Instructions:

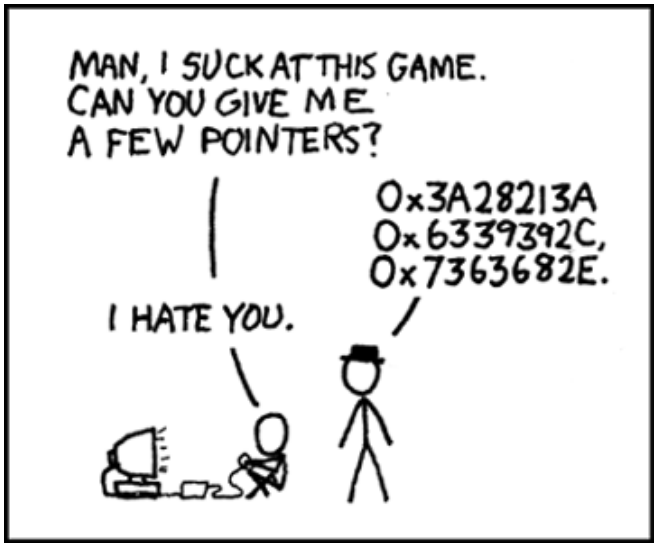
- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet.
- No calculators or other computing devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test and your test form will be collected immediately.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 7 questions, some with multiple parts, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page, sign your fact sheet, and turn in the test and fact sheet.
- Note that failing to return this test, and discussing its content with a student who has not taken it are violations of the Honor Code.

Do not start the test until instructed to do so!

Name Solution
printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

signed


MAN, I SUCK AT THIS GAME.
CAN YOU GIVE ME
A FEW POINTERS?
0x3A28213A
0x6339392C,
0x7363682E.
I HATE YOU.

xkcd.com

I. Processes 26 points

1. Suppose the following C code is executed; assume appropriate `wait()` calls occur in the missing part of the code. The executables `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, `wait`s for any children it has forked, and then terminates.

```
int main() {

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

   printf("%d is entering fork01.\n", getpid());
   int index = 0;

   int pid = fork();

   pid = fork();

   if ( pid == 0 && index == 0 ) {

      index++;

      pid = fork();

      if ( pid == 0 ) {
         execve("progA", argv1, env);
      }
   }
   if ( pid != 0 && index != 0 ) {

      pid = fork();

      execve("progB", argv2, env);
   }
   . . .
   printf("%d is exiting from fork01.\n", getpid());
   return 0;
}
```

One key is to remember that, until it execs, a child process operates in a clone of the parent's memory space. Therefore, variables like index and pid are not shared between parent and child.

Another is to realize the code is deterministic, so you don't have to worry about the order in which the various processes are scheduled to run.

After that, it's just a matter of tracing the execution, beginning with the original process.

To get you started, the original process forks() two child processes, at which point it sees pid != 0, and index == 0, and so enters neither if-statement. Hence the original process exits from the original program…

a) [4 points] Which program is the original process executing when that process terminates?

fork01 (the original program shown above) – it does not enter either if statement.

b) [10 points] How many processes terminate in each of the relevant programs?

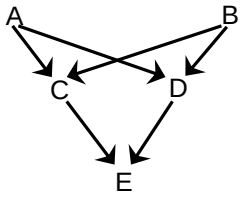
Program	# of processes that terminate there
`fork01`	**2**
`progA`	**2**
`progB`	**4**

2. [12 points] Circle the appropriate classification for each of the following statements:

A `fork`'d child process that does not call `exec()` automatically shares global variables with its parent process. **The memory space of a fork'd child is a clone of the parent's.**	true	(false)
If a shell `fork`s a background child process, the shell still needs to wait until the child terminates before terminating itself. **The parent should wait for and reap child processes, background or not.** **The phrase "needs to" may have been confusing. If you answered "false" but addressed the fact that there could be undesirable consequences, I gave credit for that.**	(true)	false
A call to `exec()` creates a new process, but does not trigger an immediate context switch to that process. **Calling exec() does not create a new process; it loads a new program into the calling process.**	true	(false)
If a process calls `fork()` to create a child process, but the parent process terminates before the child process calls `exec()`, the child process will be forced to terminate as well because the parent's memory space, including its text segment, will be reclaimed by the OS when the parent terminates. **This is straight from the discussions about process management.**	true	(false)
When a process *P* sends a signal, and the receiving process *Q* is in a blocked state, the signal is lost (never received by *Q*). **If more than one signal of the same type are sent, some may be lost since signals are not queued, but the first such signal will remain.**	true	(false)
If a process *P* is currently executing a user-defined signal handler for signal type *S*, then signals of other types or of that type sent to *P* are not automatically blocked until the current signal handler completes execution. **This is straight from the requirements for the esh.**	(true)	false

