	8.78
MEDIAN	7	2	12	22



Solutions are displayed in this color.

1. Parent and Child Processes (12 pts)

Consider the following C program executing on Linux:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

pid_t save_cpid;                                                    //  1

int main() {                                                        //  2

   pid_t cpid = fork();                                             //  3
   if ( cpid < 0 ) {                                                //  4
      perror("fork"), exit(1);                                      //  5
   }                                                                //  6

   save_cpid = cpid;                                                //  7

   if ( cpid == 0 ) {                                               //  8
      printf("Child is %d and parent is %d\n", getpid(), getppid()); //  9
      save_cpid = save_cpid + 25;                                   // 10
      sleep(10);   // sleep 10s                                     // 11
   } else {                                                         // 12
      printf("Parent is %d and child is %d\n", getpid(), cpid);     // 13
      save_cpid = save_cpid - 25;                                   // 14
   }                                                                // 15

   printf("%d sees save_cpid = %d\n", getpid(), save_cpid);         // 16

   if ( cpid == 0 ) {                                               // 17
      exit(42);                                                     // 18
   }                                                                // 19

   int child_status;                                                // 20
   pid_t wpid = wait(&child_status);                                // 21

   if (wpid == -1)                                                  // 22
      printf("Could not wait for child yet.\n");                    // 23
   else if ( WIFEXITED(child_status) )                              // 24
      printf("Child %d terminated with exit status %d\n",           // 25
               wpid, WEXITSTATUS(child_status));                    // 26
   else                                                             // 27
      printf("Child %d terminated abnormally\n", wpid);             // 28
   return 0;                                                        // 29
}
```

When the program was executed, lines 9 and 13 produced the following output (though not necessarily in this order)

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

a) (6 pts) What output would have been produced by line 16?

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

(in any order)

b) (6 pts) Which, if any, of lines 23, 25, or 28 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 25 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 18. Conversely, it would work even if the child had already exited by the time the parent calls wait.

2. IPC via Pipes (18 pts)

a) (9 pts) Consider the following program, executing under normal conditions on our rlogin cluster:

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

int
main()
{
    int fd[2];
    assert(pipe(fd) == 0);
    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;
        }
    } else {
        assert(dup2(fd[1], STDOUT_FILENO) == 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;
        }
    }
    printf("DONE\n");
}
```

Which of the following are possible outputs of this program? Check all that apply!

❑	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

The child process's stdout is redirected to a pipe from which the parent process reads. The child execs 'echo ECHO' which writes ECHO to stdout and thus to the pipe, which the parent reads from. Since execv() does not return, DONE is output only once.
Pipes keep their data even if the processes writing to them have already terminated. Thus, the above is the only possible output.

b) (9 pts) The 'time' bash built-in can be used as follows:

```
$ time <some command possibly with pipes goes here>
```

After the user's job has finished, the user will see output that has the following format:

```
real    0m<REAL>s
user    0m<USER>s
sys     0m<SYSTEM>s
```

where (a) **<REAL>**, (b) **<USER>,** and (c) **<SYSTEM>** are replaced with (a) the number of seconds that elapsed as could be observed on a wallclock, (b) the number of CPU seconds all processes involved in the job spent

executing in user mode cumulatively, and (c) the number of CPU seconds these processes spent executing in kernel mode, respectively.

Consider the following terminal session in which the user issues 3 commands that involve I/O redirection:

```
1) $ time seq 1 15000000 > /tmp/numbers-to-15mil.txt

   real    0m0.302s
   user    0m0.200s
   sys     0m0.102s

2) $ time factor < /tmp/numbers-to-15mil.txt > /tmp/factors.txt

   real    0m6.004s
   user    0m5.731s
   sys     0m0.274s

3) $ time wc /tmp/factors.txt
    15000000  72176045 328794419 /tmp/factors.txt

   real    0m1.917s
   user    0m1.839s
   sys     0m0.078s
```

The `seq` command prints a series of numbers, one per line. The `factor` program reads a number from its standard input stream, computes its prime factorization and outputs the result, repeating this process until it exhausts its input. The `wc` utility reads its standard input stream line-by-line, counting the number of characters and words along the way. All three programs are traditional, single-threaded programs.

