Write the answers this exam by filling in the blanks in the `answersheet.txt` file provided.

Write the honor pledge at the top of the answer sheet.

You may look at notes, textbooks, lectures slides, and general Internet resources about course topics (for example, slides and lecture notes from other OS courses, Linux manual pages, references for the POSIX API, technical papers and presentations) when completing this exam, but you may not search for answers to specific questions (e.g. by copy-and-pasting the question into a search engine though we suspect that would usually be ineffective), ask questions of others (including via instant message, Stack Overflow, etc.), receive personal tutoring, etc.

If you think a question is unclear, ambiguous, etc. do not try to contact us about this during the exam. Instead, explain why on your answersheet and make your best guess as to what was intended.

When you are done, upload your answer sheet text file to kytos by 6 PM eastern time on 5 May 2020. (You can submit multiple times; we will grade your last submission.)

1. Suppose an operating system has a page cache with 6 physical pages and an **LRU** replacement policy. These page cache is used by exactly two processes with the following *repeating* access patterns (where A, B, C, etc. represent physical pages);

Process 1:	A B C D A B C D A B C D A B C D ...
Process 2:	E F G E F G E F G E F G E F G ...

   Initially:

   - process 1 is running and its pages (A, B, C, D) are loaded into the page cache; the remaining pages in the page cache are empty

   Then process 2 starts running, and the system executes the two processes as follows:

   - when no page replacements are required, processes perform one page access per time unit;
   - when a page replacements is required, it takes 4 time units in addition to the time required for the actual access to the page (to perform the relevant I/O), during which another process will be allowed to run;
   - when switching away from a process to perform page replacement, the process gives up any remaining time in its time quantum. After it completes it pages replacement, the scheduler will run it again when another process gives up or completes its time quantum (or if there are no other runnable processes).
   - processes are run in a round-robin fashion with 12 time unit time quantums;
   - only one page replacement can occur at a time; if a second page replacement is required while one is in progress, it will wait until the first one completes;
   - the context switching overhead is neglible

   Because the two processes' working sets together are larger than the available physical memory, it is likely that the processes will not be able to run efficiently due to limited memory available.

   (a) (10 points) Over the long run, about what amount of CPU time will each process get? Explain briefly.

   Solution: The first time process 2 runs, it will immediately give up the CPU to bring in page E (replacing an empty page). Then second time it runs, it will successfully access page E, then give up CPU to bring in page F (replacing an empty page). Then it will successfully access page F and give up the CPU to bring up page G (replacing E), and so on (with E replacing F, F replacing G, G replacing E, etc.). Since it gives up its time quantum each time it runs this way, it will get one unit of useful work done for every unit of work process 1 gets done. So process 1 will get about 12/13ths of the CPU and proecss 2 will get about 1/13th.

   (b) (6 points) Suppose we decreased the time quantum for the round-robin scheduling to 4 time units. How would this change the amount of CPU time each process gets? Explain briefly.

   Solution: it would change it to 4/5th for process 1 and 1/5th for process 2

2. A device controller for a hard drive exposes memory-mapped control registers containing the following

   - a memory-mapped control register indicating whether to perform a read or write operation next
   - a memory-mapped writable list of up to 30 eight-byte sector numbers
   - a memory-mapped writable list of up to red 30 (not in distributed version) memory addresses to use as a buffer for each of the 30 sectors
   - read-only memory mapped control registers for the controller to specify whether any pending I/O operation is complete
   - a writable memory-mapped control regsiter that when written starts a read or write operation for all the valid sector numbers in the list

   Whenever an I/O operation completes, the hard drive controller triggers an interrupt. Since up to 30 sectors can be specified in the control registers, an I/O operation can be a read or write of up to 30 sectors. Only one I/O operation can be performed at a time.

   Consider an operating system using this hard drive along with a page cache where pages are 4096 bytes and sectors on the disk are also 4096 bytes.

   Suppose an application running on this OS reads a 40 page file from beginning to end using mmap and the entire file is not cached. The application accesses each byte of the file once in a loop, and no other activity being performed on the OS (by any application) requires disk I/O.

