CS 3214 Midterm Solutions

Solutions are shown in this style. This exam was given Spring 2016.

1. Abstraction and Protection (12 pts)

a) (6 pts) An operating system is a software layer that provides abstractions for specific hardware components as well as interfaces that allow user programs to access these abstractions.

Give one example of such an abstraction, the hardware component(s) it represents, and the interface provided to it!

The abstraction called ______________________________

represents the concrete hardware component(s) ________________

____

and it is accessed via the interface called ____________________

____.

There are many possible answers, examples are shown in the table below:

Abstraction	*Hardware*	*Interface*
Process/Thread	*CPU*	*Instruction Set, Process/Thread Management API*
Virtual Memory	*Memory*	*Virtual addresses, Memory Management API*
Files	*Storage Devices (disk, etc.)*	*File descriptor API*
Standard I/O	*I/O Terminals*	*File descriptor API*
Sockets	*Network Interface Card*	*File descriptor + socket API*
(Alarm) Signal	*Timer Chip*	*signal(2), SIGALRM*

b) (6 pts) We know that in systems exploiting dual-mode operation, the operating system will receive a trap if a user program attempts to execute a privileged instruction, as for example a C program that contains a `asm("hlt")` inline asm statement.

To avoid the overheads associated with dual mode operation, such as the cost of mode transitions, evaluate the following two proposed alternative designs and state whether they would provide the same level of protection! Justify your answer!

i. Outlaw "asm" statements so that the programmer cannot insert assembly code that invokes privileged instructions into their programs.

This would not yield the same level of protection since programmers could side-step the compiler by directly writing assembly code; code generated by just-in-time compilers that directly emit native code would also not be covered.

ii. Perform a check during the linking and loading process in which the executable is scanned for privileged instructions.

This would provide the same level of protection only if all user code must be loaded by the OS; again, a possible loop hole here would be just-in-time compilation. Closing this loop hole would require outlawing any kind of self-modifying code or just-in-time compilation, for instance, don't allow segments that are both writable and executable, or changing segments from writable to executable.
A second problem arises with variable-length instruction sets, such as x86, where the analysis would need to verify all possible branch targets as well.
Researchers have in fact proposed and built systems that execute in a single mode and restrict user programs to be written in verifiable, type-safe languages.

2. Processes (18 pts)

a) (14 pts) Assume the following timeline that maps three processes P1, P2, and P3 to their states (READY, RUNNING, BLOCKED). At certain points, processes change the state they are in. In the table below, list one possible reason for why a process may have switched state! Make sure you do not just say: "P1 transitioned from BLOCKED to READY" or "context switch" – but rather give a reason for why a particular state transition may have happened! As you consider possible reasons, note the constraints given by (#), (##), and (*)!

#	P1	P2	P3	Possible reason
	BLOCKED	BLOCKED	BLOCKED	*Initially, the system is idle*
1	READY	BLOCKED	BLOCKED	*A timer interrupt wakes up P1 (say from nanosleep())*
2	RUNNING	BLOCKED	BLOCKED	*The scheduler picks P1 to run*
3	RUNNING	READY	BLOCKED	*(#) P2 wakes up from network I/O when a packet arrives*
4	RUNNING	RUNNING	BLOCKED	*The scheduler picks P2 to run (on a second CPU)*
5	RUNNING	RUNNING	READY	*(##) P3 wakes up because of a communication-related event, say a semaphore signal.*

6	RUNNING	READY	READY	*P2 is preempted because its time slice has expired (or it calls sched_yield())*
7	RUNNING	READY	RUNNING	*The scheduler picks P3 to run*
8	BLOCKED	READY	RUNNING	*P1 starts a read() from a file on disk*
9	BLOCKED	RUNNING	RUNNING	*The scheduler picks P2 to run (on the CPU on which P1 ran)*
10	BLOCKED	RUNNING	BLOCKED	*(*) P3 blocks say on a lock or semaphore*
11	READY	RUNNING	BLOCKED	*P1's read() from disk, started in #8, completes.*

(#) provide a different reason here than in row #1.
(##) provide a different reason here than in rows #1 and #3.
(*) provide a different reason here than in row #8.

Note that the event in #11 must match the reason for why P1 blocked in #8. #1, #3, and #5 of course could have different answers as long as three examples are given for why a process may transition into the READY state. Ditto for #8 and #10.

b) (4 pts) Assuming a work-conserving scheduler (which does not unnecessarily let CPUs idle), <u>how many CPUs does the system on which this trace occurred have?</u>

It has 2 CPUs since there were 2 processes in the RUNNING state in #4 and #5.

