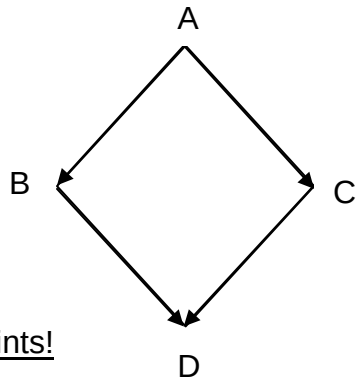
Sample Final Exam (Fall 2011)

This exam was given Fall 2011.

1. Multithreading (29 pts)

a) (12 pts) *Semaphores*. Semaphores can be used to express scheduling constraints between activities performed by different threads. Consider the diamond shown on the right, denoting the dependencies between activities A, B, C, and D, which are executed by 4 different threads.



Complete the program below to ensure these constraints!

```
// declare any semaphores you need here;
// be sure to show how to initialize them

// initial value of all semaphores is 0,
// for proper code see main function; shorthand is accepted
sem_t a_done, b_done, c_done;
```

```
static void *
thread_A(void *_) {
    printf("A\n");
    sem_post(&a_done);
    sem_post(&a_done);
}
```

```
static void *
thread_B(void *_) {
    sem_wait(&a_done);
    printf("B\n");
    sem_post(&b_done);
}
```

```
static void *
thread_C(void *_) {
    sem_wait(&a_done);
    printf("C\n");
    sem_post(&c_done);
}
```

```
static void *
thread_D(void *_) {
    sem_wait(&b_done);
    sem_wait(&c_done);
    printf("D\n");
}
```

```
int main()
{
    sem_init(&a_done, 0, 0);
    sem_init(&b_done, 0, 0);
    sem_init(&c_done, 0, 0);
    pthread_t t[N];
    pthread_create(t+0, NULL, thread_D, NULL);
    pthread_create(t+1, NULL, thread_C, NULL);
    pthread_create(t+2, NULL, thread_B, NULL);
```

```
    pthread_create(t+3, NULL, thread_A, NULL);
    pthread_exit(0);
}
```

b) (6 pts) *Condition Variables.* The first edition[1] of Thomas/Hunt's book "Programming Ruby – The Pragmatic Programmer's Guide" (the so-called "pickaxe" book) contained the following example of how to use condition variables in Ruby:

```
require 'thread'
mutex = Mutex.new
cv = ConditionVariable.new

a = Thread.new {
  mutex.synchronize {
    puts "A: I have critical section, but will wait for cv"
    cv.wait(mutex)
    puts "A: I have critical section again! I rule!"
  }
}

puts "(Later, back at the ranch...)"

b = Thread.new {
  mutex.synchronize {
    puts "B: Now I am critical, but am done with cv"
    cv.signal
    puts "B: I am still critical, finishing up"
  }
}
a.join
b.join
```

The accompanying description read:

"A condition variable is simply a semaphore that is associated with a resource and is used within the protection of a particular mutex. When you need a resource that's unavailable, you wait on a condition variable. That action releases the lock on the corresponding mutex. When some other thread signals that the resource is available, the original thread comes off the wait and simultaneously regains the lock on the critical region."

i. (3 pts) To which misunderstanding of condition variables did the authors of this book fall victim? (You don't need to know Ruby to answer this question.)

They confused condition variables with semaphores. Unlike semaphores, condition variables do not remember past signals, so if no thread is waiting on a

[1] See Hhttp://www.ruby-doc.org/docs/ProgrammingRuby/html/tut_threads.html

condition variable when it is signaled, the signal is lost. That's why a condition variable must always be used in connection with an actual programmer-defined condition, something this example fails to do. Note the lack of a while() statement before the cv.wait(mutex) call.

ii. (3 pts) Because of this lack of understanding of condition variables, explain how the example program provided by the authors could fail; how would this failure manifest itself in its execution?

