Sample Final Exam (Fall 2009)

Solutions are shown in this style. This exam was given Fall 2009

1. System Calls (16 pts)

a) (12 pts) Consider the following interaction of a user running bash in a terminal. In the table below, list the following events: (1) process-related system calls (fork, exec, exit, kill, wait/waitpid) and (2) signals delivered to a process (such as SIGCHLD, SIGINT, SIGTSTP, SIGTERM).

User input **shown in bold,** terminal output *in italics*	Shell Process	Child Process$_1$	Child Process$_2$
`$ ls`	*fork()*		
		exec("ls")	
		exit()	
	waitpid()/ SIGCHLD		
`$ sleep 100 &` `[1] 29598`	*fork()*		
		exec("sleep")	
`$ sleep 200 &` `[2] 29599`	*fork()*		
			exec("sleep")
`$ jobs` `[1]- Running sleep 100 &` `[2]+ Running sleep 200 &`			
`$ fg %2` `sleep 200`	*waitpid()*		
User types ^C			*SIGINT*
	SIGCHLD +waitpid()		
	waitpid()		
`$ fg %1` `sleep 100`	*waitpid()*		
User types ^Z		*SIGTSTP*	
	SIGCHLD +waitpid()		
`[1]+ Stopped sleep 100`			
`$ kill %1`	*kill()*		
		SIGTERM	
	SIGCHLD +waitpid()		
`$ jobs`			

[1]+ Terminated sleep 100			

Use different rows to express if events are synchronized, i.e., if it is guaranteed that event n occurs after event m, then event n should be listed in a row below event m. Note that not all rows/columns will have entries.

b) (4 pts) Explain briefly the key difference between the terms “background process” and “stopped process”!

A background process is one whose process group is not currently the foreground process group of its controlling terminal (or a process that has been detached from its controlling terminal). A background process can perform computation, consume CPU time, or be blocked while doing I/O or engaging in synchronization with other processes.

A stopped process is a background process that is in the stopped state – it is not scheduled by the OS onto any CPU, cannot perform computation or I/O until it is moved out of the stopped state via a SIGCONT signal or terminated.

2. Multithreading (18 pts)

a) (10 pts) Consider a fixed thread pool such as the one you implemented in exercise 11. Such a thread pool creates a fixed number of threads that process submitted tasks in FIFO order. Each task (or “callable”) is represented by a C function that receives a pointer to a custom argument.

```
/* Data to be passed to callable. */
struct callable_data {
   int number;
   sem_t *next, *previous;
};

/* A callable. */
void *
callable_task(struct callable_data *
callable)
{
   sem_wait(callable->previous);
   printf("Task %d ran.\n",
        callable->number);
   sem_post(callable->next);
   return NULL;
}

int
main(...) {
  ...
   const int N = ntasks;
   sem_t s[N + 1];
```

```
// .... main continued
   // create N callable tasks
   struct callable_data * callable_data[N];
   for (i = 0; i < N; i++) {
      callable_data[i] = malloc(
                    sizeof *callable_data[i]);
      callable_data[i]->number = i;
      callable_data[i]->next = &s[i + 1];
      callable_data[i]->previous = &s[i];
   }

   // submit tasks to thread pool
   for (i = N - 1; i >= 0; i--) {
      printf("Submitting task %d:
              next=%d, previous=%d\n",
         callable_data[i]->number,
         callable_data[i]->next - s,
         callable_data[i]->previous - s
      );
      thread_pool_submit(ex,
         (thread_pool_callable_func_t)
             callable_task,
         callable_data[i]);
   }
```

```
    // initialize N + 1 semaphores
    for (i = 0; i < N + 1; i++)
        sem_init(&s[i], 0, 0);
```

```
    printf("Posting first semaphore\n");
    sem_post(&s[0]);
    sem_wait(&s[N]);
    printf("Done.\n");
}
```

As in exercise 11, the number of threads and tasks can be varied.

i. (5 pts) Suppose this program is run with 4 threads and 2 tasks. The first three lines the program outputs are shown below. Describe what output, if any, the program will produce next. If no output is produced or if the output is not deterministic, state that!

```
$ ./threadpool_test 4 2
Submitting task 1: next=2, previous=1
Submitting task 0: next=1, previous=0
Posting first semaphore
Task 0 ran.
Task 1 ran.
Done.
```

In this example, each task N waits for task N-1 to signal a shared semaphore ("previous"), then signals a semaphore that is shared with task N+1 ("next"). The main thread signals the semaphore on which task 0 is waiting, then waits for the last semaphore to be signaled. This is a classic example of using semaphores to express precedence constraints.

