Sample Final Exam (Fall 2010)

Solutions are shown in this style. This exam was given Fall 2010.

1. Multithreading (18 pts)

a) (15 pts) *Mutexes and Condition Variables*. Exercise 10 asked that you implement futures, a commonly used abstraction for task parallelism, backed by a pool of threads. Each of these threads will execute a work function that processes tasks submitted to the pool. The following 5 examples were submitted by students. Point out what is wrong with them! To keep the focus of this question on the use of mutexes and condition variables to implement the monitor abstraction (which was the point of this exercise), irrelevant details have been omitted and you should assume they are correctly implemented.

```
// here, 'pop' returns the first element
// of the pool's queue, or NULL if queue
// is empty
while ( !getShuttingDown(pool) ) {
    struct future *ftr = NULL;

    pthread_mutex_lock(&pool->mutex);
    ftr = pop(pool);
    pthread_mutex_unlock(&pool->mutex);

    while ( ftr == NULL ) {
        pthread_mutex_lock(&pool->mutex);
        ftr = pop(pool);
        pthread_mutex_unlock(&pool->mutex);

        if ( getShuttingDown(pool) ) {
            return NULL;
        }
    }

    ... process task and signal semaphore ...
}
```

i) (3pts)

This attempt employs busy-waiting and does not use condition variables at all.

```
while(1)
{
    pthread_mutex_lock(&(pool->startWorkMutex));
    if (is_empty(pool->jobs))
    {
        pthread_cond_wait(&(pool->startWorkCondition),
                          &(pool->startWorkMutex));
    }
```

```
        if (!pool->shuttingDown) {
            struct future * next_Job = poll_job(pool->jobs);
            pthread_mutex_unlock(&(pool->startWorkMutex));
            ... process task and signal semaphore ...
        }
        else
        {
            pthread_mutex_unlock(&(pool->startWorkMutex));
            break;
        }
    }
```

ii) (3 pts)

This attempt uses 'if' instead of 'while'. For condition variables in monitors that employ Mesa semantics (including pthreads, Java's and C#'s monitors) the call to pthread_cond_wait() must always be preceded by a while. There is no guarantee that the predicate a thread is waiting for is true upon return from pthread_cond_wait(). Notably, another thread (not the one being signaled) could have picked the task from the pool's queue. Note that this can happen even though pthread_cond_signal wakes up only one thread, for instance if a thread who has just completed a task acquires the lock earlier than the signaled thread. (Moreover, the POSIX standard allows pthread_cond_wait() to return spuriously without any pthread_cond_signal call having happened at all!)

```
/* Wait for new futures to be enqueued in the thread pool's work queue.
*/
    pthread_mutex_lock(&pool->mutex);
    pthread_cond_wait(&pool->cond, &pool->mutex);
    while (pool->futures != NULL) {
        if (!pool->shutting_down) {
            fut = pool->futures;
            pool->futures = pool->futures->next;  // Dequeue future
            pthread_mutex_unlock(&pool->mutex);
            ... process task and signal semaphore ...
            pthread_mutex_lock(&pool->mutex);
        }
    }
    pthread_mutex_unlock(&pool->mutex);
```

iii) (3 pts)

This attempt uses a condition variable to wait for the first task only, rather than for every task. Subsequently, the thread will exit if upon completion of a task it finds the pool's task queue empty, which means this pool runs out of threads if there is any period during which no tasks are pending.

```
    while(true) {
        // Wait for signal then check if shutting down
```

```
        pthread_mutex_lock(&tp->mutex);
        pthread_cond_wait(&tp->cond, &tp->mutex);
        shutdown = tp->shutdown;

        if(shutdown)
            break;

        f = remove_future_from task();

        ... process task f ...
        pthread_mutex_unlock(&tp->mutex);
        ... signal f's semaphore ...
    }
    pthread_mutex_unlock(&tp->mutex);
```

iv) (3 pts)

This attempt does not check any predicate before calling pthread_cond_wait. If the signal occurred before the thread reached pthread_cond_wait(), it will be lost. Condition variables, unlike semaphores, don't store sent signals.

```
    struct future *current;
    for(;;)
    {
        pthread_mutex_lock(&(pool->lock));

        while(pool->list_size == 0) //if the list is empty, wait
        {
            if(pool->shutdown)
            {
                pthread_mutex_unlock(&(pool->lock));
                pthread_exit(NULL);
            }
            //else wait until not empty
            pthread_mutex_unlock(&(pool->lock));
            pthread_mutex_lock(&(pool->condLock));
            pthread_cond_wait(&(pool->job_available),
                              &(pool->condLock));
            pthread_mutex_unlock(&(pool->condLock));
            pthread_mutex_lock(&(pool->lock));
        }
        current = ... get task from queue ...
        pthread_mutex_unlock(&(pool->lock));
        ... process task and signal semaphore ...
    }
```

v) (3 pts)

This attempt (wrongly) uses a second lock. This means that giving up the pool's lock and being added to the job_available condition variable's queue is no longer atomic, thus it can lead to lost wakeups if a signal arrives between the two calls.

b) (3 pts) *Semaphores*. Access to shared variables in multithreaded programs must be synchronized. Consider the following scenario with 2 threads!

