New York University
CSCI-UA.202: Operating Systems (Undergrad): Spring 2020
Midterm Exam

- This exam is **75 minutes**.
- The answer sheet is available here:
  - **Section 001 (Aurojit Panda)**
  - **Section 003 (Michael Walfish)**

  The answer sheet has the hand-in instructions and the video upload instructions.
- There are **14** 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.
- If you have questions about the exam please go to https://campuswire.com/p/G7207B15D (if you need access, use code 3417).
- **Write your name and NetId on the document in which you are working the exam.**

Do not write in the boxes below.

I (xx/12)	II (xx/15)	III (xx/38)	IV (xx/20)	V (xx/15)	Total (xx/100)

I Fundamentals (12 points)

1. [4 points] "A _____is an instance of a running program."

Fill in the blank above with the name of the right OS abstraction.

Process.

2. [4 points] What tool takes as input a text file with program code and produces an executable?

Give your answer below, which should be a maximum of two words.

Compiler.

3. [4 points] 1 GB (gigabyte) is equal to $2^{\square}$ bytes.

What number belongs in the blank exponent above?

30.

II Programming errors, `ls` lab (15 points)

4. [6 points] What is the programming error in the code below? State the line number(s) and the problem. Don't use more than two sentences.

Line: 11, 13. The programmer returned a pointer to a local variable.

```
1   // A box. Each box has an ID and a pointer to the box that resides inside of it.
2   // If the box has nothing inside of it, inner_box should be equal to NULL.
3   struct box {
4       int id;
5       struct box *inner_box;
6   };
7
8   // Insert box: places the box "inner" inside of the box "outer".
9   // Since "outer" is being modified, we pass a pointer to "outer".
10  // Since "inner" is not being modified, we pass in "inner" directly.
11  void insert_box(struct box* outer, struct box inner) {
12      printf("insert box: placing id %d inside id %d\n", inner.id, outer->id);
13      outer->inner_box = &inner;
14  }
15
16  // Print box: prints a box and the box inside of it. This function
17  // is recursive and will end once a box is empty.
18  void print_box(struct box* first, int level) {
19      int i;
20      if (!first)
21          return;
22
23      for (i=0; i < level; ++i) {
24          printf("- ");
25      }
26      printf("id: %d\n", first->id);
27      print_box(first->inner_box, level+1);
28  }
29
30  int main() {
31      // Create three boxes.
32      struct box box1 = { .id = 37, .inner_box = NULL };
33      struct box box2 = { .id = 12, .inner_box = NULL };
34      struct box box3 = { .id = 19, .inner_box = NULL };
35
36      // The box ordering should be box1 -> box2 -> box3
37      insert_box(&box1, box2);
38      insert_box(&box2, box3);
39
40      print_box(&box1, 0);  // print the boxes starting from the outside box.
41
42      return 0;
43  }
```

5. [9 points] On the next page is an excerpt of an `ls` implementation that contains a memory or resource leak. Notice that this particular `ls` implementation is not trying to handle recursive listing—so the absence of recursive listing is not an issue.

What is the leak? State the line number(s) where the resource is allocated. Don't use more than one sentence.

The code never calls `closedir` on `dirp`, which is opened on line 9.

State the fix for the leak below. Use syntactically correct C.

Add `closedir(dirp)` after the while loop, between lines 23 and 24.

```
1  /* list_dir(): implement the logic for listing a directory in the
2   *     case where recursive listing is not desired.
3   * This function takes:
4   *    - dirname: the name of the directory
5   *    - list_long: should the directory be listed in long mode?
6   *    - list_all: are we in "-a" mode?
7   */
8  void list_dir(char* dirname, bool list_long, bool list_all) {
9      dir *dirp = opendir(dirname);
10     struct dirent *entry;
11     if (!dirp) {
12         handle_error("Could not open directory", dirname);
13         return;
14     }
15
16     while ((entry = readdir(dirp)) != NULL) {
17         char *file_name = entry->d_name;
18         char full_name[2048];
19         if (list_all || file_name[0] != '.') {
20             snprintf(full_name, 2048, "%s/%s", dirname, file_name);
21             list_file(full_name, file_name, list_long);
22         }
23     }
24 }
```

