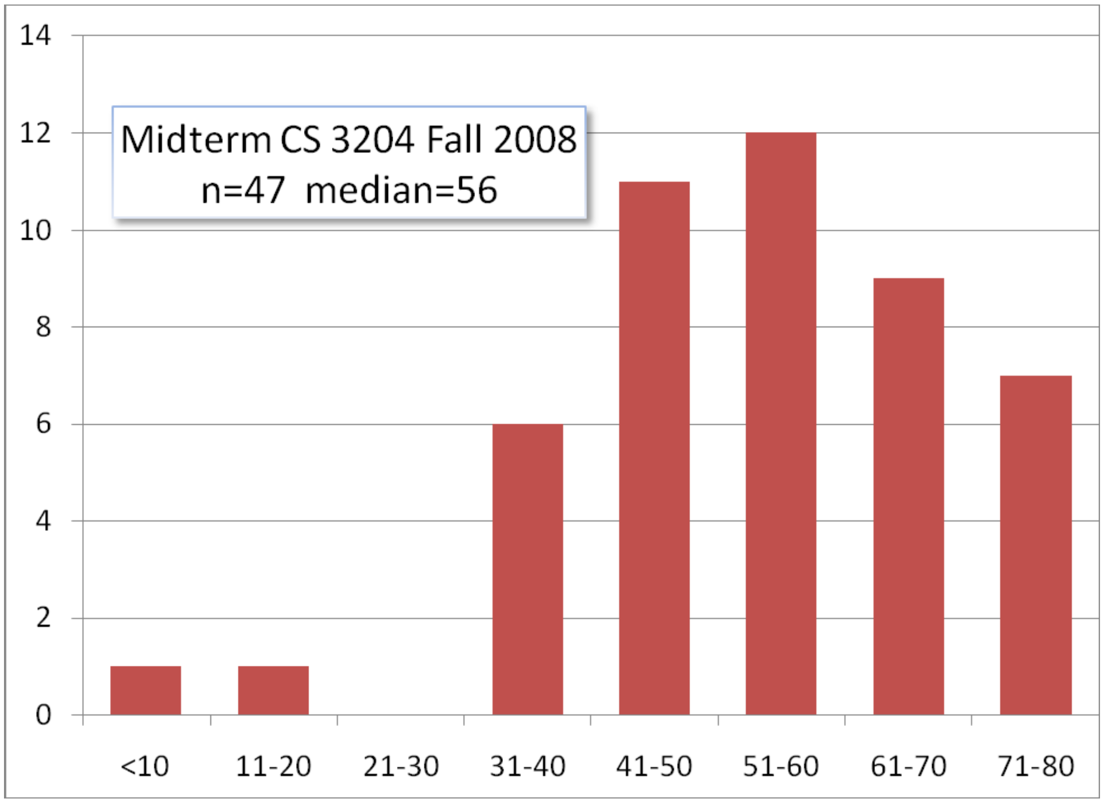
CS 3204 Midterm Solution

47 students took the midterm. The table below shows who graded which problem. If you have questions, read this handout first, then address your question to the person who graded it. If you can't find a resolution, come and see me.

Problem	1	2	3	4	Total
Possible Pts	24	26	26	24	**100**
Median	11	20	14	12	**56**
Average	11.3	18.9	13.5	10.5	**54.3**
StDev	7.0	5.4	5.6	5.8	**15.0**
Min	0	5	0	0	**8**
Max	24	26	24	24	**80**
Grader	Peter	Godmar	Peter	Godmar	

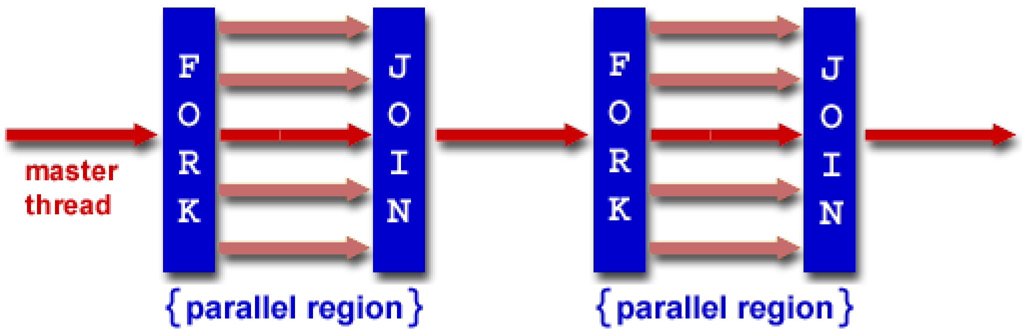
This midterm counts for about 15% of your grade. Although your final grade will depend on both exams and the project, students who have scored lower than 40 should be aware that they are at risk of failing this class. They will need to show significant improvement in the final exam. Students who have scored lower than 50 are at risk of receiving a grade lower than the C that is required by CS classes that have CS 3204 as a prerequisite.



