Sample Midterm (Fall 2011)

This exam was given in Fall 2011.

1. Compiling and Linking (18 pts)

a) (14 pts) *Separate Compilation.* Consider the following two .c files which both include the same .h file:

```
// a.h
(1) static int inc(int x) { return x + 1; }
(2) static int x;
(3) extern int y;        // or empty
(4)        int z;
(5) extern void b(void); // or empty
```

```
// a.c
#include "a.h"
#include <stdio.h>

(6)        int y = 1;
(7) extern int w; // or empty
(8) static int v = 5;

int main()
{
    x = inc(0);
    z += 4;
    b();
    printf("x = %d y = %d z = %d "
           "w = %d v = %d\n",
            x, y, z, w, v);
    return 0;
}
```

```
// b.c
#include "a.h"

(9)  int w = 4;
(10) int v = 5; // or static

void b() {
    x = inc(x);
    y++;
    z--;
    v++;
}
```

When compiled, linked, and executed, the following output results:

```
$ gcc -Wall a.c b.c -o a
$ ./a
x = 1 y = 2 z = 3 w = 4 v = 5
```

Assuming that this program compiled and linked successfully, and based on the output shown above, add `static` and/or `extern` modifiers to the blank lines (1) through (10)! Leave them blank if neither modifier would be appropriate! Any correct solution that results in successful compilation and the output shown above will be accepted.

Explanation:

(1) Needs to be static, or else linking fails due to 2 conflicting strong symbols ‘inc’ with an “already defined” error

(2) x must be static. If it weren't, and a.c and b.c shared the same x, its value would be 2, not 1.
(3) y should be extern (it must be shared between a.c and b.c, else its value wouldn't be 2.) However, leaving it without modifier works also (fwiw, int y; followed by int y = 1; compiles and the strong definition is used.) It cannot be static.
(4) z's modifier must be blank so that 'z' becomes a common symbol (shared, but weak). If extern were used, z would be nowhere defined (undefined reference error). If static were used, it would have the value 4, not 3. Note that defining 'z' in a header file is generally considered bad practice.
(5) Should be extern (or left blank, which has identical semantics for functions.) If it were static, 'b' would be undefined in a.c
(6) Must be blank – static would conflict with the earlier declaration of y in the header file. Extern, which declares a variable, is incompatible with an initialization, which can only be used with a definition.
(7) Like (3), can be extern or blank. Note that neither choice is good design. If extern, the declaration should appear in a header. If blank, the definition is redundant (and will be dominated by b.c's strong definition of w.
(8) Must be static since the value printed is 5 – a.c has its own copy of 'v'
(9) Must be blank so that 'w' is defined globally. If static, a.o would have an resolved reference. Like (6), you cannot use extern with a definition.
(10) Could be blank or static. b.c could operate on a global symbol for 'v', or have a local copy – either way, it will not affect the output. Like (9), it cannot be extern.

This question had a significant flaw in that I didn't ask you to explicitly note when you meant to omit a modifier. The default answer (simply leaving everything empty) yielded 10/14 points. Future versions of this question will require you to distinguish between "not answering" and "no modifier".

b) (4 pts) *Dynamic Linking.* One of the features AMD added to the IA32 instruction set when introducing the now-dominant x86_64 architecture was an addressing mode that allows access to data relative to the instruction pointer (e.g., using `displacement(%rip)`). Explain how the compiler makes use of this feature when creating dynamic libraries, and position-independent code in particular!

Position-independent code can be loaded at any location in a process's address space. To find out where a particular function or variable is located, indirection is used in which the address of the function or variable is stored in a location with the global offset table. The shared object is built such that the global offset table is located at a known offset from the instruction pointer, no matter where the shared object is located in a process's address space. Being able to access the instruction pointer directly is convenient because it avoids the need for PC materialization (e.g. doing a call to a compiler-generated subroutine just to learn what address was pushed onto the stack, as is done in 32-bit position-independent code.)

