CS 3214 Midterm

This is a closed-book, closed-internet, closed-cell phone and closed-computer exam. However, you may refer to your sheet of prepared notes. Your exam should have 14 pages with 4 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. In three places you are asked to write your answers on the back of the pages. 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	Stack Protocol	29	
II	Optimization	16	
III	Processes and Signals	40	
IV	Threads and Synchronization	15	
	Total	100	

I. Stack Protocol (29 points)

The following C code involves two functions, `word_sum` which has a structure as a parameter and another as its return value, and `prod` which calls `word_sum`:

```
typedef struct {
   int a;
   int *p;
} str1;
```

```
typedef struct {
   int sum;
   int diff;
} str2;
```

```
str2 word_sum(str1 s1) {
   str2 result;
   result.sum = s1.a + *s1.p;
   result.diff = s1.a - *s1.p;
   return result;
}

int prod(int x, int y) {
   str1 s1;
   str2 s2;
   s1.a = x;
   s1.p = &y;
   s2 = word_sum(s1);
   return s2.sum * s2.diff;
}
```

gcc generates the following code for the two functions:

```
word_sum:                    #  1
    pushl  %ebp              #  2
    movl   %esp, %ebp        #  3
    pushl  $ebx              #  4
    movl    8(%ebp), %eax    #  5
    movl   12(%ebp), %ebx    #  6
    movl   16(%ebp), $edx    #  7
    movl   (%edx), %edx      #  8
    movl   %ebx, %ecx        #  9
    subl   %edx, %ecx        # 10
    movl   %ecx, 4(%eax)     # 11
    addl   %ebx, %edx        # 12
    movl   %edx, (%eax)      # 13
    popl   %ebx              # 14
    popl   %ebp              # 15
    ret    $4                # 16
```

```
prod:                        #  1
    pushl  %ebp              #  2
    movl   %esp, %ebp        #  3
    subl   $20, %esp         #  4
    leal   12(%ebp), %edx    #  5
    leal   -8(%ebp), %ecx    #  6
    movl    8(%ebp), %eax    #  7
    movl   %eax, 4(%esp)     #  8
    movl   %edx, 8(%esp)     #  9
    movl   %ecx, (%esp)      # 10
    call   word_sum          # 11
    subl   $4, %esp          # 12
    movl   -4(%ebp), %eax    # 13
    imull  -8(%ebp), %eax    # 14
    leave                    # 15
    ret                      # 16
```

The instruction `ret $4` is like a normal `ret` instruction, except that it increments the stack pointer by 8 (4 for the return address plus 4 more), rather than 4.

(a) (10 points) Diagram the stack frame for the function `prod`, as it would exist immediately before the call to `word_sum` in line 11 of `prod`. Provide a description of the contents of each word in the stack frame.

Answer:

address	value	inferred from
. . .		
ebp + 12	y	
ebp + 8	x	
	return-to address	
ebp	old ebp	
ebp – 4	s2.p	prod # 10
ebp – 8	s2.a	prod # 10
esp + 8	s1.p	prod # 9
esp + 4	s1.a	prod # 8
esp	&s2 == ebp - 4	prod # 10

(b) (9 points) From lines 5-7 in `word_sum`, it appears that three values are being retrieved from the stack, even though the function has only one parameter. Describe what these three values are.

Answer:

at ebp + 8 we have a pointer to a struct that receives the return values
at ebp + 12 we have s1.a
at ebp + 16 we have s1.p

(c) (5 points) Describe the general strategy illustrated here for passing a structure as a parameter to a function.

Answer:

The values of the fields of the structure are placed, in reverse order, on the stack in the frame of the calling function.

This is clearly analogous to the way a sequence of simple parameters are passed via the caller's stack frame.

(In this case, the mechanism is adjusted because the return value is also a structure.)

(d) (5 points) Describe the general strategy illustrated here for handling a structure as a return value from a function.

Answer:

Space for the returned structure is provided within the caller's frame.

A pointer to the first word of that space is provided to the called function on the stack, within the caller's frame, placed as if it were the first parameter to the called function (i.e., after the values of the actual parameters).