For reference, here are relevant function signatures, taken from the template code that we distributed:

```
1  /* list_file(): implement the logic for listing a single file.
2   * This function takes:
3   *   - pathandname: the directory name plus the file name.
4   *   - name: just the name "component".
5   *   - list_long: a flag indicated whether the printout should be in
6   *   long mode.
7   *
8   *   The reason for this signature is convenience: some of the file-outputting
9   *   logic requires the full pathandname (specifically, testing for a directory
10  *   so you can print a '/' and outputting in long mode), and some of it
11  *   requires only the 'name' part. So we pass in both. An alternative
12  *   implementation would pass in pathandname and parse out 'name'.
13  */
14 void list_file(char* pathandname, char* name, bool list_long);
15
16 /*
17  * call this when there's been an error.
18  * The function:
19  * - prints a suitable error message (this is already implemented)
20  * - sets appropriate bits in err_code
21  */
22 void handle_error(char* what_happened, char* fullname);
```

III Concurrency (38 points)

6. [9 points] For this problem, assume sequential consistency (memory operations happen in program order). Recall the *critical section problem*: a solution must provide mutual exclusion, progress, and bounded waiting. Mutexes are a solution to this problem. An alternative solution is Peterson's algorithm. (We alluded to this solution but did not cover it.) Below is Peterson's solution, adapted to a lock-like API, and assuming two threads. It is correct (assuming sequential consistency): it provides mutual exclusion, progress, and bounded waiting.

```
/* Set all fields to zero on initialization */
struct peterson {
  int wants[2];   /* wants[i] = 1 means thread i wants or has lock */
  int not_turn;   /* not this thread's turn if other one wants lock */
};

/* i is thread number, which must be 0 or 1 */
void peterson_lock (struct peterson *p, int i)
{
  p->wants[i] = 1;
  p->not_turn = i;
  while (p->wants[1-i] && p->not_turn == i) ;
}

/* i is thread number, which must be 0 or 1 */
void peterson_unlock (struct peterson *p, int i)
{
  p->wants[i] = 0;
}
```

Using a global "`struct peterson globallock;`" two threads can synchronize with:

```
while (1) {
  peterson_lock (&globallock, i);
  Critical_section_for_thread_i ();
  peterson_unlock (&globallock, i);
  Other_stuff_for_thread_i ();
}
```

But is `not_turn` actually required? Consider an alternative implementation of `peterson_lock()`:

```
/* i is thread number, which must be 0 or 1 */
void peterson_lock_alt (struct peterson *p, int i)
{
  p->wants[i] = 1;
  while (p->wants[1-i]) ;
}
```

Is `peterson_lock_alt` an acceptable replacement for `peterson_lock`? If so, justify. If not, give a problematic interleaving, and state why it is problematic. Remember that we are assuming sequential consistency.

No, the replacement doesn't work. Consider the following interleaving:

T0: `p->wants[0] = 1;`

T1: `p->wants[1] = 1;`

T0: `while(p->wants[1]);`

T1: `while(p->wants[0]);`

Both threads spin forever. This is deadlock (technically, it's livelock, since neither thread is actually blocked, just spinning endlessly, but regardless, it's a progress issue.).

7. [8 points] As in class, we assume that threads are preemptively scheduled by the OS. Also, synchronization primitives (mutex acquire, condition variable wait, etc.) are system calls. *Note that the process state diagram applies to individual threads.* (If you don't remember the diagram, don't worry; you can still do the problem.) For example, `READY` means that the OS is free to run a thread's user-level instruction from the thread's saved instruction pointer, and `BLOCKED` means that the OS cannot run the thread (for example, because some request that the thread made is not complete). Consider the following code structure. Note that when the thread gets to `wait`, it transitions from `RUNNING` to `BLOCKED`:

```
Mutex m;
Cond cv; // condition variable

void f() {
    m.acquire();
    cv.wait(&m):
    m.release();
}
```

Is the following statement True or False? "Once the thread is `BLOCKED` above, it will not transition to `READY` until another thread signals or broadcasts on `cv`." Justify using one sentence.

False. As we discussed in class, the threading package can "wake a thread up" from `wait` at any time. That means, given the setup, that the kernel could bring the thread out of the `BLOCKED` state.