3. Linking (18 pts)

a) (10 pts) Consider the following code:

```
// link.h
#include <stdio.h>

static int s;
int c;
extern int i;
int * g(void);

static void incs(void) {
    s++;
}
```

```
// link2.c
#include "link.h"

void f() {
    s = c + 3;
    c = c + 2;
    i = 2 * s;
}

int
main()
```

`// link1.c`
`#include "link.h"`
 	
`int i = 2;`
 	
`int * g(void) {`
`    i = (c * 7) * s;`
`    incs();`
`    return &s;`
`}`
	`{`
	`    f();`
	`    if (g() == &s) {`
	`        incs();`
	`    } else {`
	`        c = c + 2;`
	`        i = 2 * s;`
	`    }`
	`    printf("s = %d c = %d i = %d\n",`
	`            s, c, i);`
	`    return 0;`
	`}`

i. (6 pts) The program is built with `gcc -Wall link1.c link2.c -o link`. What does it output when run with `./link`?

```
s = 3 c = 4 i = 6
```

ii. (4 pts) Which recommended "best practice" does this program violate?

It defines variables in header files (i.e., s and c)

b) (4 pts) The standard linker builds executables out of separately compiled .o object modules. Modern systems also support dynamically loaded libraries (i.e., that are loaded at runtime). Yet, this setup lacks many of the features provided by the existing module systems in higher-level languages. Consider one such high-level language with which you are familiar and name one feature of its module system that is not provided by the standard linker!

Examples include

- *Hierarchical namespaces (e.g. java.util.*) rather than a global namespace*
- *Multiple visibility levels (e.g. public, protected, private, package-level visibility) rather than just 2 (local + global)*
- *Non-leaky dependencies (specific "import" or "import from" statements to specify from where references should be imported or resolved)*

And possibly others.

Non-examples include:

- *Encapsulation (we have some degree of that using local symbols)*
- *Ability to load modules at run-time (we can do some of that using dlopen()/dlsym())*

c) (4 pts) Wikipedia[1] defines Position-Independent Code as:

[1] https://en.wikipedia.org/wiki/Position-independent_code

In computing, **position-independent code** (**PIC**) or **position-independent executable** (**PIE**) is a body of machine code that, being placed somewhere in the primary memory, executes properly regardless of its absolute address.

Building shared libraries as PIC allows them to be loaded at different addresses in different processes.
Name one reason for why building shared libraries as PIC is needed or useful!

It is needed because the same address may not be available in different processes, for instance, if different dynamic libraries are already loaded, or addresses are used for heap space, stacks, or mmap'd regions. Even if the same address were available, it is useful to load libraries at different addresses to provide ASLR (address space layout randomization). Without position-independent code, libraries would have to be relocated at load-time, which is expensive and would prevent those libraries from being shared, thus increasing memory use.

4. System Calls (18 pts)

a) (10 pts) The `system()` function is defined in the C99 standard. Below are relevant excerpts from its man page:

```
SYSTEM(3)                 Linux Programmer's Manual           SYSTEM(3)

NAME
       system - execute a shell command

SYNOPSIS
       #include <stdlib.h>

       int system(const char *command);

DESCRIPTION
       system()  executes a command specified in command by calling
       /bin/sh -c command, and returns after the command has
       been completed.

RETURN VALUE
       The value returned is -1 on error, and the return status of the
       command otherwise.  This latter return status is in the
       format specified in wait(2).

       system() does not affect the wait status of any other children.
```

Interestingly, system() is not a system call in Unix.
Implement it according to the (partial) specification presented here using system calls you have used in project 1!

```
int system(const char *command)
{
    pid_t child = fork();
    if (child == 0) {
        execlp("/bin/sh", "/bin/sh", "-c", command, NULL);
    } else {
        int status;
        waitpid(child, &status, 0);
        return WEXITSTATUS(status);
    }
}
```

Instead of execlp, you could also use other variants of the exec() family, such as execv:*

```
int system(const char *command)
{
    pid_t child = fork();
    if (child == 0) {
        const char *av[] = { "/bin/sh", "-c", command, NULL };
        execv(av[0], (char * const *) av);
    } else {
        int status;
        waitpid(child, &status, 0);
        return WEXITSTATUS(status);
    }
}
```