The called function can then use that pointer to set the fields of the structure, within the caller's frame, before the called function returns.

II. Optimization (16 points)

Consider the following C function `f1()` and its compiled code:

```
// Preconditions:
//       px, py and pz are not NULL
//       px != py
//

void f1(int *px, int *py, int *pz)
{

   *py = *py + *px;
   *py = *py - *px;
   *pz = *py;
}
```

```
f1:
   pushl %ebp
   movl  %esp, %ebp
   movl  12(%ebp), %edx
   movl  8(%ebp), %ecx
   movl  (%edx), %eax
   addl  %eax, %eax
   addl  (%ecx), %eax
   movl  %eax, (%edx)
   subl  (%ecx), %eax
   movl  %eax, (%edx)
   movl  16(%ebp), %edx
   movl  %eax, (%edx)
   popl  %ebp
   ret
```

The gcc compiler, even with the switch –O2, does not apply a seemingly obvious optimization to `f1()`.

(a) (4 points) What is this "seemingly obvious optimization"?

Answer:

Naively, the effect of the function seems to be to make two changes to *py that cancel each other out, and then assign the original value of *py to *pz. So the obvious optimization would be to eliminate everything except the final assignment to *pz. Hence we would have seen only fetches of py and *py and pz, and then a write of *py to *pz.

(b) (6 points) Explain why gcc cannot apply it in this case.

Answer: gcc has no way to know that px and py do not point to the same target. If so, then the result of the function would be to reset *py to 0 and assign that value to *pz.

(c) (6 points) The ISO-C99 Standard added the keyword `restrict` to the language. `restrict` can only be applied in declarations of pointer variables, and means that the pointer to which it is applied is the only means of accessing the data to which it points within the scope in which that pointer is declared. Could the use of `restrict` alter gcc's behavior in part (a)? If so, explain minimal changes you could make involving the use of `restrict`. If not, explain why not.

Answer:

The best answer is no, for a subtle reason. The precondition states that `px != py`, which implies that they cannot be aliases. Unfortunately, nothing is said about relationships involving `pz`. We could still have either `px` or `py` equaling `pz`, just not both.

That means we do not have enough information to justify placing restrict on any of the three pointers, and therefore, no way to enable the optimization.

(That was an error in the statement of the problem; I intended to modify the precondition.)

The less good answer is: change the function interface to:

```
void f1(int *px, int* restrict py, int *pz);
```

or

```
void f1(int* restrict px, int *py, int *pz);
```

Either would be sufficient to allow the optimization described above to be performed. Either way, `px` and `py` cannot alias the same target, and that eliminates the scenario described previously.

Notes:

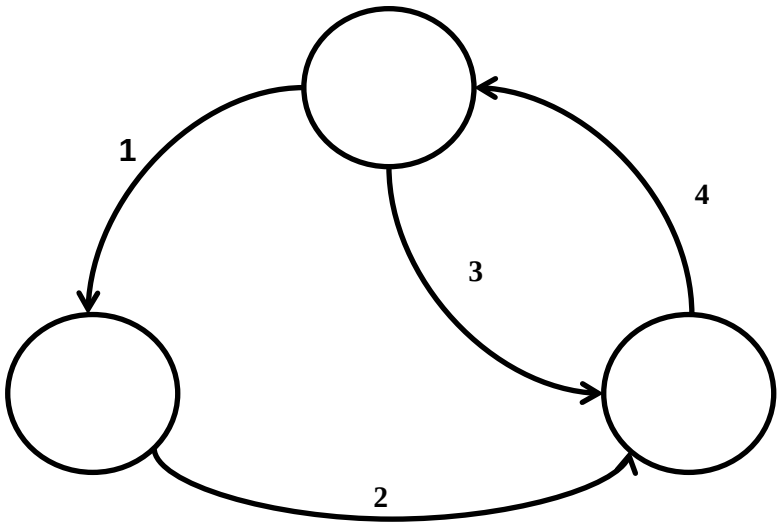
It would make no difference to the final state if `px` or `py` aliased the same target as `pz`, since the last step in the execution is to assign a value to `*pz`, and that value is independent of whatever value `*pz` may have held previously.