II. Multi-threading 22 points

3. **[16 points]** The diagram at right defines a set of dependencies among five tasks, each of which will be carried out by a different thread. For example, task C depends on both task A and task B.



Complete the code given below so that the dependencies will be satisfied, and no additional constraints are imposed.

```
// Declare any global semaphore and/or mutex variables here.
// Be sure to show how they should be initialized.

semaphore Adone = 0, Bdone = 0, Cdone = 0, Ddone = 0;
```

```
static void* thread_A(void* x) {

   printf("A\n");
   post(Adone);
   post(Adone);

}
```

```
static void* thread_B(void* x) {

   printf("B\n");
   post(Bdone);
   post(Bdone);

}
```

```
static void* thread_C(void* x) {

   wait(Adone);
   wait(Bdone);
   printf("C\n");
   post(Cdone);

}
```

```
static void* thread_D(void* x) {

   wait(Adone);
   wait(Bdone);
   printf("D\n");
   post(Ddone);

}
```

```
static void* thread_E(void* x) {

   wait(Cdone);
   wait(Ddone);
   printf("E\n");



}
```

Condition variables were explicitly disallowed.

Mutex-based solutions are inherently unnatural, since the point is to establish precedence constraints, not to enforce mutual exclusion.

```
int main() {

   pthread_t t[5];

   pthread_create(t+4, NULL, thread_E, NULL);
   pthread_create(t+3, NULL, thread_D, NULL);
   pthread_create(t+2, NULL, thread_C, NULL);
   pthread_create(t+1, NULL, thread_B, NULL);
   pthread_create(t+0, NULL, thread_A, NULL);
   . . .
}
```

4. [6 points] Consider the following C program, which spawns off two threads:

```
static int x = 0;
int y = 0;
. . .

static void* f(void* p) {

   int* param = (int*) p;
   . . .
   return NULL;
}

int main() {

   int A = 10;
   int B = 20;

   pthread_t t[2];
   pthread_create(t+0, NULL, f, (void*) &A);
   pthread_create(t+1, NULL, f, (void*) &B);

   pthread_join(t[0], NULL);
   pthread_join(t[1], NULL);

   printf("x = %d\n", x);
   exit(0);
}
```

Aside from any variables declared in the function `f()`, which of the variables declared in the program could be accessed by the first thread that is spawned?

Each thread will have its own stack, so local automatic variables are not shared. However, global variables are shared, as are the targets of pointers passed to the thread. So...

A, x, and y

Of course, the thread will also have access to any local variables declared within the function(s) it runs, so you could include:

p and param

II. Linking and Loading 32 points

5. Consider the short C program, consisting of two source files, shown below:

```
    // a.c
A1  int a = 1;
A2  int b;
A3  extern int c;

A4  int main() {

A5      printf("b = %d\n", b);
A6      int r = 10;
A7      b = f(r, a);
A8      c = f(a, b);

A9      printf("b = %d\n", b);
A10     printf("c = %d\n", c);
A11     return 0;
    }
```

```
    // b.c
B1  int b = 2;
B2  int c = 3;
B3  static int d = 42;

B4  int f(int x, int y) {
B5      static int e = 10;
B6      e = e - x + y;
B7      if ( e > 0 )
B8          e = d + c;
B9      return e;
    }
```

The given C code, without any additional code, compiles (with three warnings), links, and executes with no runtime errors.

a) [4 points] What value will be printed by the statement in line *A5*? Why?

