Name	UBID	Seat	Side

Question:	1	2	3	4	5	6	7	8	Total
Points:	10	5	5	5	5	5	5	20	50
Score:									

CSE 421/521 Midterm Exam

25 Mar 2015

Please fill out your name and UB ID number above. Also write your UB ID number at the bottom of each page of the exam in case the pages become separated.

This midterm exam consists of three types of questions:

1. **10 multiple choice** questions worth 1 point each. These are drawn directly from lecture slides and intended to be easy.
2. **6 short answer** questions worth 5 points each. You can answer as many as you want, but we will give you credit for your best four answers for a total of up to 20 points. You should be able to answer the short answer questions in four or five sentences.
3. **2 long answer** questions worth 20 points each. **Please answer only one long answer question.** If you answer both, we will only grade one. Your answer to the long answer should span a page or two.

Please answer each question as **clearly** and **succinctly** as possible. Feel free to draw pictures or diagrams if they help you to do so. **No aids of any kind are permitted.**

The point value assigned to each question is intended to suggest how to allocate your time. So you should work on a 5 point question for roughly 5 minutes.

There are **three** scratch pages at the end of the exam if you need them. If you use them, please clearly indicate which question you are answering.

I have neither given nor received help on this exam.

Sign and Date: ______________________________

Multiple Choice

1. (10 points) Answer all **ten** of the following questions. Each is worth **one** point.

   (a) What singer was *not* played before class this semester?
   ◯ Tony Rio ◯ Amanda Banana ◯ Stacy Keys ◯ Lisa Lee

   (b) All of the following are critical section requirements *except*
   ◯ mutual exclusion. ◯ concurrency. ◯ progress. ◯ performance.

   (c) Interprocess communication is harder than intra-process communication.
   ◯ True ◯ False

   (d) All of the following are private to each process thread *except*
   ◯ stack. ◯ file handles. ◯ registers.

   (e) What action does *not* require the kernel?
   ◯ Switching between two threads. ◯ Reading from a file. ◯ Creating a new process. ◯ Altering a virtual address mapping.

   (f) What part of the address space is not initialized using information from the ELF file?
   ◯ Code segment. ◯ The heap. ◯ Uninitialized static data. ◯ Initialized static data.

   (g) The Rotating Staircase Deadline Scheduler is most similar to which other scheduling algorithm?
   ◯ Lottery scheduling. ◯ Round-robin. ◯ Multi-level feedback queues. ◯ Random.

   (h) Which of the following is *not* a part of making a system call?
   ◯ Arranging the arguments in registers or on the stack. ◯ Loading the system call number into a register. ◯ Generating a software interrupt. ◯ Allocating memory in the heap using `malloc`.

   (i) What information would probably *not* be stored in a page table entry?
   ◯ The location on disk. ◯ Read, write or execute permissions. ◯ The process ID. ◯ The physical memory address.

   (j) Paging using fixed-size pages can suffer from internal fragmentation.
   ◯ True. ◯ False.

UB ID: | | | | | | | |

Short Answer

Choose **4 of the following 6** questions to answer. You may choose to answer additional questions, in which case you will receive credit for your best four answers.

2. (5 points) One way to eliminate deadlock when acquiring multiple locks is to eliminate cycles. Describe another way to avoid deadlock (2 points) and how to implement it (3 points).

3. (5 points) Describe two changes to the OS virtual memory system that you would need or might want to make to accommodate 48-bit virtual and physical addresses (1 point each). For each, describe how it would affect the computation or memory overhead of virtual to physical translation (2 points each).

UB ID:

4. (5 points) Describe how a scheduling algorithm might improve resource allocation by observing processes communicating using pipes (4 points). What is a tradeoff involved in this decision (1 point)?

UB ID:

```
#include <stdio.h>
#include <stdlib.h>

int32_t foo[1024];

int
main(int argc, const char * argv) {
  buffer = (char *) malloc(10240); // You can assume that malloc succeeds.
  int32_t bar[1024];
  // <-- Here -->
  struct fooer[24];
}
```

5. (5 points) Examine the simple program above. At the point in its execution indicated, *at minimum* how many 4K pages of memory will it require in each segment: code, stack and heap? Justify your answer (5 points).

   Note that you can ignore dynamically-loaded libraries, and assume that `malloc` does not consume any memory for its own data structures (if only).

6. (5 points) Some systems provide separate exception handlers for TLB-faults and all other kinds of exceptions, allowing the operating system to handle them separately. First, describe why this might be a useful feature for the hardware to provide (2 points). Second, suggest one way that the operating system might take advantage of this differentiation (3 points).

VPN	SLT
0	
1	450
2	
3	
4	
5	120
6	
7	
8	
9	320

(a) **TLT**

VPN	PPN
0	9
1	84
2	
3	39
4	
5	
6	14
7	10
8	
9	

(b) **SLT at 450**

VPN	PPN
0	31
1	
2	
3	72
4	62
5	
6	40
7	43
8	
9	

(c) **SLT at 320**

VPN	PPN
0	94
1	
2	
3	10
4	13
5	
6	8
7	
8	
9	

(d) **SLT at 120**

Table 1: **Process Page Tables**

7. (5 points) The `HappyStudent` architecture has byte-addressable memory with 10-byte virtual pages. The `HappyOS` operating system uses two-level pages tables to support 1000-byte virtual address spaces by dividing the virtual page number into a 10-byte second-level index and using the rest as the top-level index.

   First, given the virtual address `273`, identify the virtual page number, the offset, and the first and second level page table indices (1 point).

   Second, given the top-level page table (TLT) and second-level page tables (SLT) above for the currently-running process provided above, indicate the result of the following four virtual to physical page translations (1 point each).