Global definitions and code executed before either thread is spawned	Executed by thread 1	Executed by thread 2

`void *shared_var;`
`sem_t sem;`
 	
`sem_init(&sem, 0, 0);`
	`shared_var=compute();`
	`sem_post(&sem);`
		`sem_wait(&sem);`
		`return shared_var;`

Is this program, as presented, without race condition or do the accesses to 'shared_var' by the two threads need to be protected with a mutex? Justify your answer!

No mutex is needed since the semaphore guarantees that the assignment to shared_var happens before the use in the return statement. (In fact, this is the pattern exploited by the future_get() implementation in your thread pool!). The use of sem_post() and sem_wait() also ensures that the store by thread 1 will be seen by the subsequent load by thread 2, even if they run on different cores.

A wrong answer would be to say that a mutex is not needed because one access is an assignment/store and the other access is a use/load.

2. Memory Management (18 pts)

The following questions relate to memory management in systems exploiting virtual memory.

a) (6 pts) Consider the following C program malloc.c:

```
#include <stdio.h>
#include <stdlib.h>

int
main(int ac, char *av[])
{
    return printf("%p\n", malloc(124));
}
```

When run multiple times on Linux, the output may be:

```
$ ./malloc ; ./malloc ; ./malloc ; ./malloc ; ./malloc
0x901d008
0x953a008
0x89d4008
0x870b008
0x939a008
```

i. (3 pts) Explain why this program outputs different values every time, even though it does not read any user input (and does not make use of any random number functions)!

The different values are explained by address space randomization, a defensive technique that aims to make overflow attacks more difficult by randomly changing the virtual addresses at which the heap is located for each run of a program.

ii. (3 pts) Provide a plausible reason for why all numbers end in 008!

Since the heap is a separate segment for the purposes of virtual memory management (as shown by /proc/*/maps), it is comprised of entire pages. Like your project 4 allocator, the GNU libc allocator used in Linux must guarantee an 8-byte alignment for each allocated object. Given that the implementation of free() requires object headers located at a constant offset before the payload, the first location in a 4KB page that is 8-byte aligned and allows a header to be placed before the payload is 4KB * n + 0x008. (Note that the page size on IA32 is 0x1000 = 4096 (dec)).

b) (6 pts) Besides throughput, an allocator's performance can be described by its utilization metric. In project 4, the utilization score for your allocator was obtained by measuring the ratio of aggregate payload to overall heap size at the aggregate payload's peak. In this question, you should consider the alternative approach of computing the average size of all free blocks as a performance measure.

i. (3 pts) How is the average free block size related to an allocator's utilization?

Generally, if an allocator suffers from external fragmentation, it is likely that the free list would contain many small free blocks and few large ones.

ii. (3 pts) Why is the average-free-block-size metric probably not as relevant as the utilization ratio used in project 4?

The existence of many free blocks does not cause problems per se. They cause problems only indirectly when future requests are made that can't be satisfied from the free list, thus causing the allocator to expand its heap; this effect is captured by the peak utilization metric.

c) (6 pts) The provided memory allocator in project 4 was a naïve allocator that grew its heap every time a request from a client arrived, and that ignored calls to free() altogether. After learning about virtual memory and page replacement, your teammate proposes the following approach:

Let's exploit the OS's ability to manage virtual memory! 64-bit OS are

quickly becoming prevalent – in these systems, virtual addresses are a practically unlimited resource, relative to the memory allocated by even long-running applications before they are shut down. Let's optimize the approach so that memory is obtained from the OS in larger chunks, and a pointer is kept to the remaining area in the current chunk. Let's continue to not do anything on a free() call – the OS's page replacement algorithm will eventually page out those pages, ensuring that they do not consume actual resources in terms of physical memory.

Will this approach work? If not, where did your team mate go wrong? Justify your answer!

Although it is true that the OS will likely page out freed pages if they have not been accessed in a long time, the approach will fail because the OS does not know that the pages contain data that from the perspective of the user program is unused. Thus, it will be forced to preserve their content by writing them to swap space. Unlike virtual addresses, swap space is limited (most administrators configure it at a small multiple of physical memory).

