CS 4410 Operating Systems Final, Fall 2015
Profs. Bracy and Van Renesse

NAME: ____ NetID:___________

- **This is a closed book examination. You have 120 minutes. No electronic devices of any kind are allowed.**
- You must fill in your name and NETID above and at the top of each (odd-numbered) page. If you fail to do so, we will take off 1 point for each omission.
- Show your incomplete work for partial credit. Make any other assumptions as necessary and document them. Brevity is key.
- Please write your solutions within the provided boxes as much as possible. Write clearly. Use the scratch paper at the end if you need to practice your answer first.

Question	Points Possible
0: Omitting Name or NetID	−2
1: Multiple Choice	15
2: When you come to a fork()	10
3: Bounded Buffer	15
4: Priority-Based Scheduling	15
5: Network routing	20
6: Knab Bank (concurrency)	10
7: All’s Well That Ends Well	15
TOTAL	100

[15 pts] 1. Multiple Choice Questions

For all Multiple Choice questions, there is only ONE best answer.

[3] a) Replacement Policies. Which of the following statements about cache replacement policies is **true**?

Your Answer:

(A) LRU replacement policy will always outperform a FIFO replacement policy.
(B) LRU is rarely implemented in practice because the overhead is too great.
(C) The OPT replacement policy is rarely implemented in practice because the hardware cost is too great.
(D) The OPT replacement policy is rarely implemented in practice because Belady's Anomaly is too likely.
(E) The MRU (Most Recently Used) replacement policy is effective across a variety of application types.

[3] b) Security. Which approach to security can be summarized with the phrase "Don't contaminate the information!"

Your Answer:

(A) Access Control Lists ("Who can do what with each file.")
(B) Linux Access Rights ("What can each class of user do with each file.")
(C) Capabilities ("What you can do depends on what you have.")
(D) Bell-La Padula Model ("No read up, no write down")
(E) Biba Model ("No write up, no read down")

[3] c) System Calls. When making a system call, why doesn't the processor create a stack frame for the system call on the user stack?

Your Answer:

(A) Actually, the processor *does* create the system call's stack frame on the user stack.
(B) Because the kernel stack frame lives in a different virtual memory address space.
(C) Because the kernel stack frame needs to be accessible even after the system call is complete.
(D) Because the kernel should not be allowed to access data which the user placed on its stack frame.
(E) Because the user process should not be allowed to access data that the kernel could have temporarily placed on the user stack if the system call stack frame were there.

[3] d) Reliable Transport. Which of the following is ***NOT TRUE*** about TCP?

Your Answer:

(A) TCP is the predominant transport protocol for web traffic.
(B) A TCP connection is made with a 3-way handshake.
(C) A timeout that is too long will lead to unnecessary transmissions.
(D) If a packet goes missing, the sender could detect this via an acknowledgement of a packet with a sequence number higher than the one that went missing.
(E) When both sides of a TCP connection are sending data, it is okay for them to use overlapping sequence numbers.

Your Answer:

[3] e) Page Faults. Which of the following causes of a Page Fault ***WILL NEVER*** result in disk read access?

(A) Actually, all page faults result in a disk read access.
(B) Accessing an address in the stack segment from a frame that has never been accessed before.
(C) Accessing an address in the code segment that has never been accessed before.
(D) Accessing an address that *has* been read earlier by the same process.
(E) Accessing a global variable.

NETID:________

[10pts] 2. When you come to a fork(), take it!

Recall the following:

Function **fork** returns 0 to the child process and the child's process identifier to the parent.
Function **wait** returns −1 if there is an error, *e.g.,* when the executing process has no child.

For this question we are assuming that neither **fork** nor **wait** fail.

Consider the C program below:

```
main() {
   if (fork() == 0) {
      if (fork() == 0) {
         printf("3");
      }
      else {
         pid_t pid;
         int status;
         if ((pid = wait(&status)) > 0) {
            printf("4");
         }
      }
  }
  else {
     if (fork() == 0) {
        printf("1");
        exit(0);
     }
     printf("2");
  }
  printf("0");
  return 0;
}
```

Out of the 5 outputs listed below, which are valid outputs of this program?
Assume that all processes run to normal completion.

