

FINAL EXAM

- This test contains 9 questions worth a total of 90 points
 - 1-3 5 points each
 - 4-7 10 points each
 - 8 15 points
 - 9 20 points
- You have 110 minutes to complete this midterm.
- You may **not** use your text, or any other reference material.
- Do not turn this page or turn over the exam until instructed to do so.

Hints

- Don't spend too much time on short answer questions. If you don't know the answer right away, move on and come back.
- There may be some questions that will require more time to think and organize your thoughts. If you find yourself stuck on one of these questions, save it until you have answered all the questions you can answer quickly. If you spend too much time on one question you may get frustrated and that will impair your ability to answer the other questions.
- Don't turn your test in early. If you have time left over, use it to review your answers. Students who turn tests in early often make trivial mistakes that they would catch if they reviewed their answers.
- Students that get the highest grades usually go over their answers several times.
- The space below a question **does not** indicate how long of an answer I am expecting. Sometimes I want to start the next question on a new page.
- Double check that you understand the question before you answer it. It is very likely you will not get any points if you don't understand the question. If you don't understand the question, ask me.

- 1) (5 points) What should be assumed when writing programs for a computer with preemptive scheduling? How can this assumption (or these assumptions) be dealt with when writing a program?

Processes (and kernel-level threads) can lose the CPU at anytime. Thus the programmer should assume that the process will be interrupted at any time. The programmer should protect all critical sections so that being interrupted cannot cause a race condition.

- 2) (5 points) When transferring a binary file over a network (e.g. ftp and our network file transfer assignment) there are two common methods used to determine the end of the file. Describe both of them. List one or more advantages of one or both of the methods.

One way is to send the size of the file first. Then the receiver can calculate which byte is the last one in the file.

Another way is to create a separate communication channel (usually using sockets) for transferring the file. When the channel is closed by the sender the receiver assumes the entire file has been sent.

An advantage of the second method is that it allows for a transfer to be easily canceled. The first mechanism requires that all the bytes be transferred. There is no way to send a "cancel" message. On the other hand, the second method can terminate the transfer by closing the connection.

An advantage of the first method is that it is easier to implement.

- 3) (5 points) What would be two important advantages to using your threadpool to implement the multi-threaded file search program? Explain why the given advantages are important.

It would have been much easier to implement.

It would run more efficiently because time would not be spent creating and destroying threads.

- 4) (10 points) Explain the run time stack. What causes the run time stack to overflow? A diagram might help.

When a function is called, information about the function is placed into a structure called an activation record. The run time stack is a stack of activation records. When a function is called its activation record is pushed onto the run time stack. When the function terminates its activation record is popped off of the run time stack. If enough functions are called then the memory allocated for the stack is used up. This error is called stack overflow. The following program always causes stack overflow.

```
void f()
{
    f();
}
```

- 5) (10 points) Describe how the threads in your threadpool assignment were dispatched and how they were blocked when they were not busy.

A mutex lock is dedicated to each thread. When a thread is starting (or done executing the last job) it waits on its own mutex lock. The dispatch thread puts the function to be executed and the arguments in an array indexed by thread number. It then unblocks the correct thread.

For example, thread 42 will block on mutex lock 42. When the dispatcher determines that thread 42 is the next one to be dispatched, it will put the function pointer into slot 42 of the array of function pointers and the arguments into slot 42 of the argument pointers. Then it will unlock mutex lock 42. When thread 42 wakes up it will execute the function in slot 42 using the arguments in slot 42.

- 6) (10 points) Outline how UNIX pipes, fork(), and exec() are used to implement pipes in the UNIX shell. For example, how `$ ls | grep | sort` would be handled by the shell.

The shell creates a pipe for each neighboring command to communicate. In the above example, one pipe for `ls` and `grep` to communicate (call it pipe₁) and one pipe for `grep` and `sort` to communicate (pipe₂). Then it forks a process to run each command. Each child process redefines its input and output to the appropriate pipe and then calls `exec()` on the target command. In this example, `grep` redefines its input so it is reading from pipe₁ and its output so it is writing to pipe₂.

7) (10 points) Write pseudo code for a function that is not thread safe (write just enough to demonstrate the lack of thread safety). Explain why it is not thread safe. How could you fix this function so it would be thread safe?

```
struct tm *localtime(time_t time)
{
    static struct tm my_tm; // the "static" makes it a persistent variable (like a global variable)
    my_tm.tm_sec = get_seconds(time);
    my_tm.tm_min = get_min(time);
    ...
    return &my_tm;
}
```

If multiple threads were to call this function they would overwrite `my_tm`. For example, `thread1` calls this function and is in the middle of using the information in `my_tm` when `thread2` calls this function and overwrites the data in `my_tm`. Now when `thread1` accesses the structure it contains the wrong data – the data put in by `thread2`'s call.

