# New York University
# CSCI-UA.0202: Operating Systems (Undergrad): Fall 2021
# Midterm Exam

- This exam is **75 minutes**. Stop writing when "time" is called. *You must turn in your exam; we will not collect it.* Do not get up or pack up in the final ten minutes. The instructor will leave the room 78 minutes after the exam begins and will not accept exams outside the room.
- There are **11** problems in this booklet. Many can be answered quickly. Some may be harder than others, and some earn more points than others. You may want to skim all questions before starting.
- **This exam is closed book and notes. You may not use electronics: phones, tablets, calculators, laptops, etc.** You may refer to ONE two-sided 8.5x11" sheet with 10 point or larger Times New Roman font, 1 inch or larger margins, and a maximum of 55 lines per side.
- Do not waste time on arithmetic. Write answers in powers of 2 if necessary.
- If you find a question unclear or ambiguous, be sure to write any assumptions you make.
- Follow the instructions: if they ask you to justify something, explain your reasoning and any important assumptions. **Write brief, precise answers. Rambling brain dumps will not work and will waste time.** Think before you start writing so that you can answer crisply. Be neat. If we can't understand your answer, we can't give you credit!
- If the questions impose a sentence limit, we will not read past that limit. In addition, *a response that includes the correct answer, along with irrelevant or incorrect content, will lose points.*
- Don't linger. If you know the answer, give it, and move on.
- **Write your name and NetId on this cover sheet and on the bottom of every page of the exam.**

*Do not write in the boxes below.*

| I (xx/9) | II (xx/10) | III (xx/14) | IV (xx/15) | V (xx/30) | VI (xx/10) | VII (xx/12) | Total (xx/100) |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

**Name:** Solutions

**NetId:**

# I Basics (9 points)

**1. [3 points] How many possible settings are there for a bit?**

2. bit means “binary digit”.

**2. [6 points]** Consider the following C program; read it carefully:

```
#include <stdio.h>

int b = 4;  /* global variable */

void foo(int a) {
  a = 5;
}

void bar(int* a) {
  a = &b;
}

void baz(int* a) {
  *a = 5;
}

int main() {

  int a1 = 1, a2 = 1, a3 = 1;

  foo(a1);

  bar(&a2);

  baz(&a3);

  printf("%d, %d, %d", a1, a2, a3);

  return 0;

}
```

**What does this program print?**

1, 1, 5.

# II Stacks and assembly (10 points)

**3. [10 points]** Below is some C code and the corresponding unoptimized assembly code.

```
100  int foo(int a) {
101      int b = a + 1;
102      return b;
103  }
104
105  int main() {
106      int a = 1;
107      int b = foo(a);
108  }
```

```
200   foo:
201        pushq   %rbp
202        movq    %rsp, %rbp
203        movq    %rdi, -20(%rbp)
204        movq    -20(%rbp), %rax
205        addq    $1, %rax
206        movq    %rax, -4(%rbp)
207        movq    -4(%rbp), %rax
208        popq    %rbp
209        ret
210
211   main:
212        pushq   %rbp
213        movq    %rsp, %rbp
214        subq    $16, %rsp
215        movq    $1, -8(%rbp)
216        movq    -8(%rbp), %rax
217        movq    %rax, %rdi
218        call    foo
219        movq    %rax, -4(%rbp)
220        movq    $0, %rax
221        ret
```

Below are the meanings of some assembly instructions. This is mostly taken from the class notes.

```
movq A, B     [ move 64-bit data from A to B ]

pushq %rax    [ subq $8, %rsp
                movq %rax, (%rsp) ]

popq %rax     [ movq (%rsp), %rax
                addq $8, %rsp     ]

call 0x5000    [ pushq %rip
                 movq $0x5000, %rip ]

ret           [ popq %rip ]
```

**Name:** Solutions **NYU NetId:**

Which assembly instruction will be executed after line 209 (`ret`) executes?

**Write down the line number, for example 200.**

line 219

When the process finishes executing line 201 (the first instruction of function `foo`), the top element on the stack is the contents of register `%rbp`.

**What is the element below that one on the stack, meaning the element that would be returned by a second stack pop?**

line 219.

What are the assembly instructions that represent line 106 (in the C code)?

**Write down the line number or numbers, for example 200 or 200–222.**

lines 215–216.

During a function call, the caller needs to save call-clobbered (also known as caller-saved) registers. Where is the saving of call-clobbered registers happening in `main()` when calling `foo()`?

**Write down the line number or numbers, for example 200 or 200–222. If there is no such saving in this example, write "none."**

None.

# III Concurrent programming (14 points)

**4. [10 points]** This question asks about coding patterns with multithreaded programming. We have two helper functions or methods:

- `bool safe_to_proceed()` is a function or method that returns a Boolean variable, which is true if it's safe to proceed, and false otherwise.
- `bool progress_is_possible()` is a function or method that returns a Boolean variable, which is true if other threads can (according to the current memory state) possibly make progress, and false otherwise. *This function does not mutate (change) memory state.*

We have a shared mutex `m` and a condition variable `cv`. Assume that each of the patterns below is enclosed in the mutex's acquire/release critical section.

