Sample Midterm (Fall 2014)

Solutions are shown in this style. This exam was given Fall 2014.

1. Compiling and Linking (25 pts)

a) (12 pts) *Separate Compilation.* Consider the following two .c files, which both include the same .h file:

```
/* link.h */
extern int v[], w[], u[];
void add_u_and_v(void);
```

```
// link1.c
#include "link.h"

int w[2], v[2], u[2];
void add_u_and_v() {
    w[0] = u[0] + v[0]; // stmt 3
    w[1] = u[1] + v[1]; // stmt 4
}
```

```
// link2.c
#include "link.h"
#include <stdio.h>

extern int u[3], v[3], w[3];

int
main()
{
    v[0] = 1, v[1] = 2, v[2] = 3;   // statement 1
    u[0] = 4, u[1] = 5, u[2] = 6;   // statement 2
    add_u_and_v();
    printf("%d %d %d\n", w[0], w[1], w[2]);
    printf("v is at %p\n", v);
    return 0;
}
```

A user tries to build this obviously wrong program like so:

```
$ gcc -Wall link1.c link2.c -o link
```

(i) (2 pts) Will there be any warnings?

No, not even with –Wall. The code of each .c file is entirely correct C code.

(ii) (10 pts) Once built, the program outputs:

```
$ ./link
10 7 4
v is at 0x600988
```

From this output, infer where the linker allocated the variables v, w, and u! Assume sizeof(int) == 4! Describe the working of this program by sketching how the values it outputs are computed!

(6 pts)	*Create as many columns in this table as there are different memory locations involved. Use C syntax for symbolic names, e.g. v[0]*						
symbolic names from the perspective of link1.c	*w[0]*	*w[1]*	*u[0]*	*u[1]*	*v[0]*	*v[1]*	
symbolic names from the perspective of link2.c	*w[0]*	*w[1]*	*w[2]/ u[0]*	*u[1]*	*u[2]/v[0]*	*v[1]*	*v[2]*
virtual addresses in 0x….. notation	*0x600978*	*0x60097C*	*0x600980*	*0x600984*	*0x600988*	*0x60098C*	*0x600990*
(4 pts)	*Below, fill in the values that are stored in memory after each statement (refer to source code), in the columns you've created*						
after statement 1					*1*	*2*	*3*
after statement 2			*4*	*5*	*6*		
after stmt 3	*10*						
after stmt 4		*7*					

As you can see, the different views of the two separately compiled units resulted in v[2] being written past the end of the allocated memory, and in completely unintended results being computed and output.

b) (5 pts) Using static. Consider the following program:

```
// statics.c
static int c;
void f(int v) {
    static int c;
}

void g(int v) {
    static int c;
}
```

```
$ gcc –c statics.c
$ nm statics.o
0000000000000000 b c
0000000000000008 b c.1595
0000000000000004 b c.1599
0000000000000000 T f
0000000000000009 T g
```

When compiled the symbol table shown in the right column results.

(i) (3 pts) Why did the compiler introduce symbols `c.1595` and `c.1599`?

All three static variables 'c' are distinct and need separate storage, so the compiler appended some running count to produce different symbols. Generally, symbols defined outside of functions will appear with the C-name in the symbol table (on some systems, with a _ prefix).

(ii) (2 pts) Why does it appear that 'c' is allocated at address 0 (`0000000000000000`)?

This is the symbol table shown after compilation, before linking. The assembler lays out each .o section's symbols using consecutive addresses, starting from 0 since it does not know yet where the symbol will be ultimately allocated, which is the linker's job.

c) (4 pts) A "best practice" when writing large C programs is to declare variables that are used only within multiple functions of one .c file using the `static` modifier. What is the rationale behind this recommendation?