Solutions are shown in this color.
Grading comments are shown like this.

1. Implementing OpenMP (24 pts)

The OpenMP Application Programming Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran. In C/C++, programmers can insert #pragma preprocessor directives in their code to instruct the compiler how to parallelize their code. OpenMP uses the fork/join parallelism model in which a master thread spawns off and later joins worker threads during regions of parallel execution, as depicted in the Figure below:



Source: openmp.org

In this question, you should apply your knowledge of thread and synchronization APIs and show how a compiler might implement parts of this specification by translating C/C++ with OpenMP #pragmas into C code. You may assume the presence of a Pintos- or POSIX Threads-style thread API for C, including thread_create() and thread_join() primitives, mutexes, semaphores, and condition variables (if needed).

a) (12 pts) One of the simplest OpenMP constructs is the parallel *for*-loop. For instance, to add two vectors, a programmer may write the following OpenMP code:

```
/* Add two vectors 'a' and 'b' and place the sum in 'sum'. */
void
vector_add(double *a, double *b, double *sum, int n)
{
    int i;
    #pragma omp parallel for
    for (i = 0; i < n; i++)
        sum[i] = a[i] + b[i];
}
```

The amount of work done in the loop is evenly distributed among the participating threads.
Show C code that may be produced by an OpenMP compiler when compiling the code above. Be sure to include all necessary

synchronization code and show how the master thread distributes the work among the worker threads.
To simplify the problem, you may assume that the number of threads `NT` is a constant, and that `n % NT == 0`. You may also assume that each parallel region spawns a new team of threads (though real-world OpenMP implementations reuse a pool of threads).

```
/* A unit of work – chunks of three vectors
 * ‘a’, ‘b’, and ‘sum’ of equal length ‘n’
 */
struct vector_chunk {
    double *a, *b, *sum;
    int n;
};

/* Add a chunk of two vectors */
static void *add_chunk(void *_chunk)
{
    struct vector_chunk *chunk = _chunk;
    int i;
    for (i = 0; i < chunk->n; i++)
        chunk->sum[i] = chunk->a[i] + chunk->b[i];
    free (chunk);
    return NULL;
}

/* Add two vectors 'a' and 'b' and place the sum in 'sum'. */
void
vector_add(double *a, double *b, double *sum, int n)
{
    int thread;
    tid_t threadids[NT];

    // start NT-1 threads and assign a chunk of the vector
    // addition to each thread
    for (thread = 0; thread < NT - 1; thread++) {
        struct vector_chunk *chunk = malloc(sizeof *chunk);
        chunk->a = a + thread * n/NT;
        chunk->b = b + thread * n/NT;
        chunk->sum = sum + thread * n/NT;
        chunk->n = n / NT;
        threadids[thread] = thread_create(add_chunk, chunk);
    }

    // set up work unit for master thread
    struct vector_chunk master_thread_chunk = {
        .a = a + thread * n/NT,
        .b = b + thread * n/NT,
        .sum = sum + thread * n/NT,
        .n = n / NT
    };
    add_chunk(&master_thread_chunk);

    // wait until all threads have finished their work
    for (thread = 0; thread < NT - 1; thread++) {
        thread_join(threadids[thread]);
    }
}
```

Note: even though the actual OpenMP standard requires that the master thread performs one of the chunks of work, we also accepted solutions for full credit that spawned NT threads and in which the master thread only waits for those threads to finish.

