Full Name: ______________________________________

'On my honor as a University of Colorado at Boulder student I have neither given nor received unauthorized assistance on this work.'

CSCI 2400, Sample Summer 2013

Final Exam

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name on the front. Put your name or student ID on each page.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 100 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- You can not use a computer or calculator. Good luck!

Problem	Page	Possible	Score
1	1	10	
2	2	20	
3	4	20	
4	5	20	
5	9	15	
6	13	15	
Total		100	

1. **[10 Points]** This problem tests your understanding of process creation and event ordering

```
int counter = 2;
int main()
{
  int i;

  if ( fork() == 0 ) {
    for(i = 0; i < 2; i++) {
      fork();
      counter++;
      fprintf(stderr,"%d", counter);
    }
  }
  else{
   counter--;
  }
  fprintf(stderr,"%d", counter);
}
```

What outputs are possible? Circle the possible outputs in the list below.

(a) 13344344

(b) 13443444444

(c) 14343444344

(d) 34444134444

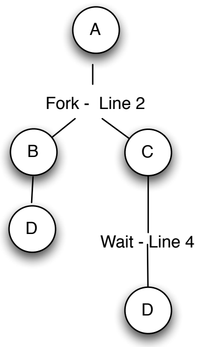
(e) 13443444444

2. **[20 Points]** This problem tests your understanding of process creation and event ordering

```
 1  void P(char *s) { fprintf(stderr,"%s ", s); }
 2
 3  main()
 4  {
 5    int status;
 6
 7    P("B");
 8    if (fork() ) {
 9      P("L");
10      if ( fork() == 0 ) {
11        P("G");
12      } else {
13        P("S");
14        wait(&status);
15      }
16    } else {
17      P("X");
18      if ( fork() ) {
19        P("W");
20        wait(&status)
21        P("Z");
22      } else {
23        P("T");
24        if (fork()) {
25          P("H");
26        } else {
27          P("M");
28        }
29        exit(0);
30      }
31    }
32    P("R");
33  }
```

(a) **[11 Points]**

Draw a "process tree" diagram. Each `fork()` in the program should cause a fork in the tree labeled with the line number of the `fork()`. The output of print statements should be shown linearly beneath the tree in the order they would be output. `wait()` statements should also be included in the tree.
Model your diagram after the diagram on the right, which shows a program that prints "A", forks a child using an —verb—if(fork())—, has the child output "B" and the parent output "C" and then waits for the child; following the if-then-else is a statement to print "D".



(b) **[2 Points]** Line 14 contains a `wait` call. Other lines (8, 10, 18, 24) contain fork() statements that spawn child precesses. What are the line numbers of the corresponding `fork()` calls that create processes that might be successfully harvested by the `wait()` statement on line 14?

(c) **[2 Points]** Repeat part the previous problem for the `wait()` on line 20?

(d) **[2 Points]** If the program is executed, can W ever appear before H?

(e) **[2 Points]** Can Z ever appear before W?

(f) **[2 Points]** Can R ever appear before L?

3. **[20 Points]**

```
int pid = 0;

void sigchild(int s)
{
  if (pid) kill( pid, SIGHUP );
}

void sighup(int s)
{
  if (pid) kill( pid, SIGHUP );
}

main()
{
  int status;
  signal(SIGCHLD, sigchild);
  signal(SIGHUP,  sighup);

  int i;
  for (i = 0; i < 2; i++) {
    if ( (pid = fork()) == 0 ) {
      KILL( getppid(), SIGHUP ); // kill parent
      exit(0);
    }
  }
  wait(&status);
}
```

In this program, a parent process (P) forks two child processes C_1 and C_2.

(a) **[4 Points]** What processes can succesfully send a SIGHUPsignal to C_1?

(b) **[4 Points]** What processes can succesfully send a SIGHUP signal to C_2?

(c) **[4 Points]** What is the maximum number of times the SIGHUP signal handler might be invoked by P? Explain.

(d) **[4 Points]** Can the `kill()` in the SIGCHLD handler ever successfully signal C_1? Why or why not.

(e) **[4 Points]** What is the maximum number of times the SIGHUP signal handler might be invoked by C_2? Explain how.

4. **[15 Points]**

The following problem concerns the way virtual addresses are translated into physical addresses.

- The memory is byte addressable.
- Memory accesses are to 4-byte words.
- Virtual addresses are 18 bits wide.
- Physical addresses are 16 bits wide.
- The page size is 4096 bytes.
- The TLB is 4-way set associative with 16 total entries.

In the following tables, **all numbers are given in hexadecimal**. The contents of the TLB and the page table for the first 32 pages are as follows. Page table entries marked "valid" specify the physical page number; those marked "invalid" specify the page number on disk.