The program could deadlock if thread 'b' executes its body before thread 'a', which is entirely possible. In that case, thread 'a' would never return from from wait() if the corresponding signal() already happened.
To the authors' credit, this error was removed from subsequent editions, and in fact recent versions of Ruby provide direct support for the monitor abstraction using a special monitor class that embodies the correct way of using this pattern.

c) (6 pts) Consider the following Java program:

```
public class IMS implements Runnable
{
    boolean signaled;
    @Override
    public void run() {
        try {
            while (!this.signaled)
                this.wait();
        } catch (InterruptedException e) { }
        System.out.println("Signaled!");
    }

    public static void main(String []av) throws InterruptedException {
        IMS r = new IMS();
        new Thread(r).start();
        Thread.sleep(1000);
        r.signaled = true;
    }
}
```

When run, the program reports:

```
$ java IMS
Exception in thread "Thread-0" java.lang.IllegalMonitorStateException
        at java.lang.Object.wait(Native Method)
        at java.lang.Object.wait(Object.java:503)
        at IMS.run(IMS.java:8)
        at java.lang.Thread.run(Thread.java:722)
```

i. (2 pts) Why is the error message referring to the term 'monitor' in the name of the error ("IllegalMonitorStateException")?

The term 'monitor' refers to the parallel programming abstraction invented by Hoare/Brinch Hansen from which condition variables, and in particular the Object.wait() method, are derived. By allowing every object to be used as a mutex and a condition variable associated with that mutex, Java's designers attempted to provide support for the monitor pattern.

ii. (2 pts) What concrete programming mistake is causing the error message?

The object 'this' was never locked, e.g. synchronized upon. The monitor pattern does not make sense if wait() is called from outside the monitor, i.e., without holding the monitor lock.

iii. (2 pts) What's the equivalent mistake when using C/Pthreads condition variables?

Calling pthread_cond_wait() without having acquired the mutex passed to pthread_cond_wait(). Note that unlike in Java, in most POSIX implementations, the result by default is undefined.

d) (5 pts) In class, we had talked about the perils of busy waiting and that it should usually be avoided at all costs. In this question, you are asked to come up with one reasonable example where busy waiting is an acceptable, or perhaps even the only approach to a synchronization problem in a concrete environment. Let us exclude synthetic loads such as project 6's /runloop service! State your assumptions as necessary!

There are at least two situations where one might consider busy waiting: first, if one can be certain, or reasonably expect, that the overall cost of busy waiting is less than the cost of synchronization primitives. That's the case for so-called spin locks, which are held for only short periods of time. In these situations blocking, and later unblocking, the thread attempting to acquire the lock would be less efficient than simply waiting, provided the lock holder executes on a different CPU or core. I should note that this applies to kernel code only; there is rarely a chance to use spinlocks in user code for the simple reason that user threads lack control over the scheduler.

Second, busy waiting is the only choice when the underlying system does not provide a notification mechanism that can be tied to the event one is waiting for. For example, if a thread needs to wait until a new file in a directory is created, but the underlying OS does not provide a directory change notification facility, busy waiting (or it's less extreme form, periodic polling) may be the only option.

2. Virtual Memory (20 pts)

a) (5 pts) The Google Android Operating System uses a Linux kernel to support its applications. Each application runs in its own process. Android devices do not typically have a disk, and they (typically) do not use swapping to stretch the amount of available physical memory when they run short. Instead, the OS may terminate processes, requiring the applications to store and restore their state when this happens. Under these circumstances, would Android still derive any benefits from Linux's virtual memory capabilities? Justify your answer!

Even though Android does not typically swap, it still benefits from the other services virtual memory provides: for instance separate address spaces, which protect and isolate applications from one another, or the ability to share memory between processes so that the Dalvik runtime environment needs to be loaded only once.

b) (5 pts) Modern operating systems use nearly exclusively on-demand paging in which a process's code and data is not brought in until it is needed. In Unix, what's the very first page fault a process encounters after it makes a successful exec() system call?

The very first page fault will be at the new program's entry point in its text segment where the new program starts executing and fetching instructions (typically a routine called 'start' which calls 'main') That's when the system will load the executable from disk (unless it's already in use by some other process).

c) (10 pts) Suppose a Java program contains a memory leak. The Java program is executed on a Java virtual machine which runs in a process in a system that uses virtual memory.

i. (6 pts) Based on your knowledge of automatic memory management in languages such as Java, what impact would you expect the application-level memory leak to have on the performance of the application, particularly with respect to its use of virtual memory?

A memory leak causes an increase in the live heap size, which in turn may cause more frequent full garbage collections. During the mark phase of each such garbage collection, the collector will touch all pages that contain live objects, making the OS think that these pages belong to the process's working set. As a result, its physical memory requirements grow, making it more likely that the system will evict pages to accommodate these requirements. A slowdown in performance results, particularly if physical memory is exhausted, leading to thrashing in the worst case.

ii. (4 pts) Is it possible that other process' performance would be affected by this leak? State your assumptions if necessary!