2. X86_64 Programs (25 pts)

The following questions relate to how programs are compiled and optimized for x86_64.

a) (20 pts) *Understanding x86_64 Assembly Code.*

Consider the following function (which may be familiar to fans of projecteuler.net), shown in C as well as compiled with `gcc -O3 -S`:

```
int problem301() {
    int n, s = 0;
    for (n = 1; n <= (1<<30); n++)
        if ((n ^ (n + n) ^ (n + n + n)) == 0)
            s++;
    return s;
}
```

```
    .globl problem301
problem301:
    movl    $3, %esi
    xorl    %eax, %eax
    movl    $1, %edx
.L3:
    leal    (%rdx,%rdx), %ecx
    xorl    %edx, %ecx
    cmpl    %esi, %ecx
    sete    %cl
    addl    $1, %edx
    addl    $3, %esi
    movzbl  %cl, %ecx
    addl    %ecx, %eax
    cmpl    $1073741825, %edx
    jne .L3
    rep
    ret
```

i. (3 pts) Which register holds the value of ‘`n`’?

$edx

ii. (3 pts) Which register holds the value of ‘`s`’?

$eax

iii. (14 pts) Even though the original code contained one for-loop and one if-statement, which would ordinarily result in the use of (at least) two conditional branches, the optimizing compiler was able to compile this code using just one conditional branch (`jne .L3`). In doing so, the compiler exploited arithmetic transformations that optimized the code without changing the outcome of the operations performed. Provide an alternate C version of 'problem301' that has the same characteristics (i.e., a single loop, no if-statement, 4 additions, 2 compares, and 1 xor), illustrating the transformations the compiler performed!

```
int problem301() {
    int n, n3 = 3, s = 0;
    for (n = 1; n <= (1<<30); n++, n3+=3)
        s += ((n + n) ^ n) == n3;

    return s;
}
```

The compiler keeps n + n + n in a separate register ($esi), which is incremented by 3 in each iteration. This can be expressed as a separate C variable n3 = 3 * n. The comparison n ^ (n+n) ^ (n+n+n) == 0 is replaced with n ^ (n + n) == (3 * n) based on the fact that a == b iff a^b == 0. Also note that C guarantees that the value of a == b is 1 if a is equal to b, and 0 otherwise.

b) (5 pts) *Ignoring Compiler Warnings.* Despite pervasive evidence and ongoing encouragement to the contrary, some students still believe that compiler warnings such as `warning: implicit declaration of function` are innocuous and can be ignored because they do not affect the correctness of their programs. Consider the following two separately compiled .c files:

```
// getidcall.c
#include <stdio.h>

int
main()
{
    printf("%ld\n", (long)getid());
    return 0;
}
```

```
// getid.c
long getid()
{
  return 0x100000000;
}
```

During compilation on a Linux x86_64 system such as the one used for the projects, the following warning is displayed:

```
$ gcc -Wall getidcall.c getid.c -o getidcall
getidcall.c: In function 'main':
getidcall.c:7: warning: implicit declaration of function 'getid'
```

Since `-Werror` is not given, gcc built an executable 'getidcall' that can be executed. However, it may or may not work. Fill in the blank in getid.c with any constant value such that 'getidcall' fails when run (i.e., does not print the value returned by getid())!

When a function is not declared, the C compiler substitutes a default definition in which this function returns 'int'. Consequently, the compiler will only consider the lower 32-bits of the result, sign-extending them like so:

```
4004a1:       e8 1a 00 00 00          callq  4004c0 <getid>
4004a6:       48 63 f0                movslq %eax,%rsi
```

As a result, any value for which sign-extension from 32-bit to 64-bit changes the value when interpreted as a 64-bit integer will cause the program to fail, such as 0x100000000. getidcall then prints 0. More precisely, any value that doesn't have all 1s or all 0s in bit positions 63 to 31 (with 63 being msb) will cause the program to fail. Note that -1 is not one of them. Sign-extending 0xFFFFFFFF yields 0xFFFFFFFFFFFFFFFF, which prints as -1.

3. Security (24 pts)

The following questions explore the relationship between program execution and system security.

a) (8 pts) In lecture, we had discussed that modern processors provide hardware support for dual-mode operation in which the processor traps when a program attempts to execute a privileged instruction when run in user mode.

   i. (4 pts) In what way does this feature contribute to a system's security?

It protects the system's security by preventing direct access to low-level hardware resources such as CPU power control, the MMU, or to I/O ports from (potentially compromised) user applications, even applications running with administrative privileges.

   ii. (4 pts) Why does this feature provide only limited protection against most current security attacks?