8. [6 points] You are programming an application that has a very large array as a data structure, and multiple threads of control concurrently operate on the data structure. Performing a given operation on the array requires a thread to modify several array items at once, but the application logic requires all of the modifications associated with a given operation to happen atomically. Also, you cannot predict at compile time *which* items your code will be accessing together.

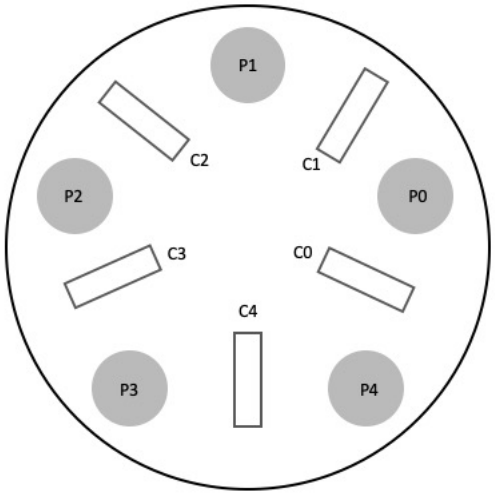
You approach the problem as follows. (1) For maximum concurrency, you create a lock for each item in the array. (2) When a thread performs an operation, it: (a) acquires all of the locks for the items that

the operation needs to modify, then (b) modifies the array items, and then (c) releases all of the locks. This approach provides the required atomicity, but it creates the risk of deadlock.

In this scenario, what do you do to avoid deadlock? Answer briefly (no more than two sentences).

Acquire the locks in some partial order, say in order of array index. This requires some logic before an operation to sort the order of requested items.

9. [15 points] In this question, you will solve a variant of the classic dining philosophers problem. There are five philosophers seated around a circular table, there is a *single* chopstick between every two philosophers, and a philosopher picks up the *two* adjacent chopsticks (on the philosopher's left and right) in order to eat. If a philosopher cannot get both chopsticks, the philosopher should not pick up either, since that would correspond to pointlessly claiming a resource. Here is a visualization:



We model this problem as follows. The table is global state. Each philosopher has an id `phil_id`; this id is a number from 0 to 4. Each philosopher is a thread, with the following pseudocode:

```
Table table;    // global

while (1) {
    think();
    table.getChopsticks(phil_id);
    eat();
    table.putChopsticks(phil_id);
}
```

Your job is to implement the `Table` as a monitor: think through (and optionally write down) the state variables and the synchronization objects. Implement the methods. Follow our class's coding requirements, including the concurrency commandments. A few notes:

- The next page provides some structure, including two helper functions, `left()` and `right()`.
- Your solution must be free from the possibility of deadlock.
- We will not insist on starvation-freedom: as with the monitor programming we have seen in class, it is acceptable if a philosopher could conceivably starve, but this should happen only if a philosopher is very unlucky. If this bullet confuses you, ignore it (it's a technicality, which we mention because this is a difference from the "official" dining philosophers problem).

```
// Helper functions. Given a philosopher's id number, these return the
// number of the left chopstick and the right chopstick. (Chopsticks are
// numbered from 0 to 4.)

int left (int phil_id) { return phil_id; }
int right(int phil_id) { return (phil_id + 1 ) % 5; }

class Table {

  public:
    Table(); // Initializes state and synchronization variables
    void getChopsticks(int phil_id);
    void putChopsticks(int phil_id);

  private:
     // FILL THIS IN




};

// Here and on the next page, give the implementations of
//     Table::Table()
//     void Table::getChopsticks(int phil_id)
//     void Table::putChopsticks(int phil_id)
```

Space for code and/or scratch paper

Data members in Table:

```
class Table {

....
  private:

     Mutex m;
     Cond  cv;

     int chopstick_status[5];
}
```

Methods:

```
Table::Table()
{
    m.init();
    cv.init();
    memset(chopstick_status, 0, sizeof(chopstick_status));
}
```

```
void
Table::getChopsticks(int phil_id)
{
    m.acquire();

    while (chopstick_status[left(phil_id)] || chopstick_status[right(phil_id)])
        cv.wait(&m);

    chopstick_status[left(phil_id)] = 1;
    chopstick_status[right(phil_id)] = 1;

    m.release();

}
```

```
void
Table::putChopsticks(int phil_id)
{
    m.acquire();

    chopstick_status[left(phil_id)] = 0;
    chopstick_status[right(phil_id)] = 0;


    cv.broadcast(&m);

    m.release();

}
```