For full credit, you didn't need to show all details, just the essential approach of splitting work, and forking and joining threads to do it.

b) (12 pts) OpenMP also supports reductions in an efficient manner. For instance, the dot product of two vectors can be computed as follows:

```
/* Compute and return the dot produce of vectors 'a' and 'b'. */
double
vector_dot_product(double *a, double *b, int n)
{
    double dotproduct = 0;
    int i;

    #pragma omp parallel for reduction(+:dotproduct)
    for (i = 0; i < n; i++)
        dotproduct += a[i] * b[i];

    return dotproduct;
}
```

Show the code an OpenMP compiler would produce to implement reduction. Your code must be thread-safe, but *must* avoid unnecessary or excessive synchronization.

```
/* A unit of work – chunks of three vectors
 * 'a', 'b', and 'sum' of equal length 'n'
 */
struct vector_chunk {
    double *a, *b, dotproduct;
    int n;
};

/* Compute partial dot product for a chunk of two vectors */
static void *dot_product_chunk(void *_chunk)
{
    struct vector_chunk *chunk = _chunk;
    int i;
    double dotproduct = 0;

    for (i = 0; i < chunk->n; i++)
        dotproduct += chunk->a[i] * chunk->b[i];

    chunk->dotproduct = dot_product;
    return NULL;
}

/* Compute and return the dot produce of vectors 'a' and 'b'. */
double
vector_dot_product(double *a, double *b, int n)
{
    int thread;
    tid_t threadids[NT];
    struct vector_chunk *chunk[NT];

    // start NT-1 threads and assign a chunk of the vector
    // dot product to each thread
    for (thread = 0; thread < NT - 1; thread++) {
        chunk[thread] = malloc(sizeof *chunk);
        chunk[thread]->a = a + thread * n/NT;
        chunk[thread]->b = b + thread * n/NT;
        chunk[thread]->n = n / NT;
        threadids[thread] =
            thread_create(add_chunk, chunk[thread]);
    }

    // set up work unit for master thread
    struct vector_chunk master_thread_chunk = {
        .a = a + thread * n/NT,
        .b = b + thread * n/NT,
        .n = n / NT
    };
    dot_product_chunk(&master_thread_chunk);
    double dotproduct = master_thread_chunk.dotproduct;

    // wait until all threads have finished their work
    for (thread = 0; thread < NT - 1; thread++) {
        thread_join(threadids[thread]);
        dotproduct += chunk[thread]->dotproduct;
        free(chunk[thread]);
    }

    return dotproduct;
}
```

Note: even though the actual OpenMP standard requires that the master thread performs one of the chunks of work, we also accepted solutions for full credit that spawned NT threads and in which the master thread only waits for those threads to finish.

2. Critical Sections, Locks, Spinlocks (26 pts)

a) (8 pts) Consider the following attempt at solving the critical section problem for 2 threads *t0* and *t1*:

```
enum { T0, T1 } turn = T0; // turn is a global shared variable
```

`// thread T0 uses these`	`// thread T1 uses these`

```
// functions to enter/leave the
// critical section
void critical_section_enter_t0()
{
      while (turn != T0)
            continue;
}

void critical_section_leave_t0()
{
      turn = T1;
}
```

```
// functions

void critical_section_enter_t1()
{
      while (turn != T1)
            continue;
}

void critical_section_leave_t1()
{
      turn = T0;
}
```

i. (4 pts) Does this proposal provide mutual exclusion for the critical section? Justify your answer!

Yes, it does. The variable ‘turn’ is either T0 or T1, and if it is T0, then only thread T0 can progress past ‘critical_section_enter_t0’, and vice versa.

ii. (4 pts) Is this proposal a satisfactory solution to the critical section problem? Justify your answer!

No, it is not. It does not guarantee progress. In particular, thread T0 cannot reenter the critical section after it has left it unless thread T1 entered and left the critical section in the interim. Consider:

```
critical_section_enter_t0();
critical_section_leave_t0();
 // may block even though t1 is not in the critical section
 critical_section_enter_t0();
 critical_section_leave_t0();
```