We assume that /bin/sh exists so that the exec() will not fail. To meet the requirement that the wait status of other children is not affected, it is necessary to use waitpid() (or wait4()).*

b) (4 pts) During exercise 3 (“Outfoxed”) a student attempted to debug a deadlock in his code by attaching gdb to the process running his code. Inadvertently, he attached to the driver script (checkoutfoxed.py) instead and posted this backtrace on the Piazza forum:

```
(gdb) backtrace
#0 0x00007f633c2cfa40 in __read_nocancel () from /lib64/libc.so.6
#1 0x00007f633c25e739 in __GI__IO_file_xsgetn () from /lib64/libc.so.6
#2 0x00007f633c253e5f in fread () from /lib64/libc.so.6
#3 0x00007f633cf36dbc in file_read () from /lib64/libpython2.7.so.1.0
#4 0x00007f633cfaf5d2 in PyEval_EvalFrameEx () from /lib64/libpython2.7.so.1.0
#5 0x00007f633cfb00bd in PyEval_EvalCodeEx () from /lib64/libpython2.7.so.1.0
#6 0x00007f633cfae76f in PyEval_EvalFrameEx () from /lib64/libpython2.7.so.1.0
#7 0x00007f633cfb00bd in PyEval_EvalCodeEx () from /lib64/libpython2.7.so.1.0
#8 0x00007f633cfae76f in PyEval_EvalFrameEx () from /lib64/libpython2.7.so.1.0
#9 0x00007f633cfae860 in PyEval_EvalFrameEx () from /lib64/libpython2.7.so.1.0
#10 0x00007f633cfb00bd in PyEval_EvalCodeEx () from /lib64/libpython2.7.so.1.0
#11 0x00007f633cfb01c2 in PyEval_EvalCode () from /lib64/libpython2.7.so.1.0
#12 0x00007f633cfc95ff in run_mod () from /lib64/libpython2.7.so.1.0
#13 0x00007f633cfca7be in PyRun_FileExFlags () from /lib64/libpython2.7.so.1.0
#14 0x00007f633cfcba49 in PyRun_SimpleFileExFlags () from /lib64/libpython2.7.so.1.0
#15 0x00007f633cfdcb9f in Py_Main () from /lib64/libpython2.7.so.1.0
#16 0x00007f633c209b15 in __libc_start_main () from /lib64/libc.so.6 #17 0x0000000000400721 in _start ()
```

Why was the driver script blocked in fread() when the student's program deadlocked and which file was it trying to read from?

To grade the exercise, the driver program reads the outfoxed program's output via a pipe it sets up for this purpose; when the program deadlocked, the driver was waiting to read its output via the file descriptor representing the pipe's read end.
Being able to write driver programs like that is one example that demonstrates the universality of the system calls (fork, exec, pipe, read, write, etc.) you gained familiarity with in your shell project. Many languages provide an API for these system calls, in Python, for instance, it's subprocess.Popen which the driver uses.

c) (4 pts) At the ACM Intercollegiate Programming Contest (ICPC), speed matters – both in writing programs quickly and in writing fast programs that meet the allotted time limit. Sometimes, programs must output large amounts of data which they have computed.
A common optimization competitors perform is to replace code such as:

```
for (int i = 0; i < 100000; i++)
    System.out.printf("%d %d\n", a[i], b[i]);
```

With this code:

```
StringBuilder out = new StringBuilder();
for (int i = 0; i < 100000; i++)
    out.append(String.format("%d %d\n", a[i], b[i]));
System.out.print(out.toString());
```

Why does the second version of this code run much faster (Note that `System.out.printf()` internally calls `String.format()`)?

Since both versions do essentially the same output formatting, the performance difference must stem from the frequency of write(2) system calls. The second version does one write(2) system call, the first version does many.

Interesting factoid: Java's System.out flushes its buffer every time a newline \n is output, performing a write(2) system call, thus 100,000 calls are made in this example. Unlike for C's stdio, this holds true even if the standard output file descriptor is not connected to a terminal, i.e., if it is redirected to a file. Why Java's designers made this decision, we don't know. Perhaps to avoid confusion such as this one. However, the same is not true for a PrintWriter() instance.