To avoid namespace pollution. If not declared static, the name will be visible to all .o modules, inviting clashes or unintended resolution if they also define variables of that name.

d) (4 pts) Making pie.

```
// mmmh, pie.c
#include <stdio.h>

int data[1];
int
main()
{
    printf("main %p\n", main); // %p means print address of
    printf("data %p\n", data);
}
```

When built and run like so, this output results:

```
$ gcc -pie -fPIC pie.c -o pie
$ ./pie
main 0x2aabb126272c
data 0x2aabb1462b58
$ ./pie
main 0x2abd2bd8672c
data 0x2abd2bf86b58
$ ./pie
main 0x2b3a60c5672c
data 0x2b3a60e56b58
```

(i) (2 pts) Based on this observation, conclude what –pie does!

Ordinarily, executables are linked such that the .text segment and the .data segment (where 'main' and 'data' are allocated, respectively) are at constant addresses determined by the linker. In this example, different addresses are chosen at each run, which means the executable must have been built in a way that allows it to be relocated at runtime and be loaded at different addresses. "PIE" stands for position-independent executable, and it is used for executables that are particularly fortified against exploits.

(ii) (2 pts) If the program had not been built with –pie, what would its output have been?

It would have been the same address each time, and the address would have been in a much lower range of the address space. For instance,

```
$ ./pie.nopie
main 0x4004c4
data 0x6008b0
```

2. Fork() and Exec() (22 pts)

a) (14 pts) While practicing for the midterm, a CS 3214 student experimented with the fork() system call. As you know, fork() creates new processes that execute concurrently, and if multiple CPUs are available, in parallel. Here is the program they came up with, which attempts to parallelize the addition of a file with 1,000,000 numbers, read from standard input. The idea they tried to implement is to use a data-parallel approach, splitting the array into 10 pieces, and have each child process work on one piece, then add together the results.

```
// forkadd.c
#include <stdio.h>
#include <stdlib.h>

#define N 1000000
int
main()
{
    int i, j;
    double *bigarray = malloc(N * sizeof(bigarray[0]));
    for (i = 0; i < N; i++)
        scanf("%lf", bigarray+i);

    double sum = 0.0;
    int children_done = 0;
    // distribute work across 10 child processes
    for (j = 0; j < 10; j++) {
        if (fork() == 0)  {
            // child 'j' sums up interval [N*j/10, N*(j+1)/10-1]
```

```
            double childsum = 0.0;  // use separate var for each child
            for (i = 0; i < N/10; i++)
                childsum += bigarray[j * N/10 + i];

            sum += childsum;        // add to total when done
            children_done += 1;
        }
    }

    // reap children before examining the result
    for (j = 0; j < 10; j++)
        wait(NULL);

    // double check that all children are done, output the result
    if (children_done == 10)
        printf("%lf\n", sum);
    return 0;
}
```

(i) (2 pts) If the compiled program is called '`forkadd`' and the data resides in a file called '`big.in`', what does the user need to type on the command line to execute this program?

The user types

```
$ ./forkadd < big.in
```

(ii) (12 pts) The students show this program to their TA. The TA looks at the program briefly and says this program cannot possibly work. Does this program work? (We define "work" as outputting the correct sum of the numbers in the input file. Ignore possible floating point issues such as overflow or associativity!)
Justify your answer and provide a detailed explanation for why the program works (or does not work)!

The program works – it in fact outputs the correct sum of the numbers in the input file.

The TA was right to be confused, however, because it does not work in the way the writer intended. Unlike when creating threads, fork() creates a child process that is initially a clone of the parent, but which is given a separate copy of all state – including variables such as childsum, children_done, and a copy of everything in bigarray. Updates to those variables will not propagate to the parent in the way the code suggests, so the entire approach is bogus.

Why, then, does it produce the correct result? Note that each child process that is created does not call exit(), but actually continues in the for loop forking more children. Eventually, each child process will exit the loop, call wait(), then check

its copy of children_done and exit silently if it is not 10. (Note that wait(NULL) will not block, but fail silently if a process does not have currently have any unreaped children.)