ii. (5 pts) Now suppose the program is run with 4 threads and 5 tasks. Describe what output, if any, the program will produce next. If no output is produced or if the output is not deterministic, state that!

```
$ ./threadpool_test 4 5
Submitting task 4: next=5, previous=4
Submitting task 3: next=4, previous=3
Submitting task 2: next=3, previous=2
Submitting task 1: next=2, previous=1
Submitting task 0: next=1, previous=0
Posting first semaphore
```

In this example, deadlock will occur because the tasks are inserted in decreasing order (first task 4, then task 3, etc.) Task 4 waits for 3, task 3 waits for 2, task 2 for task 1, and task 1 waits for task 0. Since the thread pool, as stated, will handle tasks in FIFO order and since it is limited to handling 4 tasks simultaneously, task 0 cannot be run until at least one of tasks 1-4 completed. Since these tasks depend on task 0, a deadlock occurs.

b) (8 pts) What output do the following 2 programs produce and why?

```
#include <pthread.h>
```

```
#include <pthread.h>
```

```c
#include <stdio.h>

int counter;

static void * thread_func(void * _tn)
{
   int i;
   for (i = 0; i < 100000; i++)
      counter++;
   return NULL;
}

int
main()
{
   int i, N = 5;
   pthread_t t[N];
   for (i = 0; i < N; i++)
      pthread_create(&t[i], NULL,
                  thread_func, NULL);

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

   printf("%d\n", counter);
   return 0;
}
```

Outputs: (4 pts)

The output is not deterministic – 5 threads are accessing a shared but unprotected variable concurrently. A blatant race condition.

```c
#include <stdio.h>

int counter;

static void * thread_func(void * _tn)
{
   int i;
   for (i = 0; i < 100000; i++)
      counter++;
   return NULL;
}

int
main()
{
   int i, N = 5;
   pthread_t t[N];
   for (i = 0; i < N; i++) {
      pthread_create(&t[i], NULL,
                  thread_func, NULL);

      pthread_join(t[i], NULL);
   }

   printf("%d\n", counter);
   return 0;
}
```

Outputs: (4 pts)

The output is 500000. In this case, the 5 thread do not execute concurrently, but in sequence. Each thread is started only after the previous one has exited (and was joined by the main thread.)

3. Memory Management (16 pts)

a) (4 pts) Consider the following program:

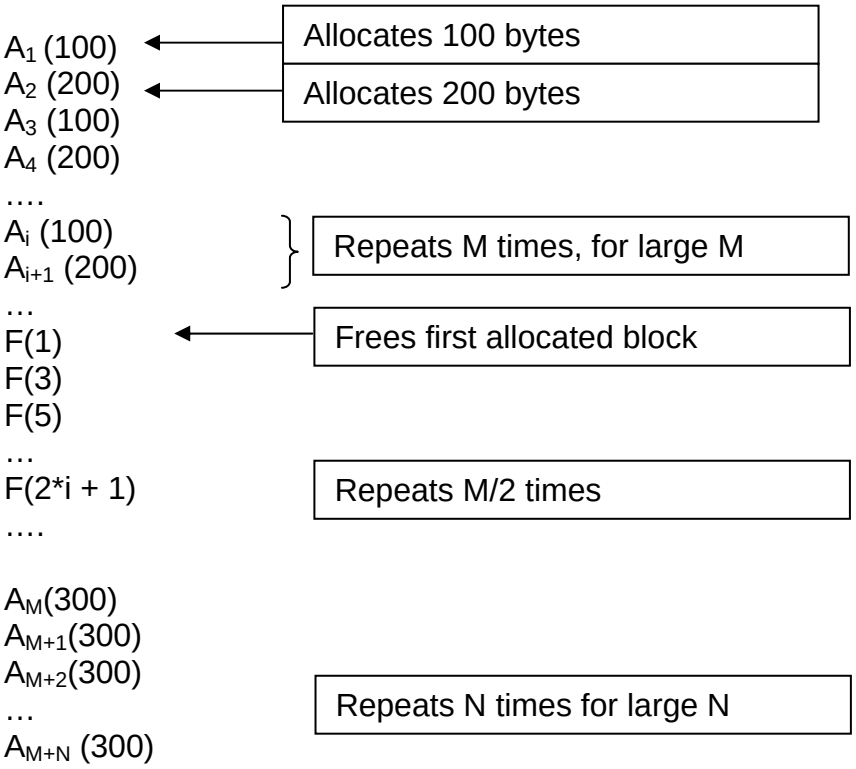
```c
#include <stdlib.h>
#include <stdio.h>

int
main(int ac, char *av)
{
    size_t *p = malloc(1);
    printf("0x%08x\n", p[-1]);
}
```

Its output, when run under a current version of Linux, is **`0x00000011`**.
Explain the likely reason for this specific output value!