That depends on the page replacement policy the system pursues. Some systems, notably Linux, use a global replacement policy in which one process's page fault can force another process's pages to be evicted, leading to slowdown for that process if it subsequently accesses those pages. A local replacement policy, such as the one used in Windows, defends against this effect by preferring a process's own pages when looking for victims.
This performance impact results from the shared nature of physical memory; another shared resource is CPU time. It is also correct to point out that a process's increased CPU needs (due to the increase in GC frequency) takes CPU time away from other processes, slowing down their progress.
A common wrong answer was to state that since each process has its own address space, their memory is isolated from one another. While this is true with respect to access, the question asked whether there is a performance impact in a system using virtual memory (where physical memory is shared).

3. Dynamic Memory Management (20 pts)

a) (8 pts) *A special purpose allocator.* Suppose all of an application's dynamically allocated objects follow a particular pattern of allocations and deallocations that is characterized by two properties. First, allocation and deallocation happen in a strict LIFO fashion, i.e., any more recently allocated object will be freed before, or simultaneously with, a less recently allocated object. Second, objects will be freed in groups of one or more objects without any intervening computation. For an example, consider the following memory allocation pattern, where … denotes sections of computation, A_i denotes allocation of object i, and $F_{\{j,k\}}$ denotes deallocation of objects j and k:

A_1, …, A_2, …, A_3, … A_4, …, $F_{\{3,4\}}$, …, A_5, …, A_6, …, $F_{\{6,5,2,1\}}$

Describe how you could implement an optimized allocator that exploits this application's particular allocation pattern! (Note that the number and size of allocated objects is not known beforehand, only the allocation pattern is!)

A so-called object stack allocator could be used which allocates memory using a chain of chunks. Within the last chunk, allocation is done simply by bumping a pointer; if the allocation request is larger than the amount remaining in the chunk, a new one is allocated chunk. Deallocation is very efficient – simply reset the the 'next allocation' pointer and chunk to the location of the lowest-number object and discard all more recent chunks, if any.

This scheme is used by the GNU obstack library and can lead to significant speedups when used in place of malloc() for objects that follow this allocation/lifetime pattern.

b) (12 pts) *Buddy allocators.* The Linux kernel uses a so-called binary buddy scheme for managing a machine's memory. The binary buddy scheme is a very simple allocator that uses a strict segregated-fit scheme. The kernel keeps 10 free lists for blocks of size 4KB, 8KB, 16KB, and so on to 1MB. The segregation is strict in that all blocks on a free list are of the same size. Allocation requests are rounded up to the next available free block size. If the free list for this size is empty, the free list of the next higher size is consulted until a non-empty free list is found. The splitting policy splits blocks in half and adds blocks to the appropriate free lists. For instance, suppose the 4KB, 8KB, and 16KB free lists are empty and a request for 3KB arrives. In this case, a 32KB block would be split into a 16KB block, a 8KB block, and 2 x 4KB blocks. One 4 KB block would be used to satisfy the allocation request, and the other would go on to the 4KB free list, and the 16KB and 8KB blocks would go onto their respective free lists.
Buddy allocators perform immediate coalescing. However, a block can only be coalesced with its buddy, which is of the same size and is located before or after it, depending on the block's address. For instance, the buddy of the 4KB block at 0x0000 is the 4KB block at 0x1000. The buddy of the 8KB block at 0x2e000 is the 8KB block at 0x2c000. If the buddy of a block of size N is free, the coalesced block will be added to the free list of blocks of size 2*N. In this way, all blocks of size N are aligned at multiples of N.
Unlike the list-based allocator you implemented in project 4, a buddy allocator does not use boundary tags to record whether a block is used or free; instead, a bitmap with one bit per 4KB page is used.

Analyze this scheme under the assumption that the workload faced by this allocator is not known!

i. (4 pts) Based on the complexity of its described allocation and free operations, what throughput would you expect from this scheme?

You would expect high throughput since both allocation and deallocation can be performed in constant time, even in the worst case. As a matter of fact, this is why this allocation scheme is sometimes used in real-time systems in which such runtime guarantees matter.

ii. (4 pts) How much internal fragmentation would you expect from this scheme?

Internal fragmentation includes the difference between allocated block size and payload. The buddy allocator always rounds up to a multiple of 4KB, so the internal fragmentation is between 0 bytes and 4KB-1; for uniformly distributed payload sizes it would be, on average, ~2KB.

iii. (4 pts) Is external fragmentation more or less likely to arise from this scheme than from the allocator you implemented in project 4? State your assumptions as necessary!