So, the initial process spawns 10 children, the first child spawns 9 children, the first of which will spawn 8 more, the second 7 more, and so on. The second child spawns 8 children, each of which will produce even more. The total number of processes spawned this way is 2^n = 1024. Only one of those children – the one spawned last - will have children_done set to 10, and the correct sum in 'sum' – it computes the last 1/10th, inherited the result of the first 9/10th from its parent, who in turn inherited the result of sum representing the first 8/10th from its parent, and so on.

Don't believe it? Try it:
http://courses.cs.vt.edu/~cs3214/fall2014/gback/examples/fork/forkadd.c
And the data is available here:
http://courses.cs.vt.edu/~cs3214/fall2014/gback/examples/fork/big.in

b) (8 pts) In the sample midterm exams, a CS 3214 student saw how programs such as `/usr/bin/time` are implemented using the exec() system call. For instance, the following command executes the `/bin/id` program and outputs how long it took.

```
$ time id
uid=979801(cs3214) gid=16151(cs3214) groups=16151(cs3214),16144(cs2505)

real    0m0.002s
user    0m0.000s
sys     0m0.002s
```

The student now attempts to implement a toy version of /usr/bin/time like so:

```
// mytime.c: simplified /usr/bin/time
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int
main(int ac, char *av[])
{
    struct timeval start, end;
    gettimeofday(&start, NULL); // take begin time stamp
    execvp(av[1], av + 1);      // execute program
    wait(NULL);                 // wait for it to finish
    gettimeofday(&end, NULL);   // take end time stamp
    printf("took %ds\n",   // print time taken rounded down to the nearest sec
                end.tv_sec - start.tv_sec);
    return 0;
}
```

i) (4 pts) Will this problem work in the intended way, i.e., to run any program and output how long it took? Justify your answer!

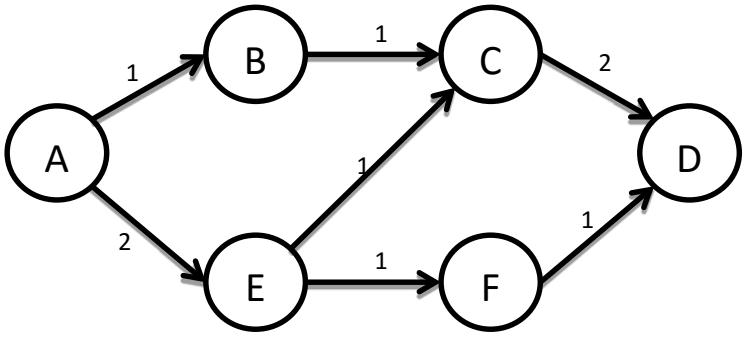
No it won't. exec() replaces the current program with a new program, so the line calling wait() (and everything below it) will not be run if the exec() was successful. To implement time, one needs to fork() and have the child exec() the command to be executed.

ii) (4 pts) What is output when running the program like so?

```
$ ./mytime id
uid=979801(cs3214) gid=16151(cs3214) groups=16151(cs3214),16144(cs2505)
```

It will still exec() the 'id' program so the output will be something like you see above, just no "took …s" message following it.