Note: for a solution to be satisfactory, it has to provide the three criteria mutual exclusion, guaranteed progress, and bounded waiting. It is not required that it prevents starvation or deadlock (even correct solutions to the critical section problem, such as locks, may lead to deadlock when used incorrectly, and may lead to starvation when combined with strict priority scheduling.) It’s also wrong to argue that if one thread never gives up the critical section, the other thread won’t make progress, because this is true for all correct solutions to the critical section problem.
Some of you pointed out that the solution uses busy waiting. This certainly reduces its efficiency, which is a desirable property for a solution to the critical section problem. However, spinning is now always less efficient than blocking; otherwise, there would be no use for spinlocks. I gave 3 pts partial credit for this answer.

b) (10 pts) The Linux kernel, which is a multiprocessor kernel, provides the following function (actually implemented as a macro):

```
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
```

This function disables interrupts on the current processor, remembering the old interrupt state in the 'flags' variable, then acquires the spinlock 'lock.'

i. (4 pts) Why does Linux provide a function that combines *disabling interrupts* and *taking a spinlock*? Explain when a kernel developer would use this function!

To safely access data structures that are shared between an interrupt handler and a thread, a thread must exclude both threads running on other CPUs, other threads running on the same CPU, and it must delay the execution of interrupt handlers. Taking a spinlock excludes threads on other CPUs, and disabling interrupts excludes interrupt handlers on the same CPU. (Note that interrupt handlers on other CPUs are excluded because they must attempt to acquire the same spinlock.) This is such a common task that the kernel developers decided to provide a convenience function for it.

Some of you pointed out that when acquiring a spinlock, one must also disable preemption (or else a thread on the current CPU might end up uselessly spinning until its time slice expires, or may lock up the entire system if strict priority scheduling is used). It turns out that Linux disables (e.g., postpones) preemption-related context switches even in the "normal" spin_lock() function, without, however, disabling *all* interrupts as spin_lock_irqsave() does. Since I didn't expect you to know this detail, and since I didn't point it out in the question, I accepted this answer for full credit as well.

ii. (3 pts) The Linux kernel contains a debugging version of spinlocks which kernel developers can use to track down problems. This debugging version prints the message "BUG: spinlock lockup on CPU#n" if a spinlock cannot be acquired after about 1 second of spinning.
Explain what is meant by "spinlock lockup" and why this debugging message suggests that this is (likely) a "bug!"

Spinlocks must only be held for short periods of time. If a spinlock cannot be acquired for over 1 second, it means that the holder is holding it for an extended period of time. That would indicate a bug.

Some of you suggested that the lockup may be due to an attempt to recursively acquire the spinlock by the thread holding the spinlock. I accepted this answer as well, even though the Linux debugging code easily detects this case and flags such recursive attempts without actually spinning (since it knows it won't ever be able to acquire the lock). For full credit, you needed to have mentioned that spinlocks, by design, should only be held for short periods of time. Simply stating that the lock apparently wasn't released is not sufficient - if the lock were blocking, having to wait 1 second would not be an issue and in fact is not uncommon.

iii. (3 pts) What concrete mistake could have led to such a bug?

For instance, the thread holding the spinlock may have called a blocking function (such as timer_sleep()) that put the spinlock holding thread into the BLOCKED state.

c) (8 pts) Lock Metering. Lock Metering tools such as SGI's "lockstat" suite of tools allow to you collect statistics about Linux kernel locks. Among other information, they display how often attempts to acquire a lock were immediately successful vs. how often they required spinning (or led to blocking).

Suppose after using this tool you find that a particular lock is almost always held when a thread attempts to acquire it.

i. (4 pts) Explain how such contention can adversely impact the performance of a computer system!

If a thread attempting to acquire a lock fails to do so most of the time, then this thread must block; during this time, it cannot use the CPU. As a result, CPU utilization is low and CPU capacity may be wasted. Moreover, since blocking, unblocking, and scheduling the thread requires at least 2 context switches and 2 scheduler calls, additional overhead is being paid which reduces the amount of CPU time available for applications to perform useful work that is not related to synchronization.

For full credit, naming one adverse impact was sufficient.

ii. (4 pts) What approach would you propose to address this problem?

Several approaches are possible. First, examine if the lock is needed or if the data being protected can be partitioned or does not need to be shared. Second, if it does, consider using finer-grained locking. Examine whether the variables protected by this thread are always accessed together and if not, use different locks. Finally, make sure that the lock is held only for the time period in which it is absolutely necessary.

