
Rules:

Duration: *3 hours*

Hint: *Look at the point spread - attack with points in mind*

Permitted: *Open notes/reference material (including online)*

Problem	***Score***	***Total***
1		*5*
2		*5*
3		*5*
4		*5*
5		*5*
6		*5*
7		*20*
8		*10*
		50

NOTE: *READ and SIGN the next page*

Make sure you have all 10 pages of the exam

- University's Academic Integrity Policy is in effect during this exam.
    - http://academicintegrity.depaul.edu/AcademicIntegrityPolicy.pdf

    3. Violations of Academic Integrity

    Violations of academic integrity include, but are not limited to, the following categories:

    3.1. Cheating

    Any action that violates University norms or instructor guidelines for the preparation and submission of assignments. This includes, but is not limited to:

    - *Copying from another student.*
    - *Offering, accepting, or otherwise obtaining or facilitating unauthorized assistance from or for another student.*
    - *Having someone take an exam or complete an assignment in one's place.*
    - *Unauthorized accessing of exam materials.*
        - *Accessing, using or possessing unauthorized materials during exams or quizzes.*

Please sign that you understand rules and university guidelines:

Name:
Date:
Num of total pages:

Problem 1: (10pts) Deadlocks

[True/False] **Which of the following are guidelines for avoiding deadlocks?**

a) Create a lock hierarchy so that locks protecting low-level data are prioritized over locks protecting high-level data.

b) Use std::recursive_mutex to allow multiple locks using the same mutex. Using a recursive mutex resolves the undefined behavior that occurs when attempting to acquire a lock on a std::mutex which has already been used for a lock. When using a recursive mutex, the user just needs to ensure all locks on the mutex are released before the thread terminates.

c) While already in possession of a lock, avoid calling user-supplied code since the user-supplied code's actions are unknown and may attempt to acquire a lock.

d) Avoid nested locks, which can be avoided by not acquiring a lock if you already possess one, and if multiple locks are required, do it in a single action using std::lock.

e) Call lock and unlock directly on the mutex, and avoid using std::lock and std::lock_guard except for the most trivial code.

f) Always lock two mutexes in the same order (such as A before B), and when the mutexes are protecting a separate instance of the same class, pass both mutexes into the constructor of the lock and use std::lock_guard with the std::adopt_lock argument.

Answer: fill in ***TRUE/FALSE*** below *(Are guidelines – True, Are NOT guidelines - False)*

A)

B)

C)

D)

E)

F)

Problem 2: (5pts) Condition_Variable

[True/False] **Which statements are correct concerning wait() and notify()?**

a) The lock used by wait() needs to be locked before wait() is called because it will be automatically unlocked by wait().

b) A thread blocked by wait_for() will only be unblocked when the condition variable is notified or the relative timeout duration expires.

c) All threads that call wait(std::unique_lock& *lock*) on the same condition variable need to acquire the *lock* on the same mutex.

d) When a thread calls wait(std::unique_lock& *lock*, Predicate *stop_waiting*), the thread will be at least blocked once.

e) When using the predicate version of wait_for(), it only returns false if the predicate function still evaluates to false after the timeout expired.

f) If there are more than one threads waiting for the condition when notify_one() is called, it is unspecified which of the threads will be unblocked.

g) notify_one() can unblock a thread that started waiting just after the call to notify_one() was made.

h) The notifying thread needs to hold the lock on the same mutex as the one held by the waiting thread(s).

Answer: fill in as ***TRUE/FALSE*** below: *(correct – True, incorrect – False)*

A)

B)

C)

D)

E)

F)

G)

H)

Problem 3: (5pts) Non-Safe code

[short answer] **What is the non-thread safe aspect of the method "GetNextCount"?**

```
#include <mutex>

class DoStuff
{
public:
    int &GetNextCount()
    {
        std::lock_guard<std::mutex> lk(mtx);
        count++;
        return count;
    }

private:
    int count;
    std::mutex mtx;
 };
```

Answer: (short answer)

Problem 4: (5pts) Future/Promise

[True/False] **Which of the following are appropriate use of a promise and future?**

a) Thread A gets an unknown number of orders from a client, thread B waits until A is told it has received all orders and sends its orders to B.

b) Thread A stores an integer that counts upwards one at a time at an arbitrary rate, and every time this integer reaches a multiple of 1000, it sends a signal to thread B, causing it to wake up.

c) Thread A and thread B share an integer that may only be modified in one thread at a time. This value may be changed multiple times.

d) Thread A and thread B run side by side simultaneously. When thread A reaches a certain result, both threads must end.

e) Thread A stores an integer that counts up an arbitrary amount every time it loops. When this integer lands on a clean multiple of 10 for the first time, thread B will begin.

f) Thread A and thread B both end with a result that the main thread must read. Main only needs to be able to read these values once thread A and B end.

Answer: fill in as ***TRUE/FALSE*** below: *(appropriate – True, inappropriate – False)*

A)

B)

C)

D)

E)

F)

Problem 5: (5pts) Join/Detach()

[True/False] **Which statements are correct concerning join() and detatch()?**

a) You do not need to call join() or detach() to safely exit a thread.

b) The thread of execution will wait for a thread object to finish when calling join().

c) The thread of execution will wait for a thread object to finish when calling detach().

d) Detach() allows the thread to operate independently from the thread that created it.

e) A thread is still joinable when calling detach().

f) One should check that a thread is joinable before calling detach().

Answer: fill in as ***TRUE/FALSE*** below: *(correct – True, incorrect – False)*

A)

B)

C)

D)

E)

F)

Problem 6: (5pts) Launching a Thread

[short answer] **Which of the following cannot launch the thread correctly?**

There is a class with a function call operator:

```
class background_task
{
public:
    void operator()() const
    {
        do_something();
        do_something_else();
    }
};
```

Scenarios:

a)
```
background_task f;
std::thread my_thread(f);
```

b)
```
std::thread my_thread(background_task());
```

c)
```
std::thread my_thread{background_task()};
```

d)
```
std::thread my_thread((background_task()));
```

e)
```
std::thread my_thread([]{
                      do_something();
                      do_something_else();
                      });
```

Answer: (list all Letters(a-e) that **cannot** launch the thread ***with*** 1-3 sentence justification)

Problem 7: (20 pts) Thought Experiment

Write an application to read 3 different binary files.

- a.bin
- b.bin
- c.bin

Concatenate these files into one final file, out.bin

At windows DOS command prompt you would type:

```
copy /b a.bin+b.bin+c.bin out.bin
```

Assume:

- these files are in different directories, requiring seek times and the files are fragmented
  - so it takes a long time to load from file to memory
- that fread(...) can load a file from disc to memory
  - It can access the files and stream the data into memory in parallel
  - Some magical DMA is installed that allows this to happen
  - this is a theoretical problem

Instead of using DOS command you write a method:

```
void Concatenate( char *outFileName, char *AFileName,
        char *BFileName, char *CFileName );
```

for the above example:

```
Concatenate( "out.bin", "a.bin", "b.bin", "c.bin");
```

How would you write this application for concurrency?

[*pseudo code / sketch* – sample code doesn't need to compile]

Problem 8: (10 pts) Reflection

You came to class with some Multithreaded views(beliefs), describe how those ideas were reinforced by our multithreading assignments (Jetsons and Maze) or changed.