Moreover, the application of `restrict` to `px` or `py` is (partially) justified by the precondition for the function; we have no information that would justify applying `restrict` to `pz`.

III. Processes and Signals (40 points)

(a) (13 points) Interpret the diagram below as one showing the states of a process or thread and the transitions between these states.

1. (3 points) Label the diagram with the appropriate state names choosing from among these names: priority, running, ready, signaled, swapped, blocked, executing, killed, runnable, empty.



Answer: Top state: running or executing; Lower left state: blocked, Lower right state: ready or runnable

2. (6 points) Fill in the following table with descriptions of actions that cause the indicated transitions. In the column labeled process, name a system call that may cause transition 1 and describe the condition related to this system call that causes transition 2. In the column labeled signal, name signals which, when received by a process, cause transitions 1 and 2. For the column labeled thread, name thread-related operations that cause transitions 1 and 2.

Transition	process	signal	thread
1	**read, or any other blocking system call**	**Sigstop or other signal that causes process to block**	**P, mutex.wait, or sem.wait**
2	**Description matching system call given in answer for transition 1**	**Sigcont or other signal that causes resumption**	**V, mutex.post, or sem.signal**

Answer: in table.

3. (1 point) Give an example of a situation in which transition 3 occurs.

Answer: end of scheduling interval, pause system call, or any equivalent

4. (1 point) What system/kernel component controls which process makes transition 4.

Answer: scheduler (or equivalent name)

5. (2 points) Give two sequences of transitions that occur during a context switch.

Answer: 1, 4 and 3,4

(b) (12 points) Write the key elements of the C code needed or the following programs. Error checking is NOT required.

1. (6 points) Write the code for a program that (1) creates a separate process to execute the program in the file "genfile", (2) waits for 10 seconds, (3) sends a SIGUSR1 signal to the separate process, and (4) obtains and prints the exit status of the separate process. Use the back of this page if needed.

```
int child = fork();
if (child ==0)
    exec("genfile",…);
sleep(10);
kill(child,SIGUSR1);
int status;
waitpid(child, &status, NULL);
printf("Child exit status &d \n\r", status);
```

2. (6 points) Write the code for the program in "genfile" that (1) repeatedly executes the function "compute()", (2) upon receiving the SIGUSR1 signal executes the function int finalize(), and terminates with an exit status equal to the return value from the finalize function.

```
int done = 0;

void handler(int signum) {
done = 1;
}

main() {

signal(SIGUSR1, handler);

while(! done) compute();  int status = finalize();
exit(status);
}
```

```
Also may use sigaction variation.


void handler(int signum, siginfo_t info, void* c) {
 done = 1;
}

and

struct sigaction sa;
sa.sa_sigaction = handler;
sigaction(SIGUSER1, &sa, NULL);
```

(c) (5 points) Use the process information in the following table to answer the questions below.

Process ID (pid)	Parent Process ID (ppid)	Group ID (gid)
2218	2200	2218
2220	2218	2220
2222	2220	2220
2251	2218	2251
2252	2251	2251
2253	2251	2220

1. Identify the name and pid of the receiver of all signals, if any, that could be sent if a SIGSTP signal is sent to process 2251. Do not identify the initial SIGSTP signal itself. If no signals are sent answer "NONE".

Answer: SIGCHLD sent to 2218

2. Identify the name and pid of the receiver of all signals (including SIGSTP), if any, that could be sent if a SIGSTP signal is sent to process group 2220. If no signals are sent answer "NONE".

Answer: SIGSTP sent to 2220, 2222, 2253 and SIGCHLD sent to 2218, 2220, 2251

3. If the process group controlling the terminal is 2220 identify the name and pid of the receiver of all signals, if any, that could be sent if process 2222 reads from the terminal. If no signals are sent answer "NONE".

Answer: NONE

4. If the process group controlling the terminal is 2220 identify the name and pid of the receiver of all signals, if any, that could be sent if process 2252 reads from the terminal. If no signals are sent answer "NONE".

Answer: SIGTTIN sent to 2252 and SIGCHLD sent to 2251

5. Identify the name and pid of the receiver of all signals, if any, that could be sent if process 2253 terminates normally. If no signals are sent answer "NONE".