3. Multithreading (28pts)



a) *(20 pts) Semaphores.* Consider the following dependency graph between 6 hypothetical tasks A through F: In this graph, edges are labeled with numbers that represent how many signals a task must produce before the dependent task can run. For instance, C cannot run until E signals it at least once. Implement this graph using threads and semaphores.
Model the tasks as simple `printf()` statements as shown below, and use semaphores' signaling facilities to implement the precedence constraints!

```
// declare any semaphores here
sem_t ab, ae, ec, bc, ef, fd, cd;
```

```
static void*
thread_A(void *_)
{
    printf("A");
    sem_post(&ab);
    sem_post(&ae);
    sem_post(&ae);
    return 0;
}
```

```
static void*
thread_D(void *_)
{
    sem_wait(&cd);
    sem_wait(&cd);
    sem_wait(&fd);
    printf("D");
    return 0;
}
```

```
static void*
thread_B(void *_)
```

```
static void*
thread_E(void *_)
```

```
{
    sem_wait(&ab);
    printf("B");
    sem_post(&bc);
    return 0;
}
```

```
{
    sem_wait(&ae);
    sem_wait(&ae);
    printf("E");
    sem_post(&ec);
    sem_post(&ef);
    return 0;
}
```

```
static void*
thread_C(void *_)
{
    sem_wait(&bc);
    sem_wait(&ec);
    printf("C");
    sem_post(&cd);
    sem_post(&cd);
    return 0;
}
```

```
static void*
thread_F(void *_)
{
    sem_wait(&ef);
    printf("F");
    sem_post(&fd);
    return 0;
}
```

```
int
main()
{
#define N 6
    int i;
    pthread_t t[N];

    // initialize your semaphores here. For each semaphore, show their initial value!

    sem_t *s[] = { &ab, &ae, &ec, &bc, &ef, &fd, &cd };
    for (i = 0; i < sizeof(s)/sizeof(s[0]); i++)
        sem_init(s[i], 0, 0);

    // ----------------------------------------
    void * (*f[])(void *) = { thread_A, thread_B, thread_C,
                              thread_D, thread_E, thread_F };
    for (i = 0; i < sizeof(f)/sizeof(f[0]); i++)
        pthread_create(t+i, NULL, f[i], NULL);
    pthread_exit(0);
}
```

i) (15 pts) Complete the code! Don't forget to show the initialization of each semaphore!

(please fill answers in table above before/after each printf() statement.)

ii) (5 pts) Among all possible outputs of this program, provide the 3 that are alphabetically first, i.e. the lexicographically smallest!

Since the graph represents a dependency graph, any topological sorting of the graph is a possible output. There are a total of five:

1) *ABECFD*

2) *ABEFCD*
3) *AEBCFD*
4) *AEBFCD*
5) *AEFBCD*

The first three, in lexicographical order, are ABECFD, ABEFCD, and AEBCFD.

b) (8 pts) Condition Variables. Consider the following snippet from a thread pool implementation:

```
// code that removes tasks from global submission queue
pthread_mutex_lock(&globalqueue_lock);
while (list_empty(&globalqueue))
    pthread_cond_wait(&globalqueue_cond, &globalqueue_lock);

// release lock before checking shutdown flag in case it is true
pthread_mutex_unlock(&globalqueue_lock);
if (pool->shuttingdown)
    return;

// now process global task
pthread_mutex_lock(&globalqueue_lock);
struct future *f = list_entry(list_pop_front(&globalqueue),
                              struct future, elem);
// run task
f->result = f->func(f->data);
pthread_mutex_unlock(&globalqueue_lock);
```

There are two concurrency-related defects in this code.
Describe them and discuss their possible impact!

i) (4 pts) Defect I

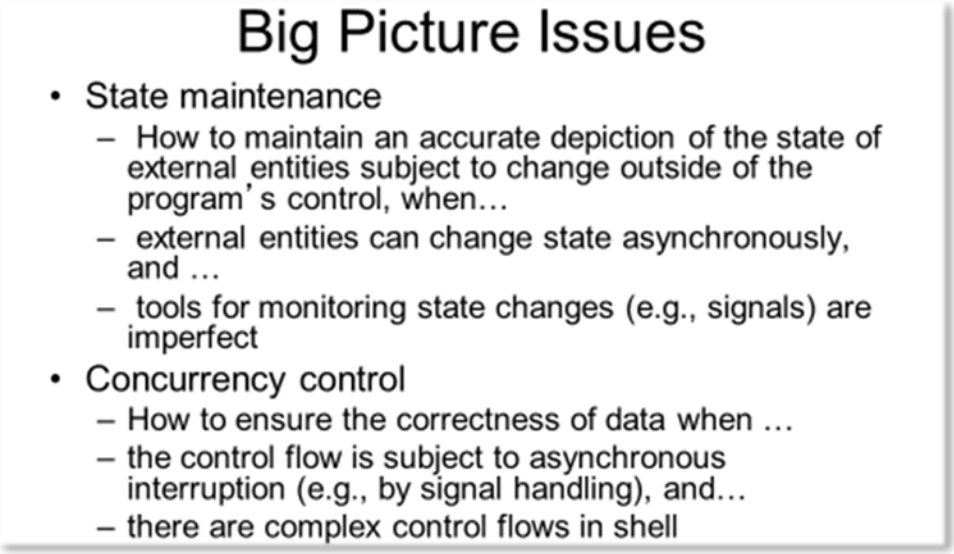
Releasing the lock after checking if the global queue is empty, then reacquiring it later to pop the item from the queue is an atomicity violation. The queue might have been changed by another thread in the interim, even under a correct locking discipline, and might be empty again by the time the item is popped.