3. Standard I/O and Unix I/O (10 pts)

Consider the following two versions of a simple file copy program (which provides similar functionality to /bin/cp):

```
// Version 1
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>

int
main(int ac, char *av[])
{
    int ifd = open(av[1], O_RDONLY);
    if (ifd == -1) perror("open"), exit(EXIT_FAILURE);
    int ofd = creat(av[2], 0666);
    if (ofd == -1) perror("creat"), exit(EXIT_FAILURE);

    int bread;
    char buf[65536];
    while ((bread = read(ifd, buf, sizeof buf)) > 0)
        if (write(ofd, buf, bread) != bread)
            perror("write"), exit(EXIT_FAILURE);

    return EXIT_SUCCESS;
}
```

and

```
// Version 2
#include <stdio.h>
```

```
#include <stdlib.h>

int
main(int ac, char *av[])
{
    FILE * ifd = fopen(av[1], "r");
    if (ifd == NULL) perror("fopen"), exit(EXIT_FAILURE);
    FILE * ofd = fopen(av[2], "w");
    if (ofd == NULL) perror("fopen"), exit(EXIT_FAILURE);

    int bread;
    char buf[65536];
    while ((bread = fread(buf, 1, sizeof buf, ifd)) > 0)
        if (fwrite(buf, 1, bread, ofd) != bread)
            perror("fwrite"), exit(EXIT_FAILURE);

    return EXIT_SUCCESS;
}
```

a) (2 pts) Which of the two versions uses Standard I/O and which version uses Unix I/O?

Version 1 uses Unix I/O and Version 2 uses Standard I/O.

b) (2 pts) Neither of the two programs closes the file descriptors that were opened (e.g., there are no calls to close() and/or fclose()). Could this omission cause either program to run incorrectly (i.e., the file not be correctly copied)? Check one box!

- ☑ Both programs run always correctly despite of this.
- ❑ Version 1 runs correctly, version 2 might not.
- ❑ Version 2 runs correctly, version 1 might not.
- ❑ Neither version runs correctly all the time.

When the program returns from main(), as both versions do, it calls exit(). Upon exit(), all standard I/O FILE* objects are closed and any buffered data is flushed to disk. In the Unix I/O version, no user-level buffering is done, so the data has already reached the kernel and will be written to the destination file.

c) (2 pts) Could the omission of the close/fclose calls cause resource leaks down the road? Check one box!

- ❑ Yes, both programs cause leaks.
- ❑ Version 1 might cause leaks, version 2 does not.
- ❑ Version 2 might cause leaks, version 1 does not.
- ☑ Neither version causes leaks.

Any FILE * objects allocated by standard I/O lie in user space, which like all user-allocated memory is discarded in its entirety when a process exits. Separately, the OS closes all of a process's file descriptors when it exits and deallocates any

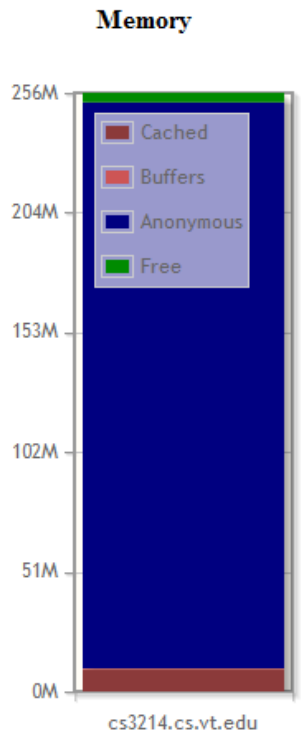
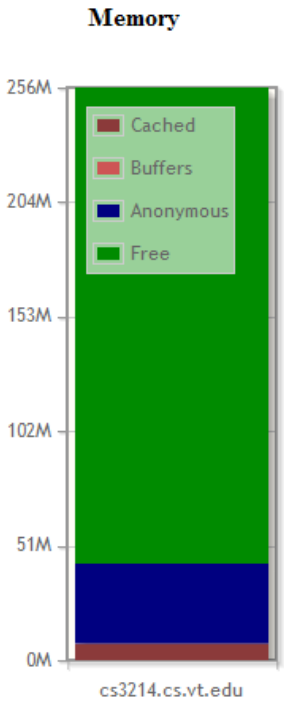
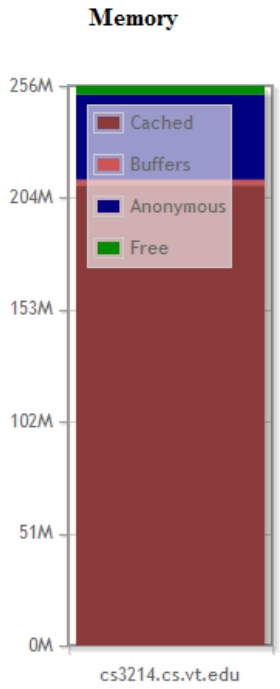
data structures the kernel may have kept. This deallocation happens even if a process is terminated by a signal. Although good programming style demands that a programmer closes files and/or stdio streams, and doing so is necessary for any long-running program, it would be unacceptable to have leaks due to simple programming flaws like this one after a program exits.

