Name: | **NetID:**

CS 4410 Operating Systems Prelim II, Fall 2019
Profs. Van Renesse and Schneider

KEEP CLOSED UNTIL START OF EXAM IS ANNOUNCED

- Turn off and stow away all electronic devices. You may not use them for any reason for the duration of the exam. Do not take them with you if you leave the room temporarily.
- This is a closed book and closed notes examination.

- **You have 110 minutes**. Do not leave the room, either during the exam or right afterward, unless you have received explicit permission. If you need to leave the room during the exam you may need an escort.

- You get 1 point if you fill in your name and NETID on this page. You get 1 point if you write your NETID in the top right corner on the remaining sheets. *DO IT FIRST.* You get 0 points for the whole exam if you do neither.

- Write your solutions within the provided boxes. Write clearly. There are extra sheets if you don't have enough space in the provided boxes. Indicate clearly in the answer box if you have used this additional space.

- **For Coding Questions:** You may use pseudo-code. *Descriptions written mostly in English will get no credit.* In addition to correctness, elegance and clarity count.

- To receive partial credit you must show your work. State any assumptions you make.

- You can use the backs of these pages as scratch paper. *THEY WILL NOT BE SCANNED.*
- Turn in your work with the pages in order, ready to be scanned.

Question	Points Possible
0: Name & NetID	2
1: LRU-k Page Replacement	15
2: BitByBit Virtual Memory	18
3: Wee-FS File System	20
4: DisksRUs Strike Again	18
5: Priority Scheduling	12
6: Multiphores Reprise	15
Total	**100**

NetID:

Use this box if you do not have enough space in a provided answer box. Please indicate in the answer box that you have used this space, and in this space indicate what question(s) you are answering. DO NOT REMOVE THIS PAGE!

NetID:

1. [15 points] You have an operating system with paged processes, but processes do not share pages. Given a set of *m* frames, called an "*m*-cache", the LRU page replacement algorithm maintains an initially empty cache of the *m* most recently used pages.

In the *LRU-k page replacement algorithm* the *m*-cache maintains the *m* most recently used pages, but the *m*-cache contains at most *k* pages for any process. So, when a process *P* references a page *p* that is not currently stored in the *m*-cache, then

1. if *P* already has pages occupying *k* frames, then the least recently used of *P*'s pages is evicted and replaced with *p*.
2. if *P* has pages occupying fewer than *k* frames and some frames are currently unused, then one of those unused frames is used to store *p*.
3. if *P* has pages occupying fewer than *k* frames and all m frames are used, then the least recently page among all pages stored in the *m*-cache is evicted and replaced with *p*.

Consider a reference string, written as a sequence of process:page pairs:

P:1	P:2	Q:1	Q:2	P:3	Q:1	P:2

a) Let *k* = 2. Fill in the table below for *m* = 3 and *m* = 4. For each, list pages in the m-cache, ordered by recency of reference, with the most recently referenced page first. (Some cells are already filled in by way of example.) Also indicate whether the reference is a page fault (i.e., a cache miss) or not.

Reference string ↓

	m = 3		*m* = 4	
ref	*m*-cache contents after reference	fault?	*m*-cache contents after reference	fault?
P:1	**P:1**	**Y**	**P:1**	**Y**
P:2	**P:2 P:1**	**Y**	**P:2 P:1**	**Y**
Q:1	**Q:1 P:2 P:1**	**Y**	**Q:1 P:2 P:1**	**Y**
Q:2	Q:2 Q:1 P:2	Y	**Q:2 Q:1 P:2 P:1**	**Y**
P:3	P:3 Q:2 Q:1	Y	P:3 Q:2 Q:1 P:2	Y
Q:1	Q:1 P:3 Q:2	N	Q:1 P:3 Q:2 P:2	N
P:2	P:2 Q:1 P:3	Y	P:2 Q:1 P:3 Q:2	N

b) Now consider extensions to the same reference string. Extend the reference string with references that each causes a cache miss (page fault) in the 3-cache but a cache hit in the 4-cache. Provide up to four such references (leave empty if you think there are no such references).

P:1	P:2	Q:1	Q:2	P:3	Q:1	P:2	Q:2	P:3	Q:1	P:2

Scratch area (not graded)

NetID:

2. [18 points] The BitByBit Computer Company (BCC) has designed a bit-addressable computer. The BCC CPU is a 48-bit computer and thus it can address 2^{48} bits. The BCC CPU supports paging, and each page has 2^{18} bits.

(a) What is the total number of pages in the virtual address space of a process running on a BCC computer? Write your answer in 2^x notation.

$2^{48} / 2^{18} = 2^{30}$

(b) A BCC Page Table Entry (PTE) has 64 bits and consists of

- A valid bit, three RWX bits, a use (reference) bit, and a dirty bit
- A 58 bit frame number

If the BCC CPU were to use a 1-level page table for a process, how large would this page table have to be. Specify both in 2^x notation and in kilobits, megabits, etc. For example, your answer might look like 2^{23} = 8 megabits. If you insist you can use International Electrotechnical Commission notation (8 mebibits in this case).