A. 2030401
B. 1234000
C. 2300140
D. 2034012
E. 3200410

Your Answer:

[15pts] 3. Bounded Buffer

You are to solve the classic Producer/Consumer problem with a bounded buffer. The bounded buffer essentially implements a blocking queue. There is a shared class BoundedBuffer with a buffer of size *N*, a "head" index where new entries are inserted, and a "tail" index where new entries are removed. Initially, head and tail are 0. You are to write two methods:

- `produce(item)` adds a new item to the bounded buffer, and should block if the buffer is full.
- `consume()` removes and returns an item, and should block if the buffer is empty.

Use locks (mutexes) and Mesa-style condition variables. No busy waiting: when a thread cannot continue, it should block.

We have provided some skeleton code in pseudo-Python (your use of "self." to access class fields is optional). Add more variables and code as needed: Not all boxes need be filled necessarily (but at least some do).

```
N = 8   # number of slots in the bounded buffer
class BoundedBuffer:
  def __init__(self):
      self.buffer = [None] * N        # buffer of size N, initially empty
      self.head = 0                   # where new entries go in
      self.tail = 0                   # where entries are read

                    # add additional instance variables below if you need them

      [box]

  def produce(self, item):            # add a new item to the bounded buffer.  Block while full.

      [box]

          self.buf[self.head] = item
          self.head = (self.head + 1) % N

      [box]
```

class BoundedBuffer *continued*:

```
    def consume(self):  # Remove and return an item from the bounded buffer.  Block if empty.

            item = self.buf[self.tail]
            self.tail = (self.tail + 1) % N

            return item
```

The Twelve Commandments of Synchronization

Commandment 0. *Thou shalt live and die by coding conventions for synchronization.*
Commandment 1. *Thou shalt name your synchronization variables properly.*
Commandment 2. *Thou shalt not violate the abstraction boundaries provided by synchronization primitives, nor shalt thou try to change the semantics of well-established synchronization primitives, and thou shall look with disdain upon he who does.*
Commandment 3. *Thou shalt use monitors and condition variables instead of semaphores whenever possible.*
Commandment 4. *Thou shalt not mix semaphores and condition variables.*
Commandment 5. *Thou shalt not busy-wait.*
Commandment 6. *All shared state must be protected.*
Commandment 7. *Thou shalt grab the monitor lock upon entry to, and release it upon exit from, a procedure.*
Commandment 8. *Honor thy shared data with an invariant, which your code may assume holds when a lock is successfully acquired and your code must make true before the lock is released.*
Commandment 9. *Thou shalt cover thy naked waits.*
Commandment 10. *Thou shalt guard your wait predicates in a while loop. Thou shalt never guard a wait statement with an if statement.*
Commandment 11. *Thou shalt not split predicates.*
Commandment 12. *Thou shalt help make the world a better place for the creator's mighty synchronization vision.*

Prof. Emin Gün Sirer

[15pts] 4. Priority-Based Scheduling

Given are a set of jobs with a particular arrival time, a duration (the amount of CPU time it will need, in time units), and a priority. Given a choice between running two jobs, the one with the higher priority wins. If they have the same priority, the one with the earlier arrival time wins. There are no two jobs with the same arrival time. All jobs are purely CPU bound. A CPU should never sit idle if there are jobs to run. That is, the scheduler always runs when a job completes.

The turnaround time of a job is the time between arrival and the time the job finishes. The response time of a job is the time between arrival and the time the job is first scheduled.