   1. `165`
   2. `578`
   3. `57`
   4. `900`

UB ID:

Long Answer

Choose 1 of the following 2 questions to answer. **Do not answer both questions.** If you do, we will only read one, most likely the one that looks shorter and more incorrect. If you need additional space, continue and clearly label your answer on other exam sheets.

8. (20 points) Choose one of the following two questions to answer:

   1. **Interface Size Tradeoffs.** Linux and other UNIX variants provide a fairly thin system call interface, consisting of roughly 300 system calls. You are now familiar with a subset of these calls, particularly those you implemented for ASST2. In contrast, the Windows kernel interface contains almost an order-of-magnitude more system calls: approximately 3000.

      First, describe the tradeoff between small and large system call interfaces. What is good about a thin interface? How about a thick interface (4 points)? Second, provide three examples illustrating the problems with the thin system call interface you are familiar with. For each, describe how adding additional system calls would help, and what their interface would be (4 points each).

      Finally, for *one* of your new system calls describe the operating system changes that would be required to implement it. You are free (but not required) to reference OS/161-specific implementation details in your answer (4 points).

   2. **Concurrent System Calls.** As core counts have increased, the OS has become a source of potential bottlenecks for multithreaded applications. One source of poor scaling behavior is when two system calls cannot be executed concurrently, and in some cases redesigning the system call interface can improve concurrency and performance on multicore systems.

      Consider the code below and a multithreaded application where, on an N core machine, N thread run `openclose`[1]. First, describe the OS performance bottleneck that this code might encounter on a multicore system. Note that `open` is implemented to return the lowest available file descriptor, but very few apps rely on this behavior. Be sure to describe the source of the bottleneck clearly (10 points). Second, describe how alter the system call interface to remove this particular performance bottleneck (10 points).

```
void openclose() {
  // Repeatedly open and close a file.
  while (1) {
    int fd = open("/tmp/foo");
    close(fd);
  }
}
```

[1]Don't worry about why an app would do anything so seemingly meaningless. If it's helpful, just assume your boss wrote it and is wondering why it doesn't work well.

UB ID:

Scratch. **Please indicate what question you are answering.**

First, describe the tradeoff between large and small system call interfaces. Large interfaces allow processes to be more specific about exactly what they want the operating system to do, which eventually leads to fewer system calls and fewer transitions into the kernel, which could improve performance. (Large systems call interfaces also *might* expose more features, although you were free to assume that the feature set exposed by both large and small interfaces was the same.)

In contrast, small interfaces are easier for the operating system to implement and maintain and make it more clear to application developers which of multiple ways to do things is the correct way. If there's only one way to accomplish something, it's what the OS developers will work to support; if there are multiple ways, it's not necessarily clear which is the most up-to-date. This was the canonical complaint about the Windows interface by third-party developers, which was (partly due to backwards compatibility), with half-a-dozen options available to accomplish a particular task the right approach wasn't clear. However, if you *worked* for Microsoft (say, deveoping Internet Explorer), you could just email the guys at the Windows team and ask "Hey, which way of reading data from a file has the best perfomance?"

Second, provide three examples illustrating the problems with the thin system call interface you are familiar with. The best way to do this is to show where multiple system calls are required to accomplish something useful that could be done using a single call. In none of the cases that we provide as solutions does the implementation required much past adding a new system call number and handler, but your solution may be different.

1. `fork` followed by `exec`: this was mentioned in class. Creating a child and loading a new image in one new call isn't just easier on the process, it also allows the OS to avoid some of the memory overhead associated with `fork`. Implementing this requires adding the system call and handling errors properly, which could occur either during `fork` or `exec`.
2. `open` followed by `read` or `write`: if I want to read the entire contents of a file into memory, why not just have a single call that takes a path and a buffer and performs the operation without requiring two calls and the maintenance associated with the process file table? Implementing this requires adding the new system call and handler, but not necessariy much else.
3. `read` or `write` and `lseek`: the fact that `read` and `write` manipulate the file offset implicitly and therefore require separate calls to `lseek` is both annoying and may cause problems when multiple threads try to use the same file handle. A separate system call could perform a `read` or `write` using an offset provided as an argument, eliminating the need to use `lseek`. Implementing this requires adding the new system call.

Scratch. **Please indicate what question you are answering.**

This question was inspired by this fantastic MIT paper on scalable interfaces. We may look at this in more detail later in the semester.

First, describe the OS performance bottleneck that this code might encounter on a multicore system. The key was to process the note that we left you, noting that `open` currently is implemented to return the lowest-available file descriptor. However, processes don't rely on this behavior and really only treat the file descriptor as a reference to the file handle, so you could alter this.

And you want to, because it's the performance bottleneck. Each time `open` is called it has to lock and search the file table from the beginning to find the lowest-available entry. (Note that in this case we *cannot* apply the read-lock-read optimization used in the 2014 midterm, because the file descriptor is supposed to be the lowest available.) The locking and long critical section caused by the required linear search have the effect of unnecessarily serializing concurrent calls to `open` that could proceed independently on multiple cores.

Second, describe how to alter the system call interface to remove this particular performance bottleneck. Once you see the bottleneck this is pretty obvious: change open to not guarantee that it will return the lowest-available file descriptor. This has two benefits. First, we can start the search at a random location, meaning that we will find an available file descriptor more quickly *even* if decide to grab a lock during the process. And if you were even more clever (and present at our midterm review), you probably noticed that you could apply the read-lock-read optimization here as well and avoid holding a lock outside of the loop.

Scratch. **Please indicate what question you are answering.**

UB ID:

Scratch. **Please indicate what question you are answering.**

UB ID:

Scratch. **Please indicate what question you are answering.**

BLANK