   (a) i. (5 points) Assume the operating system does *not* perform readahead, so it only reads the pages in on demand. How many I/O operations will it trigger on the device?

   Solution: it will have to read each page individually, so 40 times (once for each page)

   ii. (4 points) If the operating system does *not* perform readahead, it will most likely start the read operation from:

   A. code run in userspace (outside of an exception/fault/trap handler) in the application
   B. a page fault handler triggered by writing a read-only page
   C. an interrupt handler triggered by the disk controller
   D. a system call handler
   [✓ E. a page fault handler triggered by reading or writing an invalid page]
   F. a timer interrupt handler

   iii. (4 points) If the operating system does *not* perform readahead, it will most likely determine whether the read operation was successful from

   A. a timer interrupt handler
   B. code run in userspace (outside of an exception/fault/trap handler) in the application
   [✓ C. an interrupt handler triggered by the disk controller]
   D. a page fault handler triggered by writing a read-only page
   E. a system call handler
   F. a page fault handler triggered by reading or writing an invalid page

   iv. (3 points) After starting one of the I/O operations from the previous part (and before the I/O operation completes), what would make most sense for the operating system to do?

   Solution: if possible, it should run another process; otherwise, it should remain idle

   (b) i. (5 points) Suppose the operating system *does* perform readahead. What is the *minimum* number I/O operations it could perform on the device?

Solution: it can do reads of up to 30 pages at a time, so 2 times in the best case

ii. (3 points) After starting one of the I/O operations triggered by readahead (and before the I/O operation completes), what would make most sense for the operating system to do?

Solution: most of the time when it is doing readahead, it should be able to continue running the program; it may (especially on the first ten pages) not able to to do so all the time, in which case it should switch to another program until the read finishes

3. Consider the following filesystem designs:
   - Filesystem A: an inode-based filesystem where
     - the block size is 4096 bytes
     - block pointers are four bytes
     - inodes contain 12 direct block pointers
     - inodes contain 1 indirect block pointer
     - inodes contain 1 double-indirect block pointer
     - inodes contain 1 triple-indirect block pointer
     - block groups are used, where each block group has 8192 data blocks and 256 inodes
     - directory entries are 64 bytes
     - there is no support for fragments or extents
   - Filesystem B: same as A, but with 2048 byte blocks
   - Filesystem C: same as A, but with 8192 byte blocks and support for fragments as follows:
     - fragments are 2048 bytes
     - a file less than or equal to a fragment in size can be stored in a single fragment
     - files greater than a fragment in size must use at least one block (even if they could use less space if it were possible to store them using fragments)

   (a) Give an example where each of the filesystems described above would require less bytes of data blocks than each of the other designs, assuming that data blocks are allocated in the most space-efficient way possible for each filesystem.

   When counting the number of bytes of data blocks, do not include inodes, free block maps, superblocks, etc., but do count all the bytes of any data blocks which are partially used (e.g. if only half the block contains useful information). Also count any data blocks (outside of inodes) allocated to store block pointers.

   Describe your example *and calculate much space would be used with each filesystem above.* (Please show your calculations so we can give appropriate partial credit.)

   i. (8 points) Filesystem A:

   Solution: a root directory containing 62 regular files, each of which is 49152 bytes. With A: uses 1 data block for directory (giving space for a **.** and **..** directory entry), plus $12 \times 62 = 744$ data blocks for the regular files, a total of 2980KB. With B, will require an indirect block for each of the regular files, giving an additional 2KB for each file. With C, will use an extra 4KB for the root directory.

   ii. (8 points) Filesystem B:

   Solution: a root directory containing 1 regular file of 4096 bytes. With B: uses three blocks, for 6KB. With C: root directory uses a 2048 byte fragment, regular file uses one 8192 byte block, for 16KB (6KB of which is usable for fragments); With A: uses 2 blocks, for 8KB

   iii. (8 points) Filesystem C:

   Solution: a root directory containing 1 regular file of 98304 bytes. With C: uses 1 block (root dir fragment) + 12 blocks for file, for 104KB; With A: uses an extra 2KB for root dir + extra 4KB for indirect block; With B: uses an extra 2KB for indirect block