TLB			
Index	Tag	PPN	Valid
0	3	B	1
	–	–	0
	8	3	1
	–	–	0
1	3	0	1
	–	–	0
	7	E	1
	B	1	1
2	–	–	0
	–	–	0
	F	8	1
	7	5	1
3	7	3	1
	–	–	0
	–	–	0
	–	–	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	7	1	10	6	0
01	8	1	11	7	0
02	9	1	12	8	0
03	A	1	13	3	0
04	6	0	14	D	0
05	3	0	15	B	0
06	1	0	16	9	0
07	8	0	17	6	0
08	2	0	18	C	1
09	3	0	19	4	1
0A	1	1	1A	F	0
0B	6	1	1B	2	1
0C	B	1	1C	0	0
0D	0	0	1D	E	1
0E	E	0	1E	5	1
0F	D	1	1F	3	1

- Part 1

(a) The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn't exist, don't draw it on the diagram.)

VPO	The virtual page offset
VPN	The virtual page number
TLBI	The TLB index
TLBT	The TLB tag

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(b) The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

PPO	The physical page offset
PPN	The physical page number

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Part 2

  For the given virtual addresses, indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs.

  If there is a page fault, enter "-" for "PPN" and leave part C blank.

  Virtual address: `1DE3C`

  (a) Virtual address format (one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

  (b) Address translation

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

  (c) Physical address format (one bit per box)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Virtual address: `0EA48`

(a) Virtual address format (one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(b) Address translation

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

(c) Physical address format (one bit per box)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

5. **[15 Points]** Computational weather models use a volume of $N \times N \times N$ "voxels" to represent the air column. The code below models air temperature. The model use iterative algorithms that approximate air temperature based on the tempreture of adjacent voxels – the temperature of a given voxel is a function of the voxels surrounding it on all faces and diagonally. The temperature in a voxel is stored as a double precision floating point value. The following is the most time-consuming code for the weather model:

```
double v[4][4][128] = .....;

for (int lat = 1; lat < cols-1; lat++) {
  for (int lon = 1; lon < rows-1; lon++) {
    for (int alt = 1; alt < 127; alt++) {

      v[lat][lon][alt] = f(
                                v[lat-1][lon-1][alt-1],
                                v[lat-1][lon-1][alt  ],
                                v[lat-1][lon-1][alt+1],

                                v[lat-1][lon  ][alt-1],
                                v[lat-1][lon  ][alt  ],
                                v[lat-1][lon  ][alt+1],

                                v[lat-1][lon+1][alt-1],
                                v[lat-1][lon+1][alt  ],
                                v[lat-1][lon+1][alt+1],

                                v[lat  ][lon-1][alt-1],
                                v[lat  ][lon-1][alt  ],
                                v[lat  ][lon-1][alt+1],

                                v[lat  ][lon  ][alt-1],
                                v[lat  ][lon  ][alt  ],
                                v[lat  ][lon  ][alt+1],

                                v[lat  ][lon+1][alt-1],
                                v[lat  ][lon+1][alt  ],
                                v[lat  ][lon+1][alt+1],

                                v[lat+1][lon-1][alt-1],
                                v[lat+1][lon-1][alt  ],
                                v[lat+1][lon-1][alt+1],

                                v[lat+1][lon  ][alt-1],
                                v[lat+1][lon  ][alt  ],
                                v[lat+1][lon  ][alt+1],

                                v[lat+1][lon+1][alt-1],
                                v[lat+1][lon+1][alt  ],
                                v[lat+1][lon+1][alt+1]
                              );

    }
  }
}
```

You should assume: **double** takes 8 bytes; you should ignore the store / memory writes – only consider loads ; The array 'v' starts at address 0; memory addresses are 32 bits long; all scalars (lat, lon, alt and temporary values) are held in registers.

The code is run on a computer with 4096 byte pages, a 32-entry TLB with 8 sets of 4 entries in each set (*i.e.* 4-way associative).

Below, list the address for the first 15 READ or LOAD references. and indicate if it is a hit or miss in the TLB. It is easiest to express the memory address as ______$1024 +$ ____. Use decimal numbers throughout. It's easy to to first write down the array entry (*e.g.* `m[1][2[3]]`), translate that to an address and then figure out the hit or miss.

[7 Points]

Ref #	Array Entry	Address		TLB Miss?
		$\times 1024$	+	
0				
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				

[4 Points]

Below, draw a diagram to show the state of the TLB at the end of the references above. You should clearly distinguish each set. For each TLB entry, you should indicate if the entry is valid and the appropriate **starting memory address** for that page (if valid). If the entry is not valid, put a dash in the entry (we're ignoring the valid bit in this example). Rather than showing the "tag bits", which are harder to compute, indicate the *starting memory address of the page, using the notation as above*. All numbers must be decimal.

[4 Points]

Assume the loops are executed as originally shown (*i.e.*)

```
  double v[4][4][128] = .....;

  for (int lat = 1; lat < cols-1; lat++) {
    for (int lon = 1; lon < rows-1; lon++) {
      for (int alt = 1; alt < 127; alt++) {
...
```

How many TLB misses would occur when that code is executed, assuming that's the only code that executed and the TLB was initially empty? You must show or work or reasoning rather than just jot down a number.