This function could be made thread safe by making `my_tm` an argument. If `my_tm` was an argument each thread would use different memory making the function thread safe.

8) (15 points) Describe how the bounded buffer problem can be implemented using different mechanisms for sharing data. In other words, for each data sharing mechanism you can think of, **briefly outline** how the mechanism can be used to solve the bounded buffer problem (don't write code or pseudocode, it would take too long). Include a mention of how synchronization can be handled when using each method. State if this method works for threads, processes, or both. The students who list more mechanisms will get more points than the students who list fewer mechanisms. Don't limit yourself to the case where both producer and consumer are on one machine.

Pipes: the producer would write to the pipe and the consumer would read from the pipe. Synchronization is handled by the pipe. Works for both threads & processes.

Sockets: a socket connection would be set up between the producer and consumer. The producer would write to the socket on its end and the consumer would read from the socket on its end. Synchronization is handled by the socket. Works for both.

Unix Shared Memory: An array in shared memory can be used as a buffer. The producer would write into the array when it had an empty space (would wait otherwise) and the consumer would read when one space was not empty. Semaphores could be used to control access to the indexes used for the full/empty spaces and the number of elements currently in the array. Works for both.

Thread Shared Memory: Just like the above solution but mutex locks and conditional variables can be used to control access. Works only for threads.

Message Queue: Works just like the socket and pipe solutions. Synchronization is handled by the message queue. Works for both threads & processes.

9) (20 points) Write code to simulate a tennis club. In order to play tennis a tennis player needs someone to play against and a tennis court. Write pseudo code that the threads simulating the tennis players would run and code that a single court scheduling thread would run. Assume there are m tennis courts and n tennis players. Not all the tennis players will want to play at the same time. The court scheduling thread should make sure players get to play in a first-come-first-served order. List global variables and their initial values. Use conditional variables for synchronization.

```

bool court_busy[m];
int court_assignments[n];
int partner_assignments[n];
pthread_mutex_t player_mutex_locks[n]; // init to locked
queue<int> player_queue; // players waiting to play
queue<int> free_courts; // unused courts
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t waiting_players = PTHREAD_COND_INITIALIZER;

void *player_code(void *arg)
{
    int id = (int) arg;
    while (true)
    {
        // tell the club that I want to play
        pthread_mutex_lock(&mutex);
        player_queue.push(id);
        pthread_mutex_unlock(&mutex);

        // wait for the court scheduler to wake me up
        pthread_mutex_lock(&player_mutex_locks[id]);

        // play tennis
        cout << "START: player " << id << " playing with player "
             << partner_assignments[id] << " on court "
             << court_assignments[id] << endl;

        // play tennis: shortcoming: When one player quits, the other keeps playing
        sleep(rand() % 5);

        pthread_mutex_lock(&mutex);
        court_assignments[id] = -1; // flag that I'm done

        // if my partner is done, the court is free
        if (court_assignments[partner_assignments[id]] == -1)
            free_courts.push(court_assignments[id]);
        pthread_cond_signal(&waiting_players);

        cout << "END: player " << id << " playing with player "
             << partner_assignments[id] << " on court "
             << court_assignments[id] << endl;

        pthread_mutex_unlock(&mutex);

        // do something else -- remainder section
        sleep(rand() % 10);
    }
}

```

```

void *scheduler_code(void *arg)
{
    for (int rounds = 0; rounds < 100; rounds++)
    {
        // wait for two players to be in the queue
        pthread_mutex_lock(&mutex);
        while (player_queue.size() < 2 || free_courts.size() == 0)
            pthread_cond_wait(&waiting_players, &mutex);

        int player1 = player_queue.front();
        player_queue.pop();
        int player2 = player_queue.front();
        player_queue.pop();
        int court = free_courts.front();
        free_courts.pop();

        court_assignments[player1] = court;
        court_assignments[player2] = court;
        partner_assignments[player1] = player2;;
        partner_assignments[player2] = player1;;
        pthread_mutex_unlock(&player_mutex_locks[player1]);
        pthread_mutex_unlock(&player_mutex_locks[player2]);

        pthread_mutex_unlock(&mutex);
    }
}

int main()
{
    // put all the courts on the free_courts queue
    for (int i = 0; i < m; i++)
        free_courts.push(i);

    pthread_t player_tids[n];
    pthread_t scheduler_tid;

    // initialize all of the player's locks and lock them
    // create all the player's threads
    for (int i = 0; i < n; i++)
    {
        pthread_mutex_init(&player_mutex_locks[i],0);
        pthread_mutex_lock(&player_mutex_locks[i]);
        pthread_create(&player_tids[i], NULL, player_code, (void *) i);
    }
    pthread_create(&scheduler_tid, NULL, scheduler_code, 0);

    pthread_join(scheduler_tid, NULL);
    return 0;
}

```