Which of the following patterns abides by the cs202 concurrency commandments and common notions of program correctness? (Hint: the answer is not "None"; you will need to circle at least one of the items.)

**Circle ALL that apply:**

**a**
```
while (safe_to_proceed())
    cv.wait(&m);
```

**b**
```
while (! safe_to_proceed())
    cv.wait(&m);
```

**c**
```
if (! safe_to_proceed())
    cv.wait(&m);
```

**d**
```
while (progress_is_possible())
    cv.signal(&m);
```

**e**
```
while (! progress_is_possible())
    cv.signal(&m);
```

**f**
```
if (progress_is_possible())
    cv.signal(&m);
```

b and f.
a doesn't make logical sense; it's a garbled version of b.
c is a sin.
d and e could lead to infinite loops because the checked condition is either going to be true forever or not, because we aren't releasing the mutex, and nothing in the execution path of the program changes program state.

**5. [4 points]** You write some multithreaded code. You follow the CS202 concurrency commandments, using one or more monitors.

**Is deadlock possible? If so, give a concise example, or refer to one. If not, say why not. Do not write more than two sentences.**

Yes, deadlock is possible. We saw an example in class: the M monitor includes the N monitor.

# IV ls lab (15 points)

**6. [15 points]** The function `file_tree_count()`, listed below, starts at a directory (specified by `dirname`) and is supposed to count *all* files in the directory and, recursively, all of the files in that directory's subdirectories (and in their subdirectories, etc.). `file_tree_count()` uses system calls and logic similar to what you used in lab 2. However, it has three bugs, as indicated.

```
int file_tree_count(char* dirname)
{
        int count = 0;
        struct dirent* entry;
        struct stat st;
        char pathandname[512];

        DIR* dir = opendir(dirname);

        // **BUG 1**: What is the missing check between opendir and readdir?

        while ((entry = readdir(dir)) != NULL) {

                // **BUG 2**: A missing check. What kind of entries must we skip?

                snprintf(pathandname, sizeof(pathandname),
                         "%s/%s", dirname, entry->d_name);

                if (stat(pathandname, &st) == -1) {
                        continue;
                }

                if (S_ISDIR(st.st_mode)) {
                        // **BUG 3**: What is the bug in the following line?
                        count += file_tree_count(entry->d_name);
                } else {
                        count++;
                }
        }

        if (dir) closedir(dir);
        return count;
}
```

**Below, say what the three errors are. You need not propose the correct version of the code.**

BUG 1:

BUG 2:

BUG 3:

BUG 1: check whether dir is NULL.
BUG 2: skip the pseudodirectories "." and ".." (some students said to skip all files beginning with ".", but that is not correct; the problem specified that all files should be listed).
BUG 3: need to pass the full path, not just the filename piece, to file_tree_count().

# V EStore lab (30 points)

**7. [30 points]** In lab 3, you filled out an `EStore` class with an inventory driven by multiple threads executing handler functions. In this question, you will add support for two more `EStore` operations and the corresponding handler functions:

- `EStore::removeStock()`: this method takes an item denoted by `item_id` and `quantity` and removes that much from inventory. If there aren't enough of the item to remove, then wait until there are enough.
- `EStore::swapStock()`: this method atomically adds stock to one item with parameters (`add_id`, `add_quantity`) and removes stock from another item with parameters `(remove_id, remove_quantity)`. As above, if there aren't enough of `remove_id` to remove, then wait until there are enough. When adding stock, wake up any threads waiting to remove that item.

Notes:

- The lab has a notion of validity that is not operative here: you can assume that all items are valid.
- You should treat `EStore` as a monitor, meaning use a single mutex that is provided in the `EStore` definition below.
- You do not have to worry about the interaction between the methods that you write here and the ones that you implemented for lab 3, as long as the two operations introduced in this question are properly synchronized with each other.