IV Virtual memory (20 points)

10. [10 points] Consider a TLB which can store 4 mappings (the TLB is fully associative, meaning that any entry can store any mapping; if this parenthetical confuses you, you can ignore it). *The TLB uses LRU for replacing entries.* Below you will write C code to compute the sum of all integers in an array `a`, which is 6 pages in length; you will do this in a way that maximizes the number of TLB misses (equivalently, minimizes the number of TLB hits).

A few things to note:

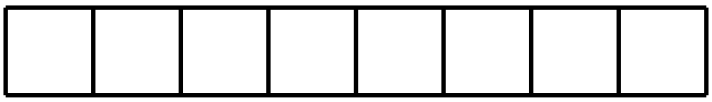
- The array is allocated to be page aligned, meaning that the first element in the array is at the beginning of a page.
- Your program can assume that the constant `PAGE_SIZE` is the size of a page in bytes and that `sizeof(int)` is the size of an integer.
- You can ignore the effect on the TLB from fetching code; in other words, you can assume that the only memory references that affect the TLB are loads from array `a`. (In real systems, there are separate TLBs for instructions and data; this question is focusing on the data TLB.)
- You can further assume that the processor does nothing else while your code is running; that is, you don't need to worry about TLB flushes from context switches.

```
uint64_t tlb_unfriendly() {

    int *a = page_alloc(6 * PAGE_SIZE);
    populate_array(a); // sets the integers in the array
    uint64_t sum = 0;

    /* YOUR CODE HERE: compute sum in the most TLB-unfriendly way possible */









    return sum;
}
```

```
for (int i = 0; i < PAGESIZE/sizeof(int); i++)
    for (int j = 0;  j < 6; j++)
        sum += a[j*PAGESIZE/sizeof(int) + i];
```

11. [10 points] Consider a tiny machine that has 8-bit virtual addresses, 16-byte pages, and 64 bytes of RAM (awww, this machine is so cute).

In the representation of the 8-bit virtual address below, indicate which bits determine the page (use *A* for such bits), and which bits determine the offset (use *B* for such bits).



A A A A B B B B

What is the maximum size in bytes of a process's virtual address space?

$2^8 = 256$ bytes.

How many physical frames exist in this system?

$64/16 = 4$.

If the page size were reduced to 8 bytes but the address remained 8 bits, what effect would that have on the maximum size of a process's virtual address space?

It would have no effect.

V Scheduling, reading, and feedback (15 points)

12. [8 points] Consider a system with three processes **A**, **B**, and **C**:

Process A is a compute-intensive job that never performs a blocking call, and only gives up any assigned CPU resource when preempted.
Process B whenever scheduled can perform up to 5 ms of computation but then must read from disk. Assume disk reads take 5 ms.
Process C when scheduled executes for 1 ms and then sleeps for 9 ms.

Assume that the system has a single processor (core), and performs preemptive scheduling. Consider a scheduling policy: *round robin, with a scheduling quantum (timer interrupt interval) of 10 ms*.

State the fraction of CPU time that each process gets. By "fraction", we mean the long-term average over many time periods.

A: 5/8, B: 5/16, C: 1/16

13. [6 points] Which of the following statements about the Therac-25 paper is true?

Circle all that apply:

- **A** The authors interviewed some of the survivors of the accidents.
- **B** The authors interviewed some of the hospital physicists.
- **C** The authors interviewed some of the engineers at AECL (the Therac-25's manufacturer).
- **D** The Therac-25 manufacturer estimated that the probability of a mode error was higher than $1/3$.
- **E** One of the Therac-25 users learned how to reliably trigger a massive overdose.
- **F** The gruesome toe injury (this was in Yakima, Washington) caused the FDA to request a corrective action plan (CAP) from the Therac-25 manufacturer.
- **G** One of the Therac-25 software errors involved overflowing a byte, called `class3`.

E, G

14. [1 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.

End of Midterm