   (b) (4 points) When using block groups, a goal is to contain the data and metadata for a directory and its files within one block group. In filesystem A, if it used block groups, which of the following could have its data and metadata contained within one block group? Select all that apply. **Select all that apply.**

A. a directory of 200 files, each of which is 165000 bytes *41 data + 1 block pointer blocks per file = 8400 blocks*

B. a directory of 300 files, each of which is 100000 bytes *too many inodes*

C. a directory of 10000 empty (zero-length) files *too many inodes*

D. a directory of 150 files, each of which is 217000 bytes *53 data + 1 block pointer blocks per file = 8100 blocks (+3 for directory)*

4. A system uses two-phase commit for a distributed database that holds information about product inventories and orders. The information is split up so machine A stores the product inventories and machine B stores the orders.

   To dispatch an order, a transaction will update both machine A and machine B using two-phase commit.

   For the questions below, suppose two transactions occur at the same time. Each of these transactions has a different coordinator machines C and D, but in both transactions machine A and machine B are involved as workers.

   Both transactions involve marking the last unit of inventory for a product on machine A as used, and therefore it is not possible for both to complete.

   (a) (5 points) If machine C has received an agreement to commit from machines A and B, then which of the following statements could be true? **Select all that apply.**

      A. machine D, performing its transaction concurrently, may receive an agreement to commit from machine B

      B. machine D's transaction may abort

      C. if machine C fails before telling machine A and B to commit or abort the transaction, then, before machine C recovers from its failure, machine D could end up committing its transaction

      D. machine D, performing its transaction concurrently, may receive an agreement to commit from machine A

      E. machine C may tell machine A and B to commit the transaction

   (b) (5 points) If machine C sends a message to commit to machines A and B, then which of the following statements could be true? **Select all that apply.**

      A. if machine C becomes totally unreachable after A and B receive the message to commit, then machines A and B can still commit the transaction

      B. if machine C crashes and reboots and is not able to get any acknowledgment afterwards from machine A or B (despite multiple attempts to do so), it can send a message to abort to machines A and B

      C. if machine C's message to commit is not received by machines A and B and machine C is not able to get any acknowledgment from machine A or B (despite multiple attempts to do so), it can send a message to abort to machines A and B

      D. if machine A crashes before receiving the message from machine C, then machine B will not be able to commit until machine A reboots

      E. if machine A crashes after recording the message to commit from C in its log but before machine A performs the updates for the transaction, then machine B will not be able to commit until machine A reboots

(c) (5 points) Suppose machine C has receives an agreement to commit from machines A and B, and immediately after this, all of machines A, B, and C crash and are rebooted. The machines perform recover using their logs. What could happen during this recovery? **Select all that apply.**

A. machine A has no record of the transaction after it is rebooted until it is informed about it again by machine C's recovery code

B. machine C asks machine A and machine B for their votes (agree to commit or agree to abort) again, even though it received them before the crash

C. if machine D contacts machine A after rebooting before machine C does, then machine A will perform machine D' s transaction even though it conflicts with machine C's transaction

D. machine C aborts the transaction

E. machine A receives a message from machine D asking it to perform a conflicting operation and so decides to abort the traction being performed by machine C

5. Suppose a virtual machine is implemented using the trap-and-emulate strategy we discussed in lecture and using shadow page tables filled in on demand (in response to page faults) like we discussed in lecture. The following occurs:

   A program running in the guest OS on the virtual machine uses a system call to send a message on the network and then uses another system call to receive a reply to this message.

   To implement sending and receiving messages, the guest OS sets up a buffer in its memory on behalf of the program. The guest OS determines what would be the physical address of the buffer if the guest OS were not running in a virtual machine. Then it writes that address to a control register on the emulated network device controller. After doing this, the emulated network controller triggers an interrupt in the guest OS to indicate that receiving or sending the message is complete.