[5 pts] a) Assuming there is no preemption, fill in the following table:

Job	Arrival Time	Duration	Priority	Turnaround Time	Response Time
A	0	25	0	25	0
B	15	25	1		
C	20	5	3		
D	40	5	2		

Optional: show your work by drawing a Gantt chart or a time table

[5 pts] b) Suppose the arrival of a job at time T causes the scheduler to run and select a job to run at time T, being able to preempt the job that was running. Fill in the following table:

Job	Arrival Time	Duration	Priority	Turnaround Time	Response Time
A	0	25	0		
B	15	25	1		
C	20	5	3		
D	40	5	2		

Optional: show your work by drawing a Gantt chart or a time table

[5 pts] c) Assume that the scheduler runs every five time units (i.e., at time 0, 5, 10, etc.) and also whenever a job finishes. This time the priority of a job is not pre-assigned and static, but determined by how long it has been on the run queue (aka *ready queue*) since the last time it ran (or since it arrived in case the job hasn't run yet). In case of a tie, the job that arrived earlier runs. Fill in the following table:

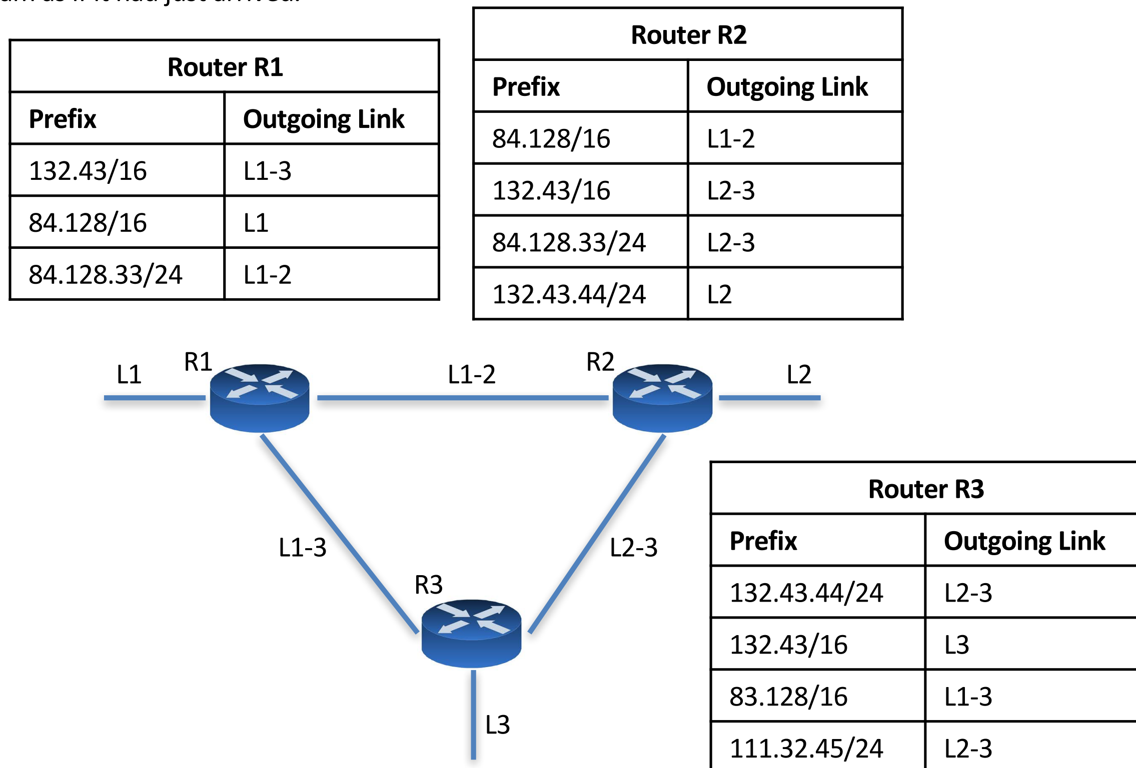
Job	Arrival Time	Duration	Turnaround Time	Response Time
A	0	25		
B	15	25		
C	20	5		
D	40	5		

Optional: show your work by drawing a Gantt chart or a time table

[20pts] 5. Network Routing

Depicted is a tiny part of the Internet with three routers R1, R2, and R3, and six bi-directional links, L1, L2, L3, L1-2, L2-3, L1-3. Also shown are the routing tables of each of the three routers. Each row shows an IP address prefix and the corresponding outgoing link. Recall that /16 stands for a netmask 255.255.0.0 (i.e., the only the first 16 bits are significant), and /24 stands for a netmask 255.255.255.0. Also recall that each IP datagram (aka packet) has a source address, a destination address, and a TTL (Time-To-Live, usually used as a maximum hop count). For simplicity, assume there are four types of IP datagram: TCP, UDP, TIME_EXCEEDED, and UNREACHABLE.