The strong definition of "b" in line B1 will override the weak declaration in line A2, so the printf() statement in line A5 will print:

b = 2

b) [4 points] Compilation/linking succeed if the `extern` shown above is present, but not if it is omitted. However, compilation/linking succeed even without any declaration (in `a.c`) for the symbol `f`. Why?

Function names are handled differently than variable names. For the call to f() in line A7, the complier will generate an implied declaration, which happens to agree with the actual interface of f() in line B4, so the linker will match the call to that definition and all is well.

But the compiler will not generate an implied definition for a variable. Since the symbol c is not defined in A, without an extern declaration, the compiler would view line A8 as a reference to an undefined symbol.

c) [12 points] Exactly which symbols will require relocation by the linker? Cite the definitions of the variables, writing your answers in the form `Line:Variable`; e.g., `C17:foo`. Be complete.

A1: a
A5: printf
B1: b
B2: c
B3: d
B4: f
B5: e

d) [2 points] A student in CS 2506 writes the following rule in a makefile:

```
a.o : a.c b.o
     gcc -c -O0 -ggdb3 -Wall a.c b.o
```

When "`make a.o`" is run the following message is generated:

```
gcc: warning: b.o: linker input file unused because . . .
```

Why is this warning generated?

Because the -c switch specifies that compilation, but not linking is to be carried out. Even if a.c contains references to symbols defined in b.o, those references are not resolved until linking occurs.

---

6. [10 Points] Examine the code below; note each line is numbered.

```
static int A = 2;                          //  1
static int B;                              //  2
int C = 20;                                //  3
int D;                                     //  4

void func(int X) {

   int  r = A * X;                         //  5
   int* list = malloc( A * sizeof(int) );  //  6
   B = r * r + C;                          //  7
   list[0] = list[0] + C;                  //  8
   D = list[0] + list[1];                  //  9
}
```

During linking/loading, different parts of the code above are mapped to different memory sections in the address space of the program. In the table below, place an X whenever the given line of code is related to the corresponding memory section. (It's possible for a single line to relate to more than one memory section.) Assume that function parameters are passed in registers, as in the usual x86-64 protocol.

Line	BSS	DATA	STACK	HEAP
1		X (A)		
2	X (B)			
3		X (C)		
4	X (D)			
5		X (A)	X (r, X)	
6		X (A)	X (list)	X (malloc)
7	X (B)	X (C)	X (r)	
8		X (C)	X (list)	X (list[0])
9	X (D)		X (list)	X (list[0, 1])

IV. Explicit Memory Management 20 points

7. Consider the design of an explicit memory management system, as in the malloclab project.

a) [8 points] Suppose an explicit list is used to organize the free blocks. Would coalescing blocks on a call to free() be simpler if the explicit list organized the free blocks in ascending order by address? Justify your answer.

On a call to free(), we have a pointer to the payload within an allocated block, from which we can obtain pointers to the boundary tags in the preceding and following blocks on the heap (which is, of course, in physical order). The existence of an explicit free list is not necessary in order to do this.

One could argue that the use of an explicit free list actually makes the process more complex, not less complex, since the resulting free block must be integrated into the explicit list by setting and modifying a number of explicit pointers.

b) [6 points] Suppose that a `malloc()` call occurs, requesting a block of *N* bytes, and the allocation is going to be served by using a currently free block which is large enough to hold a payload of *N* + *M* bytes (plus whatever overhead the implementation imposes). Briefly discuss the pros and cons of splitting the free block into a new free block and an allocated block, versus simply allocating the entire free block to satisfy the request.

Splitting requires more computation, and hence more time. Not splitting means that we have made N bytes of memory available to a client who did not request them, and so will not use them, and therefore made those N bytes unavailable for satisfying future requests from other clients.

I expected a good answer to address at two different pros/cons of splitting vs not splitting.

c) [6 points] Explain the difference between *internal fragmentation* and *external fragmentation*.

Internal fragmentation consists of memory that is allocated to a process but that cannot be used by that process (e.g., the "slop" that results when we do not split).

External fragmentation consists of memory that is not allocated to any process, but that may be relatively unusable to satisfy future allocation requests (e.g., small "slivers" of free space that result from splitting). In other words, it consists of small blocks of free memory that are discontiguous and cannot be coalesced.