Direct, low-level access to hardware resources is not required, and usually not used, for most attacks to do significant damage. Compromised processes will ask the kernel to provide it with access to any resource they need for their purpose (e.g. installing backdoors, deleting files, etc.), usually after attempting to gain administrative privileges. There are a (relatively) small number of attacks against kernels, in which case an attacker's code would run in kernel mode and could execute privileged instructions.

b) (4 pts) In lecture, we had discussed that most modern systems exploit address space layout randomization (ASLR) to avoid canonical stack overflow attacks such as the one you constructed in project 2. Has the wide-spread use of ASLR made stack-based buffer overflow vulnerabilities benign? Justify your answer!

No. Even though an attacker may not gain control of the machine, they usually succeed in denying service to users because a vulnerable program will likely crash when attacked. There is also a residual likelihood that the attacker will guess correctly and the attack will succeed.

A common misconception was that nop sleds can render ASLR useless. That's not the case. Nop sleds help with small variations, such as those introduced by different values for environment variables, but the variations introduced by ASLR are so large that a nop sled would need to be very big, typically much larger than the entire stack segment. In this case, a segmentation fault will occur when the vulnerable function attempts to write the nop sled onto the stack because it would reach into the unmapped address space above the stack.

A second misunderstanding was that an attacker would have multiple tries to guess the correct address. In fact, it varies from run to run and a failed guess leads to the termination of the program; if the program is restarted, it'll have its stack at a new address.

It is true that in 32-bit systems, the smaller virtual address space leaves less room for which address to choose for the stack, this fact, in combination with other factors, increases an attacker's likelihood of success. That's not true in today's 64-bit systems, however. For an example of exploiting the limited randomness in 32-bit systems, see Shacham et al. "On the effectiveness of address-space randomization" http://dl.acm.org/citation.cfm?id=1030124

c) (6 pts) Some attacks against vulnerable programs rely on knowing, or predicting with some certainty, the addresses of dynamically allocated objects. For this reason, current system attempt to randomize heap addresses where possible. Based on your knowledge of explicit memory allocators as discussed in class, describe 2 ideas for how an allocator's policies could be varied to achieve such randomization!
(continued on next page)

Many ideas are possible. Allocators could vary

- The start of the heap (picking a random initial brk)
- The placement policy (maybe choose randomly from 'first-fit', 'second-fit', 'third-fit' policies for each allocation)

- Padding of objects – adding a small random amount to each object will introduce variation.
- The splitting policy – toss a coin to split from the top vs. bottom
- The coalescing policy – perform or delay coalescing based on a random value
- The free block insertion policy (head or tail of free list, or at nth position)

In practice, in today's Linux, only the start of the heap is varied, although other ideas are being experimented with. See http://research.microsoft.com/en-us/projects/robustheap/ for information on a Microsoft Research Project that addresses this topic.

Note that acceptable answers needed to include a choice element that was acted on with some randomization.

d) (6 pts) Some researchers have proposed system-call sandboxing, in which a potentially vulnerable program that operates on data from untrusted sources (such as PDF documents downloaded from websites) is run in a "jail" or "sandbox" in which that program's access to system calls is controlled. Sandboxing does not address inherent vulnerabilities such as buffer overflow vulnerabilities, and programs may still suffer from attacks that allow the execution of an attacker's injected machine code. What, then, is the rationale for sandboxing?

The key rationale behind system call sandboxing is that system calls are a *process's only means* of interacting with the outside. Everything else a process has access to (CPU, memory) is virtualized so that whatever a compromised process does to it will not affect the outside (modulo resource consumption such as CPU time or physical memory). This is a key principle of operating systems.

4. Processes in Unix (33 pts)

The following questions relate to how processes execute on Unix.

a) (4 pts) *Process States.* Recall the simplified process state diagram discussed in lecture (consisting of states RUNNING, READY, and BLOCKED).

In which state does a new process start its life after it is created in fork()? Justify your answer!

It's in the READY state. It could run and make use of the CPU as soon as it's being picked by the scheduler. It cannot be placed directly into the RUNNING state because doing so requires action by the scheduler (and there may not be a CPU available for it). It's not BLOCKED because it's not waiting for anything besides a CPU.

In practice, some systems support a dedicated “NEW PROCESS” state to perform long-term scheduling; admitted processes are then moved into the READY state.