Definitely more likely. The limited coalescing ability of this allocator (for instance, it is unable to coalesce blocks of different sizes, even neighboring ones, and it is unable to coalesce with a free neighbor that is not a block's buddy) makes external fragmentation more likely than your p4 allocator.

4. HTTP and Web Servers (20 pts)

a) (5 pts) In a persistent HTTP/1.1 connection, why is the 'Content-Length:' header field necessary?

It is necessary to denote the boundaries between returned HTTP objects. HTTP/1.1 uses 1 connection for multiple objects, so it must have a way of saying where one object ends and the next one starts; it does this by counting the number of bytes in each object, which is placed in the Content-Length header field.

b) (10 pts) Suppose a browser connects to a server to retrieve a webpage. In its response, the server indicates that the connection supports HTTP/1.1; the server does not send a Connection: close header, thus encouraging the client to reuse the connection. After parsing the HTML, the browser finds that it has to retrieve several embedded objects for this webpage before it can be rendered and displayed to the user: Style-sheets, Javascript, images, flash, etc. The client has a choice between opening additional connections to the server, or pipelining the requests on the already established persistent connection.

i. (5 pts) Under which circumstances should the client prefer to open additional connections?

Whether multiple connections are more beneficial than pipelining on an existing connection depends on the length of the round-trip time when considered in relation to the overall time it takes to send a request and receive an object, and on whether the server has the ability and resources to handle multiple requests in parallel. Note that several connections can be established nearly in parallel, allowing the additional round-trip times needed to establish those TCP connections to at least partially overlap.

If an additional round-trip time does not make a major contribution to overall response time, and if the server has the resources (e.g., additional cores for CPU-bound requests, or the ability to use multiple threads or programming to overlay request processing with I/O), opening additional connections makes the most sense. As discussed in class, that's in fact the assumption most modern

browsers make. (They still use persistent connection and they still limit the number of connections to a server to protect the server's and the network's resources.)

The use of multiple connections also avoids the potential disadvantage of pipelining that a server cannot send the response to an already received pipelined request until they are ready to send the response to an earlier request; in fact, most servers will not even look at the next request until they are done with the previous one.

ii. (5 pts) Under which circumstances should the client prefer to pipeline additional requests on only one existing connection?

Pipelining in lieu of additional connections would make the most sense if round-trip times are large compared to overall response times (say for small services or objects), and/or when it is known that the server cannot handle multiple requests in parallel (e.g., a single-threaded, iterative server that handles only one client at a time, such as those often used in Internet-connected embedded devices.)

c) (5 pts) In project 6, you wrote a single-process, multi-threaded web server. Let us compare this approach to a fork-on-demand, multi-process approach. In such an approach, a new process is created for each connection to serve that connection. Outline how the multi-threaded approach you implemented made robust engineering for your project more difficult!

One key problem that you all faced was that when something went wrong with one connection, say an error in any of the system calls needed to service a request, you couldn't simply exit() the process because that would have shut down the entire server. Similarly, any unintended crash would have had the same effect.

There are a number of related issues (optimal concurrency control, race conditions, sharing file descriptors, etc.) that some answers brought up. Note that the per-process model does not necessary help here: optimal concurrency control is as challenging a problem with processes as it is with threads; if there's shared state, a multi-process solution needs to use shared memory and must deal with race condition just like a multi-threaded solution would, and sharing file descriptors across processes involves its own potential for errors (e.g. failing to close fds in the parent).

5. Essay Question: The NSF/IEEE TCPP Curriculum (16 pts)

IEEE's Technical Committee on Parallel Processing (TCPP), supported by funding from the National Science Foundation (NSF) and Intel and IBM Corporations, has been developing a Parallel and Distributed Computing Curriculum that seeks to *embed parallel and distributed computing throughout all courses in an undergraduate Computer Science (CS) curriculum*. The authors of this proposal write:

"In the past, it was possible to relegate issues regarding parallelism and locality to advanced courses that treat subjects such as operating systems, databases, and high performance computing: the issues could safely be ignored in the first years of a computing curriculum. But it is clear that changes in architecture are driving advances in languages that necessitate new problem solving skills and knowledge of parallel and distributed processing algorithms at even the earlier stages of an undergraduate career."

Discuss this claim! Do you agree or disagree? Justify your opinion!

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

No Solution Provided.