$2^{30} \times 2^6 = 2^{36}$ bits = 64 gigabits (gibibits)

(c) The hardware engineers at BCC decided that it would be more convenient to have each page table fit in frame. How many PTEs fit in a frame?

$2^{18} / 2^6 = 2^{12} = 4096$ PTEs

(d) How many levels of page table are necessary to cover a 48-bit virtual address space?

3 (2 levels is not enough only gets to 2^{24} pages – need 2^{30} pages)

(e) Given your answer to (d), how many frames need to be allocated at minimum so that addresses 0x000,000,001,000 and 0xFFF,FFF,FFF,FFF are both in memory and mapped?

7 (2 data frames, 1 root page table, and 2x2 internal page tables)

NetID:

3. [20 points] Consider a 1 TB (terabyte) disk partition consisting of 2^{28} blocks of 4KB (2^{12} bytes) containing an FFS file system. Block 0 is used for the superblock. Suppose an i-node takes up 256 bytes.

(a) How many i-nodes fit in a block?

$2^{12} / 2^8 = 2^4 = 16$

(b) If the file system has to support 2^{20} files and directories, how many blocks for i-nodes are needed? (2^x notation for the answer is good enough.)

$(2^{20} \times 2^8) / 2^{12} = 2^{16}$

Many file systems have a large number of small files (which may include directories), often smaller than a single block. You are tasked with designing a variant of an FFS file system that is optimized for storing such small files. You quickly realize that it is inefficient to store the i-node of a small file separate from the data of the file.

(c) Briefly mention two sources of inefficiency that arise when storing an i-node and its first data block in separate disk blocks.

1) Wasted space because one block is enough to store both instead of two
2) Wasted time because two disk accesses are needed to read the file

NetID:

The idea emerges to store the i-node of a small file in the same block as the data in that file. To support small files, your "Wee-FFS" file system has support for "wee-files", which are special blocks that store a file with up to 4096 – 256 data bytes as well as storing the i-node. The data is stored at the start of the block, and the i-node is stored in the last 256 bytes. To be consistent with other i-nodes, the i-node number for a wee-file in block *b* can be computed by the formula *b* x *N* – 1, where *b* is the block number and *N* is the number of i-nodes per block (answer to (a)). Inversely, given an i-node number *i*, the i-node is in block 1 + floor(*i* x 256 / 4096), while *i* x 256 mod 4096 is the offset of the i-node in the block.

(d) What rationale explains the designer's choice of putting the i-node at the end of a block rather than at the beginning?

In FFS, the data in the first block of a file is always starts at the beginning of the block. By putting the i-node at the end of the block, no "if" statement is needed in Wee-FFS to retrieve data from the block in case of a wee-file, reducing complexity of the file system code.

(e) You want to read the password file in /etc/passwd stored by Wee-FFS. For compatibility, the root directory is stored in i-node 2 and therefore is not stored as a wee-file, even though it easily fits in a block. Directory /etc is stored as a wee-file. Also, the password file is small enough to be stored as a wee-file. Assume only the superblock is in the cache. How many blocks must be read to retrieve /etc/passwd?

4 blocks

If you like, you can briefly clarify your answer for partial credit in case the number above is wrong.

1) read block 1 to retrieve i-node 2
2) read the block of the root directory to find the i-node number of /etc
3) read directory /etc (i-node and data stored in the same block)
4) read /etc/password (again, i-node and data stored in the same block)

(f) Still assuming only the superblock is in the cache, how many blocks would have had to be read in the worst case to retrieve the same password file in a standard FFS file system?

6 blocks

If you like, you can briefly clarify your answer for partial credit in case the number above is wrong.

Two more to retrieve the i-nodes of /etc and /etc/passwd.

(g) A file system administrator who keeps careful track of file system inconsistencies after crashes notices that after deploying Wee-FFS the average number of inconsistencies is different from before, when an ordinary FFS file system was used. Describe briefly if you think Wee-FFS will be observed to have fewer or more inconsistencies as a result of crashes.

Wee-FFS likely has fewer inconsistencies because i-nodes and data are written in one disk operation instead of two. So you can't have wee-files that point to a free block or duplicate data blocks, for example.

NetID:

4. [18 points] The engineers at DisksRus are considering improvements to their 1HD basic disk drive product. The parameters of 1HD are:

- 1 surface (on 1 platter)
- 1 read/write head
- 1 msec per rotation
- seek time: 2 msec/track traversed
- 200 tracks
- 100 blocks/track
- block size 512 bytes

The product data sheet for 1HD defines “disk request processing time" as the sum of:

(i) “seek time” (time delay to position the disk arm over the correct track)

(ii) “rotational delay” (time delay for the disk surface to rotate so the correct disk block is under the read/write head)

(iii) “xfer time” (time delay for the read/write head to transfer the contents from/to the block, i.e, the time taken for the head to pass over the block).