For full credit, naming a single approach was sufficient.
Some suggested that disabling interrupts should be used instead, which is not a good idea for two reasons: first, it would only work on a uniprocessor (which Linux isn't), and second, it would only work if the thread didn't block inside the critical section (which doesn't always hold true).

3. Semaphores (26 pts)

a) (18 pts) Consider the following “roller coaster” problem (taken from Greg Andrews’s book on *Concurrent Programming*):

Suppose there are n passenger threads and a car thread. The passengers repeatedly wait to take rides in the car, which can hold C passengers, where C < n. The car can go around the tracks only when it is full. The following rules apply:

• Passengers should invoke board() and unboard().
• The car should invoke load(), run() and unload().
• Passengers cannot call board() until the car has invoked load()
• The car cannot depart until C passengers have boarded.
• Passengers cannot unboard() until the car has invoked unload().

Your solution should use *only* semaphores for synchronization and (if needed) integer variables. Introduce variables as needed and show the code for the “car” thread and each “passenger” thread. You may use short-hand notation such as “semaphore s(0)” to declare a semaphore with an initial value of 0.

For brevity of notation, we assume that a semaphore’s down() and up() operations takes an integer argument K, and that sema_up(K) is equivalent to calling sema_up() K times and that calling sema_down(K) is equivalent to calling sema_down() K times.

Declarations:

```
semaphore mayBoard(0);
semaphore allAboard(0);
semaphore mayUnboard(0);
semaphore allAshore(0);
```

Car thread:

```
while (1) {
  load();
  mayBoard.up(C);    // signal C passengers that they may board
  allAboard.down(C); // wait until all passengers have boarded
  run();
  unload();
  mayUnboard.up(C);  // signal C passengers to unboard
  allAshore.down(C); // wait until all passengers have unboarded
}
```

Passenger thread:

```
while (1) {
  mayBoard.down();
```

```
        board();
        allAboard.up();
        // ride
        mayUnboard.down();
        unboard();
        allAshore.up();
      }
```

Common mistakes included not treating the semaphore as an abstract datatype and attempting to access its 'value' field directly. Semaphores provide only up and down operations, their value cannot be read.

b) (8 pts) In project 1, you implemented priority inheritance for mutexes, but not for semaphores. Would implementing priority inheritance for general, counting semaphores make sense? Justify your answer!

In general, it does not make sense. If a high-priority thread blocks while attempting to down a semaphore, we do not know which thread should inherit its priority, because we cannot easily predict which thread will signal ("up") this semaphore. Unlike locks, semaphores do not have a notion of ownership.

It makes sense only if the semaphore is used to represent a resource – in this case, a thread blocking on sema_down() could donate to the last thread that successfully downed the semaphore, which would need to be remembered. That would be, in essence, what you did in Project 1 for locks. We gave partial credit for this answer.

4. Condition Variables and Monitors (24 pts)

a) (9 pts) In project 0, your memory allocator's mem_alloc() function returned NULL if it could not satisfy an allocation request. In this question, you are asked to extend your allocator such that a thread that attempts to allocate memory when there is no sufficient memory is instead blocked until a free block of sufficient size becomes available. If a block of memory is freed, all allocation requests that could subsequently be satisfied should be satisfied without undue delay.

Your solution should use 1 condition variable (in addition to the pthread_mutex you are using to protect the free list). It should not be using busy-waiting.

```
pthread_cond_t alloccond;
pthread_mutex_t alloclock;

void *mem_alloc(size_t length)
{
    pthread_mutex_lock(&alloclock);
```

```
        while (!have_free_block_of_size(length))
            pthread_cond_wait(&alloccond, &alloclock);

        // allocate free block
        …
        pthread_mutex_unlock(&alloclock);
    }

    void mem_free(void *ptr)
    {
        pthread_mutex_lock(&alloclock);

        // put block on free list and coalesce
        …
        pthread_cond_broadcast(&alloccond);
        pthread_mutex_unlock(&alloclock);
    }
```

Note that the question asked that you use a condition variable, not reimplement one via thread_block/thread_unblock. Common mistakes included using 'if' instead of 'while' and using pthread_cond_signal instead of pthread_cond_broadcast. Your solution needed to show clearly where the pthread_cond_* operations are invoked.
Also, notice that unlocking and reacquiring the monitor lock is done inside pthread_cond_wait(), you must not code 'pthread_mutex_unlock(), pthread_cond_wait(), pthread_mutex_lock()'.

b) (3 pts) Discuss the efficiency of your solution, given the restriction that only 1 condition variable may be used.