d) (4 pts) Assume a system in which system calls are comparatively expensive, and assume that large files are being copied. Under these conditions, would either of the two versions run significantly faster than the other? Justify your answer!

Any performance difference is likely to be marginal. The Unix I/O is already written in a way that it issues large read() and write() requests so that there is no need for, or expected benefit from, the buffering performed by standard I/O. The situation would be different if the Unix I/O version used much smaller buffers or wrote 1 byte at a time. In addition, the standard buffer size (BUFSIZ) is 8192, which is smaller than the size used in the example, so it's likely that fread() and fwrite() will bypass buffering altogether and simply directly invoke the read()/write() system calls.

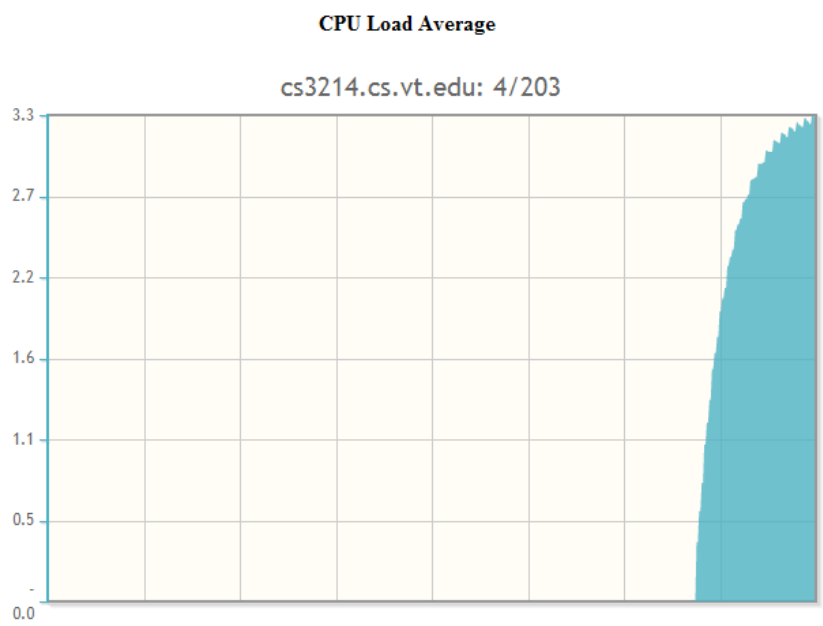
4. sysstatd Diagnostics (18 pts)

In project 5, you implemented a JSON-based web service that reported a Linux machine's physical memory and CPU usage, which could be integrated into web pages. In this question, you are asked to extend this service to implement additional functionality. To be able to more easily verify that your service reports meaningful and live data, you should implement additional service URLs that affect the machine's CPU and/or physical memory consumption.

		
The large area is physical memory dedicated to program data (“**Anonymous**”)	The large area is “**Free**” physical memory, available for immediate use	The large area is physical memory that caches file data (“**Cached**”)
`/anonymous?amount=250000`	`/free?amount=250000`	`/cached?amount=250000`

The unit in `amount` is in KB. `/anonymous?amount=250000` means, for instance, that your service should attempt to get the system to devote 250000KB to anonymous memory. The amount need not be met exactly.

Hint: The value reported by /proc/meminfo represents the current physical memory usage by all processes running on this machine. Assume on-demand paging and a global page replacement strategy as in standard Linux.



`/load?target=3`

(Here, 3 is the desired target value for the load average. Assume the machine is initially unloaded.)

The service URLs for each effect are shown below. For instance, if your service is running at kefka:20001, then visiting the URL http://kefka:20001/load?target=3 should produce the change shown above in the widget display. (To avoid creating lasting resource impact on the machine on which these diagnostic services execute, all services should have a timeout associated with them after which their action is undone, if possible. For the purposes of this question, *you need not show how* to implement such as timeout.)