When a datagram arrives at a router, the router first checks if the destination address matches an entry in the routing table. If so and the TTL > 0, then the router decrements the TTL and forwards the datagram to the outgoing link in the entry. If not, there are two cases. If the datagram type is not UDP or TCP, then the router simply drops the datagram. Otherwise the router swaps the source and destination address in the datagram and sets the TTL to 100. It sets the type to UNREACHABLE if there was no match in the routing table, or to TIME_EXCEEDED otherwise. It then treats the datagram as if it had just arrived.



Router R1	
Prefix	**Outgoing Link**
132.43/16	L1-3
84.128/16	L1
84.128.33/24	L1-2

Router R2	
Prefix	**Outgoing Link**
84.128/16	L1-2
132.43/16	L2-3
84.128.33/24	L2-3
132.43.44/24	L2

Router R3	
Prefix	**Outgoing Link**
132.43.44/24	L2-3
132.43/16	L3
83.128/16	L1-3
111.32.45/24	L2-3

Below we will present several examples of IP datagrams arriving at a particular router. We will give you the router and IP datagram header upon entry at the router. You are to describe the path it will take, and the contents of the datagram "exit" header on the last link it travels in this diagram. In case a router drops the datagram, that would be the contents of the datagram just before it got dropped. For example, if router R1 receives a UDP datagram for destination 111.32.45.68, R1 turns the type into UNREACHABLE and sets the TTL to 100 (because there is no entry for the destination address in the routing table) and forwards the datagram to R3 over link L1-3 (because the destination address is now 132.43.22.33). Finally, R3 forwards the datagram to link L3:

Router R1	**Source**	**Destination**	**TTL**	**TYPE**
Entry:	**132.43.22.33**	**111.32.45.68**	**10**	**UDP**
Path:	R1 ➔ L1-3 ➔ R3 ➔L3			
Exit:	111.32.45.68	132.43.22.33	98	UNREACHABLE

NETID:________

[5 pts]

Router R2	Source	Destination	TTL	TYPE
Entry:	142.23.33.19	132.43.44.2	5	TCP
Path:				
Exit:				

[5 pts]

Router R1	Source	Destination	TTL	TYPE
Entry:	142.23.33.19	132.43.44.2	5	UDP
Path:				
Exit:				

[5 pts]

Router R3	Source	Destination	TTL	TYPE
Entry:	84.128.22.3	132.43.44.2	1	TCP
Path:				
Exit:				

[5 pts]

Router R1	Source	Destination	TTL	TYPE
Entry:	132.43.44.33	84.128.33.100	4	UDP
Path:				
Exit:				

[10pts] 6. A New Job at Knab Bank

You have just started working for the Knab Bank to maintain their core code. A now retired programmer has written some highly concurrent code that allows many operations on bank accounts to go on concurrently. The programmer took great care to make sure that it would never be possible to ``see inconsistent state'', for example, halfway through a transfer from one account to another when money has been withdrawn from the one account but not yet deposited into the other. All that seems to work great. Unfortunately, some operations like deposit and transfer sometimes seem to hang for ever. Your job is to find the bug and fix it. Below is an excerpt of the code:

```
class Account:                      # account object
    def __init__(self):
        self.lock = Lock()          # lock on the account
        self.balance = 0            # amount of money in the account

class Bank:
    def __init__(self):             # initialize instance variables
        self.lock = Lock()          # lock on the list of accounts
        self.accounts = []          # append-only list of accounts

    def newAccount(self):           # create an account and return new account number
        with self.lock:
            acct_number = len(self.accounts)
            self.accounts.append(Account())
            return acct_number

    def deposit(self, acct_number, amount):        # add money to account
        acct = self.accounts[acct_number]
        with acct.lock:
            acct.balance += amount

    # transfer money from one account to another. Return whether successful or not
    def transfer(self, acct_number_from, acct_number_to, amount):
        acct_from = self.accounts[acct_number_from]
        acct_to = self.accounts[acct_number_to]
        with acct_from.lock:
            with acct_to.lock:
                if acct_from.balance < amount:             # insufficient funds
                    sufficient_balance = False
                else:                                      # update both accounts
                    sufficient_balance = True
                    acct_from.balance -= amount
                    acct_to.balance += amount
        return sufficient_balance                          # return success status
```

Briefly describe the bug(s) and how to fix it (them) in the box below. Use plain English, not code.