To ensure that any blocked thread whose allocation request could be satisfied by a freed block can return from mem_alloc(), we have to use pthread_cond_broadcast() to wake up all waiting threads. This may lead to a "thundering herd" phenomenon in which many threads are being woken up, but only one (or a small number of them) can make progress before memory is again exhausted. More efficient solutions would require a targeted signaling which requires multiple condition variables.

c) (6 pts) Java and C# provide integrated support for monitor-style synchronization in which every Java or C# object can be used as both a mutex and a condition variable. By comparison, C API such as POSIX threads do not combine mutexes and condition variables; the programmer is responsible for using them in the style of a monitor. Compare these two approaches to each other! Discuss 1 advantage and 1 disadvantage of the Java/C# approach when compared to the C approach!

i. (3 pts) Advantage of Java/C#'s approach:

Two advantages include:

- The association between protected variables and monitors is clear. Private fields are protected by synchronized methods.
- The association between the monitor lock and the condition variable is clear. The runtime system can check that the correct lock is held when a condition variable is used (e.g., in Java, that the object on which wait() or notify() is invoked is the one used in the surrounding synchronized block – if not, Java throws an IllegalMonitorStateException)

ii. (3 pts) Disadvantage of Java/C#'s approach (Hint: consider the implementation of the Reader/Writer problem discussed in class!)

Two significant disadvantages include:

- The approach allows for only 1 condition variable per monitor. As such, problems that require 2 condition variables, such as the Reader/Writer problem discussed in class, require rather complex solutions.
- Either the overhead for all objects is increased, even for those that aren't used as monitors, or the runtime system is more complex, requiring sophisticated optimizations to efficiently support using only some objects as monitors.

I accepted other reasonable advantages and disadvantages for full or partial credit. Note, however, that the question asked for advantages of language-supported monitors vs. manually-arranged monitors as in C, so naming general advantages or disadvantages of monitors received less or no credit.

d) (6 pts) In a classic 1978 paper on "The Duality of Operating Systems Structures," Lauer and Needham discuss the concept of monitors and observe:

A process may WAIT anywhere within the ENTRY procedure or any other monitor procedure called by it, not just at the beginning as we have suggested in the canonical style. However, this is not without its disadvantages. […] In this sense, the WAIT statement is almost as ill-structured as the notorious 'go to' statement. Perhaps, it should be confined to, say, the entry and exit points of an ENTRY procedure for more clarity.

(In this quote, an "ENTRY" procedure is one that threads use to enter the monitor – such as a public synchronized method of a Java object. The term "processes" is used instead of "threads").

Why do Lauer and Needham recommend restricting the use of "WAIT" to the entry and exit points of a monitor procedure?

The full quote reads like so:

A process may WAIT anywhere within the ENTRY procedure or any other monitor procedure called by it, not just at the beginning as we have suggested in the canonical style. However, this is not without its disadvantages. **The procedure which WAITS must ensure that the monitor invariant is true, even**

though it might be deep inside an inner block of an inner procedure and may have captured all sorts of monitor information and temporary results in its local variables, results which could easily be invalidated by another process entering the monitor. In this sense, the WAIT statement is almost as ill-structured as the notorious 'go to' statement. Perhaps, it should be confined to, say, the entry and exit points of an ENTRY procedure for more clarity.

What Lauer and Needham point out is that whenever a thread calls WAIT(), the thread will leave the monitor, and when it reenters the monitor, other threads may have changed the monitor's state. We saw a taste of that in class when discussing the case of 2 consumers and 1 producer in the producer/consumer problem (and that was even while following L&N's "canonical style.")

Some of you claimed that calling WAIT, which releases the lock, does not guarantee mutual exclusion – that's not the case since the thread must reacquire the lock before reentering. Even when WAIT is used, only one thread can be in the monitor, and WAIT must release the lock by design to allow other threads to enter. Also, WAIT is a blocking call that gives up the CPU until a SIGNAL is done; no busy waiting is involved.