Answer: SIGCHLD sent to 2251

(d) (5 points) Put an "X" in the appropriate column to indicate if each of the following statements is True or False.

Statement	True	False
(a) The effect on processor design of Moore's Law ended in approximately 2005 when processors were no longer doubling in speed every 18 months.		**x**
(b) Multiple threads may both read and write the same global memory.	**x**	
(c) Multiple threads may read but not write the same global memory.		**x**
(d) A context switch from Process A to Process B implies that a mode switch also occurred.	**x**	
(e) A mode switch implies that a context switch will also occur.		**x**
(f) A signal blocked by a process will be marked as "pending" by the kernel and delivered to the process later.	**x**	
(g) The kernel keeps a queue of pending signals to deliver to a process when the process unblocks the signal.		**x**
(h) The fork system call creates a new process running the same program as the original process.	**x**	
(i) The exec system call creates a new process running a program different from the original process.		**x**
(j) The return value from a fork system call is always a process id.		**x**

(e) (5 points) Shown in the left column is code that creates and manipulates file descriptors. For each read/write operation shown in the middle column indicate in the right column the data stream affected by the operation. If the read/write operation is erroneous, write ERROR. Assume that all standard streams are open, that file descriptors are assigned in sequential order, and that no other file descriptors are used except those shown.

code	read/write operation	affected stream

`int f1, f2, fd[2];`
 	
`f1 =open("file1",O_RDWR);`
`f2 =creat("file2",S_IRWXU);`
`pipe(fd);`
 	
`dup2(3,0);`
`dup2(5,3);`
`close(5);`
	`read(0,…)`	**file1**

	`write(0,…)`	**file1**
	`write(1,…)`	**STDOUT**
	`write(2,…)`	**STDERR**
	`read(3,…)`	**pipe**
	`write(3,…)`	**ERROR**
	`write(4,…)`	**file2**
	`write(5,…)`	**ERROR**
	`read(6,…)`	**ERROR**
	`write(6,…)`	**pipe**

IV. Threads and Synchronization (15 points)

(a) (10 points) Shown below is the code, including synchronization using Dijkstra's semaphores, for thread 1 in an application that has 3 threads. Also shown is the code with no synchronization for threads 2 and 3. Add the semaphore operations needed in threads 2 and 3 to be correctly synchronized with thread 1. Assume that all variables are declared as global ints, all functions are correctly used, and the semaphores S and T are Boolean semaphores (i.e., the internal value is initialized to 1).

Thread 1	**Thread 2 (5 points)**	**Thread 3 (5 points)**

`P(S);`
`y = 0;`
`x = f(y);`
`z = g(x);`
`V(S);`
 	
…
 	
`P(T);`
`a = 0;`
`b = g(a)`
`c = h(a) + b;`
`V(T);`
	`P(S)`
	`y = 0;`
	
	`z = h(y);`
	`V(S)`
	
	`d = 6;`
	
	`e = f(d);`
	
	`j = g(e);`
		
		`n = 15;`
		
		`P(S)`
		`x = f(n);`
		
		`P(T)`
		`c = g(x);`
		`V(S)`
		
		`p = 7;`
		
		`r = g(p);`
		
		`b = g(p);`
		`V(T)`

Answer: Semaphore code shown in table above.

(b) (5 points) A concurrent programming language has the constructs

```
block <blockname> {…}
{before + after} <blockname> do {…}
```

that names a block of code and specifies whether a block should be executed by one thread before or after another block of code is executed by a different thread. Show the pattern of the code that a compiler could generate for blocks A and B to enforce the meaning of the before/after constraints in the following cases.

(a) `before B do { code for block A }`

```
…
block B {code for block B}
```

Answer:

```
semaphore S(0); //semaphore initialized to 0

code for block A;
V(S);

P(S);
code for block B;
```

(b) `after B do { code for block A }`

```
….
block B { code for block B }
```

Answer:

```
semaphore S(0); //semaphore initialized to 0

code for block B;
V(S);
V(S);

P(S);
code for block B;

P(S);
code for block C;
```