NETID:________

[15pts] 7. All's Well That Ends Well

A slight variant of this question was asked on Prelim 2. Suppose you have a Terabyte partition on a disk. To be precise, the partition has 2^{40} bytes on it, subdivided into blocks of 8 Kbytes (8192 = 2^{13} bytes).

a) [1] How many blocks are on the partition?
(Write your answer in the format 2^{xxx}.)

Briefly explain (or provide the work for) your answer:

You want to put a Unix-like file system on the partition, with one superblock in position 0, followed by a sequence of blocks filled with i-nodes. Each i-node is 128 = 2^7 bytes. You want to have enough i-nodes to store 2^{20} (about a million) files.

b) [1] How many i-node blocks do you need to store all these i-nodes?
(Answer in 2^{xxx} format.)

Briefly explain (or provide the work for) your answer:

A block pointer identifies a block on the partition, and is 4 bytes long (enough to identify 2^{32} blocks). An "indirect block" (a block filled with block pointers) can have 8192 / 4 = 2048 (2^{11}) block pointers.

Suppose now that an i-node contains 13 block pointers. The first 10 point to the first 10 data blocks. The next three point to an indirect block, a double indirect block, and a triple indirect block. The maximum file size can be approximated by just the number of data blocks reachable from the triple indirect block pointer (the rest is negligible).

An i-node also contains a "last modified time" that is updated whenever a file is written, but in our case there are no other timestamps in an i-node.

c) [2] In theory, how much data (in bytes) could be accessed from the triple indirect block pointer in the i-node? For this question, assume the size of the disk is unbounded. (Answer in 2^{xxx} format.)

Briefly explain (or provide the work for) your answer:

Question 7. (cont'd)

d) [2] Assume now that the file system cache is empty except for the superblock. Assume the file with i-node #2015 has the string "Hello World" in it (that is, the file is just 11 bytes long). How many disk accesses would be necessary to read the contents of this file, given that you (and the kernel) know the i-node number?

Briefly explain your answer:

e) [3] In reality this same file's i-node number has to be retrieved first. Suppose the name of the file is /etc/test.txt. Assume that the contents of each directory fits in a single block. The root directory / is described in i-node #2 by convention. Assume /etc is in i-node #5 (you know this, but the kernel doesn't). Again, assuming only the superblock is in the cache and a cache large enough so the same block never has to be read more than once, how many *disk* accesses are required to read the file?

Briefly explain your answer:

Question 7. (cont'd)

NETID:________

f) [3] File /etc/shakespeare.txt (which you know to be in i-node #7, but the kernel doesn't) contains the complete works of Shakespeare (2^{22} bytes or about 4 Megabytes). Assuming only the superblock is in the cache, how many disk accesses are required to retrieve the whole thing?

Briefly explain your answer:

g) [3] Suppose somebody wants to add the text "All's Well That Ends Well." to the end of the complete works of Shakespeare (the new text will be contained in a new data block at the very end). Suppose that the file system has only the superblock in its cache and can allocate free blocks without going to the disk. How many disk reads and how many disk writes are necessary (assuming a "write-through" cache? (Assume that among the i-nodes only i-node #7 has to be updated for this operation.)

Briefly explain your answer:

SCRATCH PAPER (WILL NOT BE SCANNED NOR CONSIDERED FOR GRADING)

SCRATCH PAPER (WILL NOT BE SCANNED NOR CONSIDERED FOR GRADING)

SCRATCH PAPER (WILL NOT BE SCANNED NOR CONSIDERED FOR GRADING)