   (a) (8 points) What operations involved in the program in the guest OS sending and receiving messages over the network involve exceptions (of any kind, including interrupts, traps, etc.) on the real hardware (that is, in the host OS), which would **not** require exceptions if the guest OS were running directly on the hardware (that is, outside of a virtual machine)?

   Solution: writing to control registers on the network controller; switching from kernel back to user mode

   (b) (4 points) To emulate the network device controller, the virtual machine needs to read or write from the buffer location specified by the guest OS. How should it do this?

      A. by triggering a page fault in the guest OS to perform the translation from the buffer location to a usable address

      B. by looking up the buffer location in the hypervisor's mapping from guest OS physical addresses to machine (host OS physical) addresses, then adding that physical address to a page table entry in the host OS and accessing the virtual address corresponding to the added page table entry

      C. by dereferencing a pointer containing the buffer location while the shadow page table is active (filling in a new shadow page table entry, if necessary)

      D. by looking up the buffer location in the hypervisor's mapping from guest OS physical addresses to machine (host OS physical) addresses, then dereferencing a pointer with that address within the host OS kernel

      E. by looking up the buffer location in the guest page table, and accessing the address contained in the guest page table in the host OS kernel by dereferencing a pointer containing that address

6. (5 points) Which of the following are appropriate strategies for an Unix-like operating system to enforce access control lists when a process attempts to open a file? **Select all that apply.**

   A. have the system call wrapper for open() mark the file to be opened as set-user-ID and attempt to execute it before trying to open() it

   B. check the access control list in the system call wrapper for open() included in the standard library

   C. have the system call handler for open() in the kernel require the process to pass a file descriptor for the directory containing the file in order to open the file

   D. add an extra user and group ID argument to the open() system call, and use it to check the access control list in the system call handler in the kernel

   E. check the access control list in the system call handler in the kernel using a user and group ID included in the process control block

7. (20 points) Suppose we want to implement a thread pool that dynamically adjusts the number of threads it runs. To do this, it supports the following interface:

```
class Task {
public:
  virtual void Run() = 0;
};
void SubmitTask(Task *task);
void SetNumberOfThreads(int number);
```

SubmitTask submits a new task to the thread pool, calling its run method, and SetNumberOfThreads() adjusts the number of threads in the thread pool, either starting new threads or waiting for them as necessary. Initially the number of threads is 0. Supply the contents of the blanks in the incomplete implementation shown on the next page:

(Assume all necessary headers and forward declarations are included. Ignore minor syntax errors.)

```
int activeThreadCount = 0;
int targetThreadCount = 0;
std::deque<Task*> tasks;
pthread_mutex_t lock; pthread_cond_t queue_wait_cv; pthread_cond_t finish_cv;

void SubmitTask(Task *task) {
  pthread_mutex_lock(&lock);
  tasks.push_back(task);
  pthread_cond_signal(&queue_wait_cv);
  pthread_mutex_unlock(&lock);
}

void SetNumberOfThreads(int newCount) {
  pthread_mutex_lock(&lock);
  targetThreadCount = newCount;
  pthread_cond_broadcast(&queue_wait_cv);
  while (activeThreadCount > targetThreadCount)
    pthread_cond_wait(&finish_cv, &lock);
  while (activeThreadCount < targetThreadCount) {
    pthread_t thread;
    pthread_create(&thread, NULL, RunThread, NULL);
    pthread_detach(thread);
    activeThreadCount += 1;
  }
  pthread_mutex_unlock(&lock);
}

void *RunThread(void *ignored_argument) {
  while (true) {
    pthread_mutex_lock(&lock);
    while (tasks.size() == 0 && activeThreadCount <= targetThreadCount) {
      pthread_cond_wait(&queue_wait_cv, &lock);
    }
    /* was misprinted with reversed condition in real exam */
    if (activeThreadCount > targetThreadCount) {
      activeThreadCount -= 1;
      pthread_cond_broadcast(&finish_cv);
      pthread_mutex_unlock(&lock);
      break;
    }
    Task *t = tasks.front();
    tasks.pop_front();
    pthread_mutex_unlock(&lock);
    t->Run();
  }
  return NULL;
}
```