Note that I explicitly referred to the nomenclature of process states used in lecture (RUNNING, READY, BLOCKED), with the meanings as discussed in class. Linux has a different terminology; in Linux, a new process is in the RUNNING state, though it will not necessarily have a CPU assigned to it. If you answered “RUNNING” you would need to explicitly refer to Linux’s meaning of RUNNING and discuss when/how a CPU will be assigned to the new process.

b) (15 pts) *timeout*. Unix’s design philosophy favors the use of many small utility programs over large, monolithic services. In this problem, you are asked to implement a ‘timeout’ utility which can be used to terminate any program running longer than a set amount of (wall-clock) time. The timeout command should take the number of seconds after which to terminate the program as its first command-line argument, followed by the command (including any arguments) which the user wants to run. If the command exceeded the given timeout, a message “Timed Out.” should be printed, otherwise, “Ok” should be printed. For example, using the standard ‘sleep’ program, timeout would exhibit the following behavior:

```
$ ./timeout 2 sleep 1
Ok.
$ ./timeout 2 sleep 4
Timed out.
```

<u>Complete timeout.c, shown on the next page!</u>

Additional stipulations:

- If the program exits before the timeout has expired, ‘timeout’ should print “Ok” immediately.

```
// timeout.c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <sys/wait.h>
#include <errno.h>

static int child;
static void alarmhandler(int sig)
{
    kill(SIGTERM, child);
    printf("Timed out.\n");
    exit(EXIT_FAILURE);
}
```

```
int
main(int ac, char *av[])
{
    int seconds = atoi(av[1]);

    if ((child = fork()) == 0) {
        execvp(av[2], av+2);
        perror("execvp:");
        exit(EXIT_FAILURE);
    }

    signal(SIGALRM, alarmhandler);
    alarm(seconds);
    waitpid(child, NULL, 0);
    sigblock(sigmask(SIGALRM));
    printf("Ok.\n");
    exit(EXIT_SUCCESS);
}
```

Simply arming the timer and executing the desired program (without forking a new process) will not work since signal handlers are reset to their defaults on exec() (since the child will not have the handler function).

To avoid being interrupted in printf("Ok\n") if the alarm goes off right after the child exited, SIGALRM must be blocked for the subsequent printf() call, which is not async-signal-safe.

The example shows the old-style signal API (signal, sigmask) but keep in mind that new applications should use the corresponding POSIX API (sigaction, sigprocmask).

Other approaches are possible, such as setting both a SIGCHLD and SIGALRM handler, then pause()ing the main program.

c) (9 pts) *Pipes.* Shells use the Unix system call `pipe(2)` to connect a program's standard output to another program's standard input. Some of you have implemented this in your shells. Here is an example program that shows how such redirection works. It implements a program '`spipe.c`', which starts two child processes whose standard input/output is connected through a pipe. In other words, `./spipe program1 program2` is equivalent to running `program1 | program2` in a shell that supports pipes:

```
$ ./spipe uname cat
Linux
$ uname | cat
Linux
$ ./spipe uname wc
      1       1       6
```

```
$ uname | wc
      1       1       6
```

For simplicity, spipe does not support passing on any arguments to the commands it launches. For presentation purposes, error handling was omitted as well.

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

/* Launch a child process, redirect its standard out to stdout,
 * standard in to stdin.  Also close 'fdtoclose' in child.
 * Close any redirected file descriptors in parent before
 * returning.  Returns pid of child process.
 * (Implementation not shown.)
 */
int launch_child(char *prgname, char *prgargv[],
                 int stdout, int stdin, int fdtoclose);

int
main(int ac, char *av[])
{
    char *firstprogargv[] = { av[1], NULL };
    char *secondprogargv[] = { av[2], NULL };
    int pipefd[2];

    pipe(pipefd);                           // create pipe

    // launch first child
    pid_t leftchild = launch_child(av[1], firstprogargv,
                                   pipefd[1], 0, pipefd[0]);
    waitpid(leftchild, NULL, 0);            // reap first child

    // launch second child
    pid_t rightchild = launch_child(av[2], secondprogargv,
                                    1, pipefd[0], pipefd[1]);
    waitpid(rightchild, NULL, 0);           // reap second child
    return EXIT_SUCCESS;
}
```

Users observed the following behavior of `spipe`. For many combinations of programs, `spipe` appears to work as intended. For some (such as '`./prog2`'), however, it "gets stuck." When it gets stuck, ps shows:

```
$ ps f
  PID TTY      STAT   TIME COMMAND
24889 pts/9    Ss     0:00 -bash
31462 pts/9    S      0:00  \_ ./spipe ./prog2 wc
31463 pts/9    S      0:00  |   \_ ./prog2
31490 pts/9    R+     0:00  \_ ps f
```

Based on this information, answer the following questions:

i. (3 pts) Does this apparent bug of `spipe` cause excessive CPU usage on the machine on which the user runs it? Justify your answer!

No, all processes involved are in the BLOCKED state (Linux calls this ‘S’ for Sleeping as can be seen in the output of ps), they do not use any CPU.

A frequent wrong answer was that the processes are STOPPED. Linux uses ‘T’ for STOPPED processes (stopped for any reason, including SIGTSTP, SIGTTOU, SIGTTIN, SIGSTOP). “Stopped” is different from “Sleeping.” A “stopped” process cannot proceed until SIGCONT is sent, even if it is not blocked on any event and even if a CPU is available.

Another frequent wrong answer was that “stuck processes” eat up CPU time, or become zombies.

ii. (3 pts) Why does ‘`spipe`’ not finish? Be precise!

Pipes have finite capacity; when a process attempts to write to a full pipe, it is blocked. Here, prog2 filled the pipe, then blocked when attempting further writes. The process that consumes data from the pipe (wc in this case) hasn’t been started because the parent spipe program is waiting for prog2 to finish first. A deadlock results. If you’ve done thorough testing on your pipe implementation, your testing should have covered this case, which I also discussed in class.

FYI, here’s my prog2.c, which writes exactly 64KB + 1 bytes.

```
#include <stdio.h>

int
main()
{
    int i;
    for (i = 0; i < 65537; i++)
        fputc('A', stdout);
    return 0;
}
```

Though the information provided didn’t allow you with certainty to conclude that prog2 is blocked writing to its standard out stream, it allowed you to rule out a number of other scenarios. First, there are no bugs with respect to file descriptors, or else spipe wouldn’t have worked at all. Second, the command line shown in the ps output made it clear that prog2 is the first program in the pipe, not the second, ruling out any answer that speculated that the second child had already exited. Another frequent wrong answer was that some suspected a race condition with the waitpid() call, “reaping” the child too early. Calling waitpid() to reap a child doesn’t terminate it, despite what the imagery accompanying the term suggests. On the contrary, the use of waitpid() avoids any such race conditions.

iii. (3 pts) Suggest a way to fix spipe!

Don’t wait for the first child until after both children have been forked; i.e. move the ‘waitpid(leftchild, NULL, 0);’ call after the second call to launch_child(). This will allow the children to run concurrently so that wc can drain the pipe whenever prog2 writes into it.

d) (5 pts) *Understanding Signals.* Some groups in project 3 attempted to implement the shell built-in command kill in the following fashion:

```
// jobid - id of job to be killed
// killsignal - number of signal to be used,
//              e.g. SIGTERM or SIGKILL
void builtin_kill(int jobid, int killsignal)
{
      struct esh_pipeline * job = find_job_from_id(jobid);
      killpg(killsignal, job->pgrp); // send kill signal
      remove_job_from_job_list(job);
}
```

Why is this implementation approach incorrect?

Programs cannot assume that signals are delivered and acted upon immediately. After the signal is sent, it may take some time for the process to receive it and act on it. In addition, for some signals, such as SIGTERM, a process could decide to block and/or ignore it. In this case, killpg() would not lead to the termination of those processes and the shell should not assume it does. Instead, the shell must wait for SIGCHLD, then call waitpid() to learn about the actual fate of the process(es) the user attempted to kill.
Note that there are even situations where it takes a relatively long time for processes to be terminated via SIGKILL, such as when processes are executing in certain sections of kernel code.

The code contains a second, unintended mistake some of you spotted: I got the argument order in killpg() wrong: it should be killpg(job->pgrp, killsignal).

We accepted for full credit both answers. A number of students thought that killpg() takes a negative argument (it doesn’t – the negative argument trick is used in kill()), or that killpg() only sends a signal to a process group’s leader rather than the entire process group. Also, we didn’t accept answers such as “you have to block SIGCHLD while manipulating the jobs list,” which, though true, would presumably be done in find_job_in_jobs_list and remove_job_from_job_list, but the latter function shouldn’t be called from builtin_kill anyway.