**Complete the (a) request structs, (b) handlers, (c) EStore members, and (d) EStore methods.**

```
// **** The following two structs belong in Request.h
struct RemoveStockReq
{
    EStore* store;
    // FILL IN BELOW




};

struct SwapStockReq
{
    EStore* store;
    // FILL IN BELOW




};
```

```
// **** The following two handlers should be in RequestHandler.cpp
// * You need not print any handling messages.
// * Remember to delete your request.
void remove_stock_handler(void *args) {
    // FILL THIS IN









}

void swap_stock_handler(void *args) {
    // FILL THIS IN









}



// Possibly helpful definition, from EStore.h

class Item {
    public:
      bool valid;     // assume always true
      int quantity;
      double price;
      double discount;

    Item();
    ~Item();
};
```

```
// The following is a snippet of class EStore
class EStore {

  private:
    Item inventory[INVENTORY_SIZE];
    smutex_t mutex;
    // ADD MORE HERE



  public:
    // ...
    void removeStock(int item_id, int quantity);
    void swapStock(int add_id, int add_quantity,
                   int remove_id, int remove_quantity);
    // ...
};

void EStore::removeStock(int item_id, int quantity) {
    // FILL THIS IN









}

void EStore::swapStock(int add_id, int add_quantity, int remove_id, int remove_quantity) {
    // FILL THIS IN









}
```

```
struct RemoveStockReq
{
    EStore* store;
    // FILL IN BELOW

    int item_id;
    int quantity;
};

struct SwapStockReq
{
    EStore* store;
    // FILL IN BELOW

    int add_id;
    int add_quantity;
    int remove_id;
    int remove_quantity;

};
// **** The following two handlers should be in RequestHandler.cpp
// * You need not print any handling messages.
// * Remember to delete your request.
void remove_stock_handler(void *args) {
    // FILL THIS IN
    struct RemoveStockReq* req = (struct RemoveStockReq*) args;
    req->store->removeStock(req->item_id, req->quantity);
    delete req;
}

void swap_stock_handler(void *args) {
    // FILL THIS IN
    struct SwapStockReq* req = (struct SwapStockReq*) args;
    req->store->swapStock(req->add_id, req->add_quantity,
                          req->remove_id, req->remove_quantity);
    delete req;
}



// Possibly helpful definition, from EStore.h

class Item {
    public:
      bool valid;     // assume always true
      int quantity;
      double price;
      double discount;

    Item();
```

```
    ~Item();
};



// The following is a snippet of class EStore
class EStore {

  private:
    Item inventory[INVENTORY_SIZE];
    smutex_t mutex;
    // ADD MORE HERE
    scond_t cond;


  public:
    // ...
    void removeStock(int item_id, int quantity);
    void swapStock(int add_id, int add_quantity,
                   int remove_id, int remove_quantity);
    // ...
};

void EStore::removeStock(int item_id, int quantity) {
    // FILL THIS IN
    smutex_lock(&mutex);

    while (inventory[item_id].quantity < quantity)
        scond_wait(&cond, &mutex);

    inventory[item_id].quantity -= quantity;

    smutex_unlock(&mutex);

}

void EStore::swapStock(int add_id, int add_quantity, int remove_id, int remove_quantity) {
    // FILL THIS IN

    smutex_lock(&mutex);

    while (inventory[remove_id].quantity < remove_quantity)
        scond_wait(&cond, &mutex);

    inventory[remove_id].quantity -= remove_quantity;
    inventory[add_id].quantity += add_quantity;

    scond_broadcast(&cond, &mutex);

    smutex_unlock(&mutex);
}
```

In SwapStock(), many students added items to the stock and then did while() wait();. But wait() releases the mutex (before acquiring it again), so this pattern is not atomic, contrary to the specification in the question. To atomically add and remove, both the add and remove have to happen without releasing the mutex.

*Additional space if needed*

# VI Scheduling (10 points)

**8. [10 points]** Consider the setup below, which follows an example from class. Time is divided into epochs. Jobs arrive at the very beginning of the epoch listed under "arrival epoch"; after they have been given the CPU for "burst length" epochs, they leave. Processes do no I/O.

| process | arrival epoch | burst length |
|---|---|---|
| P1 | 0 | 5 |
| P2 | 2 | 4 |
| P3 | 3 | 1 |
| P4 | 4 | 1 |

Assume that the scheduling algorithm is STCF (Shortest Time to Completion First, a preemptive algorithm). Of course, in general, STCF cannot be implemented because an algorithm cannot know in advance how long each process will run, but here we are given that information.

**Write down the process scheduled for each epoch beneath the epoch number (as we did in class):**

```
0   1   2   3   4   5   6   7   8   9   10

P1  P1  P1  P3  P4  P1  P1  P2  P2  P2  P2
```

**What is the average turnaround time?**

$7+9+1+1 = 18/4 = 4.5$

# VII Virtual memory and feedback (12 points)

**9. [5 points]** Assume an architecture with a 4-level page table, like the x86-64. But unlike the x86-64, here there are 44 bits used for addressing, broken down as follows:

```
[7 bits | 9 bits | 8 bits | 10 bits | 10 bits ]
```

The top 7 bits index into the L1 page table, the next 9 bits index into a L2 page table, and so forth. The bottom 10 bits determine the offset into the physical page. (Side note: this architecture is not a good idea, but that's not relevant to the question.)

**What is the maximum number of L2 page tables that can be allocated for a process on this architecture?**

$2^7 = 128$. There is one L1 page table. It has 128 entries. Each of these entries can point to another L2 page table.

**10. [5 points] Describe a case where a memory reference on a machine with a hardware-managed TLB (like the x86-64) results in a TLB miss but not a page fault. Do not write more than two sentences.**

The process references virtual memory that is mapped (according to the page tables) but hasn't been referenced in a while, so there's no TLB entry for it. In this case, the hardware walks the page tables, and installs the correct mapping in the TLB. Alternatively: A process is context-switched in, so its TLB is cold, but the page tables are present and marked valid.

**11. [2 points]** This is to gather feedback. Any answer, except a blank one, will get full credit.

**Please state the topic or topics in this class that have been least clear to you.**

**Please state the topic or topics in this class that have been most clear to you.**

*Scratch space if needed*

*Scratch space if needed*

# End of Midterm