The most likely reason is that the GNU C library's memory allocator uses Knuth's boundary tag method. The boundary tag must be located at a fixed offset from the payload (here: -sizeof(size_t) bytes). 0x00000011 may encode the size of this block and perhaps the status of this block or the preceding block. Likely, a 16-byte block (size = 0x10) was used to satisfy this allocation request.

b) (8 pts) Consider a dynamic storage allocator like the one you implemented in project 4. Consider the following sequence of allocation requests A_i(size) and matching free requests F(i):



i. (4 pts) What performance would you expect if a single, explicit free list is used for this trace, and why?

This workload alternately allocates many small (100) and large (200) byte chunks, then frees all small chunks. Afterwards, the single free list will contain M/2 free 100 byte chunks. This large list must be traversed for each of the following N requests only to find that no free block is large enough to satisfy the 300 byte allocation requests. This results in an extraordinary amount of runtime spent traversing the single free list to find a suitable block.

ii. (4 pts) Suggest a method to improve the performance of the allocator for this trace!

Key is avoiding to scan the long free list of 100 bytes block to check if any of them is large enough to hold 300 bytes. This could be accomplished by using any segregated scheme in which 100 and 300 fall in different size classes.

c) (4 pts) Based on what you learned about explicit dynamic storage management (i.e., using malloc()/free() or equivalent functions) and automatic storage management (as used in Java, via new and garbage collection), would you agree or disagree with the statement that "a long-running program using garbage collection can outperform a version of the same program using explicit memory management?"
Justify your answer!

Yes, it is definitely possible for programs using garbage collection to outperform those using explicit memory management. This is particularly true if large numbers of temporary, short-lived objects in the nursery of a generational collector are created, which a garbage collector often can summarily discard, whereas an explicit collector will have to handle each such object individually, updating free lists and performing coalescing as necessary. In addition, the ability of most automatic memory management schemes to compact objects can lead to more efficient allocators since free lists don't have to be searched for free objects. Lastly, garbage collection can be more easily parallelized, something more difficult to do in explicit schemes.

4. Virtual Memory (16 pts)

a) (6 pts) Application programmers rarely notice the existence of virtual memory. As is stated in Chapter 10 of the textbook, virtual memory works "silently and automatically, without any intervention from the application programmer." Give 2 examples of such silent and automatic virtual memory functionality!

Examples of such functionality includes:

- *On-demand loading of executables (bring in code as a program executes it)*
- *Automatic stack growth (adding stack page as a program accesses them)*
- *Paging (moving program data to and from disk as needed)*
- *Caching of file data (whether or not mmap() is used)*
- *Protecting a process's data from unauthorized access by other processes*

b) (4 pts) Do you agree or disagree with the following statement?
"The OS guarantees that a user program can make *efficient* use of *all* memory allocated by malloc()."
Justify your answer, stating your assumptions if necessary!

No. Memory allocated with malloc() refers to virtual pages that may or may not be backed by physical memory. The OS may decide to evict pages, causing a performance penalty on subsequent accesses that bring those pages back into memory. A program's accesses to memory may cause thrashing if the working set (the subset of pages accessed in some recent

period) grows beyond what can be held in physical memory, in which case memory accesses slow down by an order of magnitude (the speed of the disk).

Moreover, as I had demonstrated in class, Linux does not even check if there's enough physical memory + swap space system wide to hold all data that a user program would store in the malloc()'d space. It may even be the case that the OS decides to terminate a process if too many processes start using all virtual addresses obtained via malloc(). Even systems such BSD or Solaris that use more conservative policies when deciding whether to grant virtual address requests do not guarantee effective use – thrashing may occur there as well if the frequence with which data is paged in/out grows.

c) (3 pts) Name 1 purpose of the mmap() system call!

mmap() maps a file's data into the address space of a process, allowing the file data to be accessed like conventional memory.

mmap() can also be used to create anonymous regions of virtual address space that is backed by swap space rather than a specific file. This facility is used by malloc() to allocate large chunks of heap memory (typically, for malloc() with size >= 256K).

d) (3 pts) A virtual memory system can be viewed as a cache in which physical memory holds some portion of a larger set of data that is stored on disk. This cache is fully associative.
Explain in one sentence what "fully associative" means in this context!

"Fully associative" means that every physical page frame can hold the data of any virtual page. In other words, there are no restrictions on where to place program or file data in physical memory.