Answer the following questions:

a) (4 pts) What changes would you need to make for all 4 proposed services to integrate them into your existing sysstatd framework? Be specific about which part of your HTTP protocol processing code you'll have to change and how!

The code in which each HTTP request's path is parsed would need to be extended to recognize the additional paths "cached", "anonymous", "free", and "load." In addition, the query string tokenizer would need to capture the 'amount' and 'target' keys and extract their values.

For all following parts, you may use either concrete code, or pseudo code, or a description using specific terms, including necessary API calls as appropriate!

b) (4 pts) Suppose you have identified that the user visited the /anonymous diagnostics URL. How would you implement this diagnostic service?

Using malloc(amount) (or an equivalent call to mmap() or sbrk()), and then touching each allocated page will cause the system to devote the desired amount of anonymous memory. Note that simply calling malloc() will not achieve this effect because of on-demand allocation of physical memory.

c) (2 pts) Suppose you have identified that the user visited the /free diagnostics URL. How would you implement this diagnostic service?

Same steps as /anonymous, but performed in a separate child process which then exits after touching each page. Or, if mmap() was used within the sysstatd process, munmap(). Note that simply calling free() may or may not immediately free the physical memory and thus may not have the desired effect.

d) (4 pts) Suppose you have identified that the user visited the /cached diagnostics URL. How would you implement this diagnostic service?

Read or write data from/to different files until the desired amount is reached. For instance, the service could traverse the root file system and read() and discard each file's content, just to force the OS to bring the file data into physical memory.

e) (4 pts) Suppose you have identified that the user visited the /load diagnostics URL. How would you implement this diagnostic service?

Spawn 'target' many threads or processes and have each thread execute an infinite `while (1);` loop. This ensures that this many threads are either in the READY or RUNNING state, contributing to the load average.
(Note that spawning a thread and have it sleep is a wrong answer – sleeping threads are in the BLOCKED state, thus they don't contribute to a machine's load.)

5. Short Questions (20 pts)

a) (4 pts) Creating a file involves executing a system call in the kernel. It is impossible to implement file creation as a library function without resorting to system calls. Explain why!

Creating a file involves changes to the file system stored on some underlying medium, which is a shared resource for which only the kernel provides protection and manages access. Accessing the I/O hardware also typically requires privileged instructions that can execute only in kernel mode.

b) (4 pts) Are all recursive functions automatically also thread-safe? Briefly say why or why not!

No. For instance, recursive functions that access global variables may not be. Only recursive functions that operate only on their arguments and local variables are thread-safe; those are said to be reentrant.

c) (4 pts) Why can a Java garbage collector move objects to a different address in memory whereas a C memory allocator cannot?

Because Java supports only typed references and does not support the conversion of integers to pointers; thus, all references to an object can be updated by the virtual machine when an object is moved.

d) (4 pts) How might global climate change be related to the strategy of busy waiting?

According to prevailing theory, global climate change may be caused by greenhouse gases such a carbon dioxide, which is created through the burning of fossil fuels. Busy waiting prevents a thread from moving to the BLOCKED state, thus preventing the OS from reducing a machine's power consumption by placing unused cores in a low-power idle state.

e) (4 pts) Assuming a sufficient supply of cores, would it be feasible and meaningful for an operating system to parallelize the servicing of page faults?

No. Page faults cannot generally be predicted and they need to be serviced synchronously; that is, there is nothing a thread could usefully do while a page fault is being handled.
(Note that this does not mean that prefetching of pages expected to be accessed in the future is without benefit. The question asked about servicing a page fault that has already occurred.)

6. Essay Question: Next Generation Bittorrent (16 pts)

Suppose you are hired by a company to oversee the technical development of the overlay transport layer for a next generation Bittorrent project. Bittorrent is a peer to peer (P2P) file sharing network that allows Internet users to exchange files with little or no centralized control by sending them to each other in a reliable fashion. The deployment time frame for your product will be 2012 and it is expected to be in use for several years.

Discuss the implications of the current transition from IPv4 to IPv6, and simultaneously the still wide-spread use of IPv4 NAT, for your overlay design and for the development of this application!

Note: *This question will be graded both for content/correctness (10 pts) and for your ability to communicate effectively in writing (6 pts). Your answer should be well-written, organized, and clear.*

No answer is provided. A correct answer would discuss the need for a protocol-independent implementation of all TCP/IP functions, as well as the complications arising from the fact that it's impossible or difficult to initiate direct TCP connections to machines located behind NAT gateways. This would either require the users to set up port forwarding, or the use of connection reversal (if one machine is public) or hole punching techniques. Relay servers would likely be inappropriate, given the P2P design of Bittorrent.