ii) (4 pts) Defect II

As those of you who completed the fork/join project will have undoubtedly noticed, holding a lock during the execution of a task results either in deadlock or loss of parallelism since it serializes the execution of all tasks on this lock, so that only one can run at a time.

4. Big Picture Issues (15 pts)

Reflecting on the esh project, you may recall the following lecture slide:



Answer the following questions:

a) (3 pts) What "external entities" that are "subject to change" does this slide talk about?

The processes started by the shell.

b) (3 pts) What "state," specifically, does the shell care about?

It cares if those processes have exited, been terminated with a signal, or stopped due to job control or terminal access requirements. If so, the shell informs the user.

c) (3 pts) Why are changes of this state "outside of the program's control?" (the "program" here refers to the shell.)

The state changes depend on only what the program does, or input by the user (^C, ^Z), neither of which the shell controls.

d) (3 pts) Why are signals an "imperfect" tool?

For at least two reasons: first, they don't queue, which means that multiple children exiting results in only one SIGCHLD, forcing the shell to call waitpid() to figure out which children are affected. Second, once pending, they can't be canceled. So if SIGCHLD becomes pending during a waitpid() call, the child might be reaped there, but the signal will come through and appear spurious.

e) (3 pts) How does a shell ensure the "correctness of its data" in the face of asynchronous interruptions?

A common way is to block those signals (SIGCHLD) during sections of the code in which the delivery of signals could lead to corruptions.

5. Reasoning about Process States (10pts)

Understanding the demand a program imposes on systems resources is a crucial skill for effective application programmers. Common sense dictates that one should design applications that do not create unnecessary processes or threads, but what are the actual resources a system must provide? The answer depends on the state a process or thread is in. Provide examples:

a) (3 pts) If a process is in the BLOCKED or READY state, it occupies the following system resources (name at least 3):

Examples include:

- *Entry in the process table*
- *Some kind of memory to hold their data and code (ignoring details of virtual vs physical for now)*
- *File descriptors referring to kernel objects such as open files, pipes, etc.*
- *Kernel memory to hold their state so that they can be resumed*

b) (2 pts) If a (single-threaded) process is in the RUNNING state, it occupies the following resources in addition:

It also uses a CPU/core.

c) (2 pts) Assume a kernel-level threading implementation. If a new thread is created, the system must allocate the following resources. Do not include resources listed in a) or b) that are part of the containing process.
Name 2:

A new thread needs an entry in the thread/process table as well as space to hold its execution state if it is blocked. It also needs space for its stack. (Actually, two separate stacks are allocated – one for when it runs code in user mode, and one when it runs code in kernel mode.)

d) (3 pts) Can a process figure out what state (RUNNING, READY, BLOCKED) it is currently in?
If so, sketch how. If not, say why not!

A process that's executing code is, by definition, currently in the RUNNING state, so the question has a trivial answer.

That said, if the question had been could a process figure out if it was RUNNING or READY in the past: a process could record timestamps while it

is running and thus create a time line of when it was actually RUNNING, and by inversion conclude that it must have been preempted and in the READY state during those times for which it does not have timestamps. Similarly, a process could time the duration of system calls to infer if it was BLOCKED in those calls.