(a) If disk requests are randomly and uniformly distributed over the set of blocks, then what is:

the minimum disk request processing time for reading a block?	1/100 = 0.01 msec
the maximum disk request processing time for reading a block?	2x199 + 1 + 0.01 = 399.01 msec

Briefly explain your answers here

NetID:

The 2HD disk drive product improves on 1HD by adding a second read/write head on an independent disk arm. The drive electronics allows I/O requests to be routed to one or the other of the disk arms according to some internal programmable logic. And the drive electronics has the capacity to perform in parallel a transfer operation with each read/write head. The intention is that each block be stored only once on a surface (unlike the scheme we explored in your homework).

All of the other parameters for 2HD are the same as for 1HD.

Assume the requests are randomly and uniformly distributed over the set of blocks that the disk stores.

(b) If engineers wanted to reduce the maximum rotational delay for reading a block, what's the best that they could expect to achieve?

½ msec (both heads on the same track 180 degrees apart)

(c) If engineers wanted to reduce the maximum seek time for reading a block, what's the best that they could expect to achieve?

198 msec (one head for tracks 0-99, the other for tracks 100-199)

Briefly explain your answers to (b) and (c) here for partial credit in case you made a mistake in either.

NetID:

5. [12 points] Given is an operating system scheduler that implements priority scheduling without preemption. The scheduler uses a pre-defined *prio expression* to calculate a priority of each task, where terms that may be referenced in a prio expression for a given task T include:

- DL[T]: time by which T needs to be completed (i.e., the deadline of T)
- SRV[T]: the service (aka execution) time for completion of task T
- ARV[T]: the arrival time for task T
- PAY[T]: the amount the task is worth

Tasks having smaller values of a prio expression are executed before tasks having larger values.

As an example, a prio expression to cause the scheduler to implement "highest priority to least valuable task" would be: PAY[T]

(a) Give a prio expression for each of the following scheduling policies:

	prio expression
First Come First Served (aka FIFO)	ARV[T]
Shortest Job First	SRV[T]
Earliest Deadline First	DL[T]

(b) A set S of tasks is *schedulable* if there exists a schedule, where each task T is started at or after its arrival time ARV[T] and finishes at or before its deadline DL[T]. Give a 3-task counterexample to the claim that if every task T in a set satisfies

ARV[T] + SRV[T] <= DL[T]

then the set be schedulable?

	ARV[T]	SRV[T]	DL[T]
Task 1	0	1	1
Task 2	0	1	1
Task 3	0	1	1

NetID:

(c) Suppose that employing an *earliest deadline first (EDF)* policy does not cause all tasks in some set to meet their deadlines. Can we conclude that no scheduling policy will permit all of those deadlines to be met? Explain why by either (i) giving an informal proof or (ii) giving a counter-example comprising a set of tasks (with arrival time, service time, and deadline), along with a schedule that is not EDF but meets all deadlines, and a schedule that is EDF but does not meet all deadlines.

Counter-example: two tasks:

As has been our practice we assume all tasks are available for scheduling at time 0

T1: arr = 2, svc = 1, dl = 3
T2: arr = 0, svc = 2, dl = 4

EDF: T1 then T2
Non-EDF: T2 then T1

NetID:

6. [15 points] Recall, a general semaphore (or counting semaphore) s is a non-negative integer that supports two operations P(s) and V(s) whose behavior can be described as follows (where 〈 … 〉 denotes an operation that is executed as an indivisible action):

P(s): 〈 await s > 0 then s := s – 1 〉
V(s): 〈 s := s + 1 〉

Other operations could have been defined, though, as we now explore.

Define a *multiphore* mp(integer initial_value) to be a non-negative integer that supports two operations Pmult(mp, amt) and Vmult(mp, amt) whose behavior is described as follows:

Definition of multiphore operations:

Pmult(mp, amt): 〈 await mp – amt >= 0 then mp := mp – amt 〉
Vmult(mp, amt): 〈 mp := mp + amt 〉

A multiphore must be initialized to some non-negative integer value.

Suppose a system supports monitors that perform the Mesa operations on conditions: c.wait, c.notify, and c.notifyAll.

Complete the code on the next page for monitor mp that provides the indicated operations for multiphores. Thus, invoking "call mp.Pmult(amt)" should have the same effect as Pmult(mp, amt), and "call mp.Vmult(amt)" should have the same effect as Vmult(mp, amt), as defined above in **Definition of multiphore operations**.

Scratch area (not graded)

NetID:

mp: **monitor**(initial_value: **integer**)

monitor variables declared below

```
var val: integer initially initial_value
    needVs: condition        // condition is val >= v
```

Pmult(v: **integer**) **operation**

Pmult code below

```
while val<v do needVs.wait end
val := val - v
```

end Pmult

Vmult(v: **integer**) **operation**

Vmult code below

```
val := val + v;
needVs.notifyAll
```

end Vmult

end mp

*(**bold face** used for keywords)*