5. Locks (12 pts)

a) (12 pts) We know that in multithreaded processes, file descriptors are shared between all threads in a process. If multiple threads simultaneously call `read()` on a file descriptor, only one thread will be successful in reading any available data up to the buffer size provided, the others will remain blocked until more data is available to be read. Based on this knowledge, implement an intraprocess mutual exclusion device (i.e., a lock) using Unix pipes! (Reminder: the read end of a pipe is at index [0], the write end at index [1].)

```
/* Define a pipe-based lock */
struct pipelock {
    int fd[2];
};

/* Initialize lock */
void lock_init(struct pipelock *lock)
{
    pipe(lock->fd);
    write(lock->fd[1], "a", 1);
}

/* Acquire lock */
void lock_acquire(struct pipelock *lock)
{
    char c;
    read(lock->fd[0], &c, 1);
}

/* Release lock */
void lock_release(struct pipelock * lock)
{
    write(lock->fd[1], "a", 1);
}
```

The crucial hint was given in the question: if multiple threads attempt to read, only one thread will read data, the others will block. That's exactly the semantics you need for a lock: if multiple threads try to acquire it, only one thread will succeed, and the others will remain blocked. Pipe locks are widely used for their portability and fairness.

6. Condition Variables and Semaphores (22 pts)

a) (10 pts) In the lecture slides, we discussed the motivation for higher-level signaling facilities using a simple example of a coin toss: one thread was "tossing a coin" and the other thread must be informed of the outcome of this coin toss quickly, reliably, and efficiently.
A long time ago, Dr. Back tried to write a version of this program using condition variables. The program is still in the class website's examples directory. It is reproduced below:

```
/*
 * Synchronization via condition variables.
 *
 * This program contains a bug.  Find it and fix it.
 */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>

int coin_flip;          // 0 or 1, heads or tails
bool coin_flip_done;    // states whether 'coin_flip' contains valid result
// lock below protects coin_flip_done;
pthread_mutex_t coin_flip_done_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t coin_flip_done_cond = PTHREAD_COND_INITIALIZER;

static void *
producer(void *_items_to_produce)
{
    int i, n = *(int *)_items_to_produce;
    for (i = 0; i < n; i++) {
        pthread_mutex_lock(&coin_flip_done_lock);
        while (coin_flip_done)
            pthread_cond_wait(&coin_flip_done_cond, &coin_flip_done_lock);

        coin_flip = rand() % 2;
        coin_flip_done = true;
        printf("thread %p: flipping coin %d\n",
            (void *)pthread_self(), coin_flip);
        pthread_cond_signal(&coin_flip_done_cond);
        pthread_mutex_unlock(&coin_flip_done_lock);
    }
    return NULL;
}

static void *
consumer(void *_items_to_consume)
{
    int i, n = *(int *)_items_to_consume;
    for (i = 0; i < n; i++) {
        pthread_mutex_lock(&coin_flip_done_lock);
        while (!coin_flip_done)
            pthread_cond_wait(&coin_flip_done_cond, &coin_flip_done_lock);

        coin_flip_done = false;
        printf("thread %p: coin flip outcome was %d\n",
            (void *)pthread_self(), coin_flip);
        pthread_cond_signal(&coin_flip_done_cond);
        pthread_mutex_unlock(&coin_flip_done_lock);
    }
    return NULL;
}

int
main()
{
    int i;
    #define N  4
    pthread_t t[N];
    int items [N] =             { 3,        2,        4,        1 };
    void * (*func [N])(void*) = { consumer, consumer, producer, producer };
    srand(getpid());
```

```
        for (i = 0; i < N; i++)
            pthread_create(&t[i], NULL, func[i], items + i);

        for (i = 0; i < N; i++)
            pthread_join(t[i], NULL);
        return 0;
    }
```

Dr. Back now wishes he had commented his program because he doesn't remember what bug this program contained! His intent was for 2 consumer threads to produce (3+2=5) coin tosses and for 2 producer threads to consume the same number (4+1=5). Producer threads should toss coins only when `coin_flip_done` is false and consumer threads consume them only when `coin_flip_done` is true.
Analyze the example program and find the bug!
Specifically, describe

i. (2 pts) How the program might fail (what behavior you would observe if it did)

The program will deadlock/hang/get stuck/not finish.

ii. (4 pts) The root cause of the failure

A single condition variable is used. As such, when a consumer signals that it is time to flip another coin, the call to pthread_cond_signal() may wake up the other consumer thread instead of a producer. As a result, the woken up consumer will find that no coin has been tossed and will go back to pthread_cond_wait(), never waking up any producer. Remember that pthread_cond_signal() will only wake up one thread. The same scenario can also happen when a producer's signal will wake up the other producer instead of a consumer.

iii. (4 pts) A possible approach to fixing it.