5. Networking (18 pts)

a) (8 pts) The Java package java.net provides a number of classes that expose the BSD socket API in Java. The classes rely on implementations in C contained in native libraries. Based on the documentation of the classes and based on your knowledge of networking, sketch the implementation of the following methods/constructors! Provide all specific C functions you would find in the actual implementation!

```
package java.net;
class Socket {
      /** Creates a stream socket and connects it to
         the specified port number on the named host. */
      public Socket(String host, int port)
             throws UnknownHostException, IOException {
          /* insert part i) here (4 pts) */
         /* these are not the actual calls to these
            functions, just a sketch */
         s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
         /* alternatively, gethostbyname() can be used */
         getaddrinfo(host, port, &addr);
```

```
            connect(s, &addr, ..)
        }
    }

class ServerSocket {

    /** Creates a server socket and binds it to the specified local port
    number, with the specified backlog. A port number of 0 creates a
    socket on any free port.

    The maximum queue length for incoming connection indications (a
    request to connect) is set to the backlog parameter. If a connection
    indication arrives when the queue is full, the connection is
    refused. */
    public ServerSocket(int port, int backlog) throws IOException {
       /* insert part ii) here (4 pts) */

            /* these are not the actual calls to these
               functions, just a sketch to provide the essence */
            s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
            // alternatively, getaddrinfo can be used
            addr.sin_addr = INADDR_ANY;
            addr.port = htons(port);
            bind(s, &addr, ..)
            listen(s, backlog);
    }
}
```

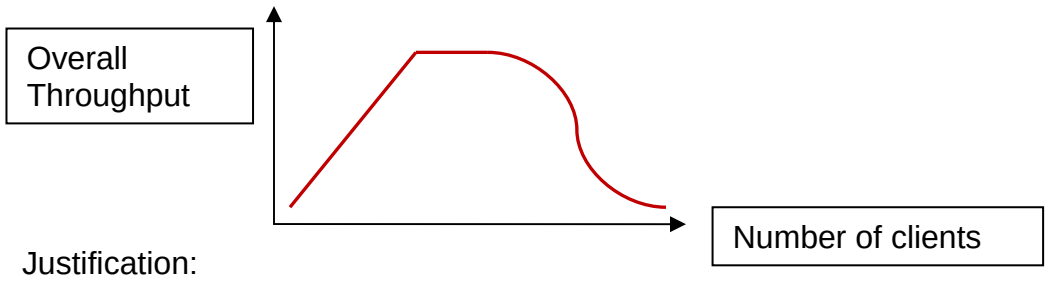
b) (10 pts) Consider the web service in project 5, which supported multiple clients and persistent HTTP/1.1 connections (if you didn't implement support for multiple clients and/or persistent connections, answer the question as if you had). In this question, you are asked to consider the performance of your service as the number of connected clients and the frequency with which they issue HTTP requests changes. Provide a chart with the performance curve you would expect, and provide a short justification. State your assumptions, or provide details of your implementation, as necessary!

i. (5 pts) First, consider what happens as the frequency of requests increases (without an excessive number of simultaneous clients):

Justification:

For the suggested implementation (which spawns one thread per connection), throughput should increase as the request frequency increases, up to a maximum that is determined by the next bottleneck encountered. For the specific web service you implemented (which produces its content entirely in memory, and requires a few system calls to retrieve the necessary information from the kernel), it's likely that the bottleneck will be the speed of the outgoing network link, so its bandwidth will determine the maximum. It's also likely that you'll stay at this maximum and do not experience a drop-off – the maximum frequency with which clients can issue requests would be back-to-back.

ii. (5 pts) Second, consider what happens as the number of clients increases:



Justification:

Assuming again a one-thread per client implementation, you would see similar increase with an increase in the number of clients, but the fact that you're creating new threads for each connection will likely lead to a drop-off in throughput beyond a certain number of clients, unless more sophisticated resource management methods are used.

6. Essay Question: Virtualization Trade-Offs (16 pts)

Consider the spectrum of multiprogramming arrangements ranging from multi-threaded, single-process applications on one end and dedicated Type I hypervisors that can support multiple guest operating systems on the other end! Describe this spectrum and outline how moving along the spectrum changes which hardware resources are abstracted or virtualized! Explain the trade-offs involved, and describe using examples which characteristics of a given application scenario influences the choice of multiprogramming or virtualization model!

Note: *This question will be graded both for content/correctness (10 pts) and for your ability to communicate effectively in writing (6 pts). Make sure you define the spectrum and the involved trade-off clearly and elaborate on its meaning and consequences. Your answer should be well-written, organized, and clear.*

No answer provided for this question.

In grading this question, I looked for your ability to recognize that multi-threaded programs, single-threaded programs, Type II and Type I hypervisors are points in a design space that provide different degrees of resource virtualization and separation. A common mistake was to simply relay implementation details about virtualization, rather than discussing the trade-offs in choosing a point in the spectrum. Many students failed to relate their discussion to application scenarios.