For instance:

```
$ seq 16 19 | factor
16: 2 2 2 2
17: 17
18: 2 3 3
19: 19
```

i. (3 pts) Referring to the simplified process state diagram discussed in lecture, check, for each command, the state in which its process spent most of its time:

	RUNNING	READY	BLOCKED
`1) seq 1 15000000 > …`	☑	❑	❑

2) factor < … > …	☑	❑	❑
3) wc < …	☑	❑	❑

Process 1) used 0.200s + 0.102s CPU seconds in 0.302 seconds, which is 100%. A process can use CPU time only if it's in the RUNNING state. Similar numbers apply to process 2) and 3) – all three processes were CPU bound, and there was apparently no contention for CPUs on the machine I ran them on (that is, the machine had enough free cores at the time.)

ii. (6 pts) Now the user types instead:

```
$ time seq 1 15000000 | factor | wc
15000000 72176045 328794419

real    0m????s
user    0m????s
sys     0m????s
```

Which of the following would be the most likely output when run on the multiprocessor machines of our rlogin cluster (currently 2 processor machines with 10 cores per processor)?

A ❑		B ❑		C ☑		D ❑	
real	0m8.223s	real	0m6.251s	real	0m6.251s	real	0m3.129s
user	0m7.700s	user	0m5.805s	user	0m8.282s	user	0m9.220s
sys	0m0.454s	sys	0m0.446s	sys	0m0.446s	sys	0m0.879s

Running the three commands in a pipe allowed them to be started simultaneously on different cores. As the first one produced output, the second process could process those numbers as per its description. The output of `factor` was in turn fed to `wc` as soon as it is produced. This way, the executions of these three programs could overlap. The overall consumed user + system time was roughly equivalent (0.200+5.731+1.839 = 7.7 vs 8.282s, and 0.102+0.274+0.078 = 0.454 vs 0.446s). However, the execution of 'factor' (roughly 6s when run separately) could almost fully overlap with the execution of 'seq' and 'wc'.

Answer A could be ruled out because it would imply that 'factor' and 'seq' run in sequence rather than concurrently.
Answer B could be ruled out because it would mean that the overall consumed CPU time would somehow be much smaller even though the processes were doing the same work.
Answer D could be ruled out because merely running serial code in a pipeline doesn't parallelize it – 'factor' will still need roughly 6s to perform its task.

3. Short Answers (20 pts)

Evaluate each of the following statements. If the statement is true, say so. If the statements is false, explain why.

a) (4 pts) "A successful call to `fork()` creates a child process and triggers a context switch to that child process."

❑ is true

☑ *is false, because there is no guarantee of a context switch to the child process. When (and even on which CPU) the child process will be run is entirely up to the decisions of the scheduler. Unix OS's have used different scheduling policies over time. If the child process runs on a different core, the parent wouldn't have to context switch at all.*

b) (4 pts) "When user-implemented signal handlers are called, they execute in kernel mode."

❑ is true

☑ *is false, because signal handlers are still provided by a user. If they executed in kernel mode, they would have full control of the machine and could compromise security and availability, inadvertently or maliciously.*

c) (4 pts) "When a process is executing on a CPU under a multitasking operating system, an interrupt handler may cause a context switch to another process."

☑ *is true.*
This is a common occurrence – an interrupt handler may signal the completion of a time slice, or perhaps the arrival of a new network packet, or completion of disk I/O – all of these are events that may cause the OS's scheduler to context switch to another process that was either waiting for the CPU or for one of the I/Os to complete.

❑ is false, because

d) (4 pts) "When a user process attempts to access invalid memory addresses, it is stopped by default until an administrator can examine its state and terminate the process."

❑ is true.

☑ *is false, because the OS will send a SIGSEGV signal whose associated default behavior is to terminate the process.*

e) (4 pts) “Processes executing systems calls always transition to the BLOCKED state where they remain until the system call completes.”

❑ is true.

☑ *is false, because the BLOCKED state is entered only if the system call causes the process to wait for something that will complete in the future. For instance, if a process makes a system call to receive a packet from the network, it will be BLOCKED only if no packet has already been received and buffered. Similarly, a process trying to read from disk may block only if the data is not already cached in memory. Some system calls (e.g., getpid()) will not block at all. Some system calls (e.g. waitpid(,,WNOHANG)) have an option to explicitly instruct the OS to avoid blocking even if the system call cannot complete right away. In all of these cases, the process remains in the RUNNING state unless preempted.*