You should fix it by using 2 separate condition variables – one that producers signal and consumer wait on, and the other for consumer to signal and producers to wait on. See rw lock example in lecture slides for a similar approach.

b) (4 pts) Edsger Dijkstra, the inventor of semaphores, reflected on their use in the 'THE' – Multiprogramming system in a 1968 paper as follows:

During system conception it transpired that we used the semaphores in two completely different ways. The difference is so marked that, looking back, one wonders whether it was really fair to present the two ways as uses of the very same primitives.

Which two separate uses is Dijkstra talking about?

He is talking about the uses of semaphores for mutual exclusion (initialized with 1) and signaling/scheduling (initialized with 0).

In fact, the quote continues: "On the one hand, we have the semaphore used for mutual exclusion, on the other hand, the private semaphores."
(They were called "private" because only a single process will ever call P() on them, i.e., wait on them.) See [Dijkstra 1971, pg. 346]

c) The concept of a monitor, from which condition variables arose, was invented later in the 1970s. The 2012 book *Operating Systems: Principles and Practice* by Anderson and Dahlin quotes Dijkstra's reflection on semaphores and relates them to condition variables in the following paragraphs:

Semaphores considered harmful. *Our view is that programming with locks and condition variables is superior to programming with semaphores, and we advise you to always write your code using those synchronization variables for two reasons.*

First, using separate lock and condition variable classes makes code more self-documenting and easier to read. As the quote from Dijkstra above notes, there really are two abstractions here, and code is clearer when the role of each synchronization variable is made clear through explicit typing.

Second, a stateless condition variable bound to a lock turns out to be a better abstraction for generalized waiting than a semaphore. By binding a condition variable to a lock, we can conveniently wait on any arbitrary predicate on an object's state. In contrast, semaphores rely on carefully mapping the object's state to the semaphore's value so that a decision to wait or proceed in P() can be made entirely based on the value, without holding a lock or examining the rest of the shared object's state.

Answer the following 2 questions:

i. (4 pts) Illustrate the phrase "carefully mapping the object's state to the semaphore's value" with an example of an object whose state can be mapped to a semaphore's value!

If the semaphore represents some countable resource, say the number of slots available in a bounded buffer, it can be mapped directly. Calling post() increments the number of available slots by one, calling wait() decrements it. Another example is using 0/1 for "not signaled"/"signaled".

ii. (4 pts) Provide a reasonable counter argument to Anderson's view (i.e., a justification for preferring semaphores in certain scenarios, despite the undisputed fact that all such uses could, indeed, be replaced with condition variables)!

In my view, certain uses of semaphores for simple signaling tasks are perfectly safe and idiomatic. An example would be the simple rendezvous pattern (where thread A signals to thread B that it has reached some point and that B can access results A has computed and will not access again), and thus in which no state needs to be protected using locks or other semaphores. The resulting code is probably more concise, readable, and as robust as if it had been written using condition variables and locks.
Another use is the use of counting semaphore to regulate concurrency (the number of threads entering a certain section of code).

It would be much harder to defend the use of semaphores when it comes to using them for mutual exclusion (where locks are preferred), as well as for tasks that involve multiple accesses to shared state (where you would need to use one semaphore for mutual exclusion and another for signaling). Dahlin and Anderson, I suspect, were probably having these more complex uses in mind.

Weaker arguments included that condition variables are harder to use due to having to deal with spurious wakeups and having to combine them correctly with locks. We should assume that the programmer choosing from these alternatives masters each correctly.

The question asked you to take a position in pursuit of a particular line of argument, so I tried to grade your answer based on how reasonable I thought your position to be.

A number of you pointed out that semaphore remember signals whereas condition variables do not. This is true, and this is why condition variables need to be tied to predicates that are evaluated under the protection of a lock. I don't see this as an argument against Anderson's view.

Lastly, a number of you claimed performance or lower overhead benefit for semaphores. This is not true, as practically the same operations are performed in both scenarios – in the common case (uncontended) an atomic instruction to either change the semaphore's value or acquire the lock, followed by a test. The dominant cost is that of the atomic instruction. The un-signaled case is also similar in both cases: a Boolean check, following by some way to block the calling thread.