CS-537: Final (Spring 2016)

"More" Fun With Operating Systems

Please Read All Questions Carefully!

There are fourteen (14) total numbered pages, with fourteen (9) questions.

Please put your FULL NAME (mandatory) on THIS page only.

Name: SOLUTIONS

Grading Page

	Points	Total Possible
Q1		10
Q2		10
Q3		20
Q4		10
Q5		10
Q6		20
Q7		10
Q8		10
Q9		10
Total		110

Directions

"Less is more." –Ludwig Mies van der Rohe

This question deals with the topic of **more.** You always want more, don't you? Well, now is your one last chance to explore the concept of "more" in operating systems, on this very exam! So you're lucky in this way.

Specifically, each question will deal with adding more of something to a system and figuring out the repercussions. For example, we might need to figure out how to handle really large files, or what happens when you have a lot of memory, or how to build really big RAID arrays, or what happens when an operation in a network file system takes a really long time.

Please **read each question carefully.**

Exam-taking strategy should be: **easiest-problem-first.** This scheduling discipline will ensure you finish as much of the exam as possible, and also builds your confidence. Don't get stuck working on one hard problem.

Do note that **some problems are worth more than others.** Specifically, two longer problems about SSDs and RAIDs are worth 20, while other problems are each worth 10. The easy way to remember this is that **each page is worth 10 points** (and two questions are two-pages long).

Good luck! And remember, as old Ludwig told us above, more isn't always better, especially when it comes to answers. Keep them short and sweet!

1. Berkeley Fast File System

The Berkeley Fast File System (FFS) was one of the first file systems that treated the disk like a disk and thus improved performance through spatial locality. It also has a "large file exception" and thus is a perfect candidate for a question on this exam about "more".

a) Before delving into this exception, let's first do some basics. What on-disk structures does FFS use to track **allocation** of inodes and data blocks?

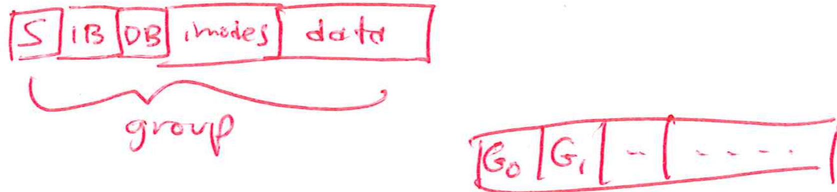
Bitmaps

b) Why was this an improvement over the classic Unix file system?

old used linked list

⇒ Fragmentation

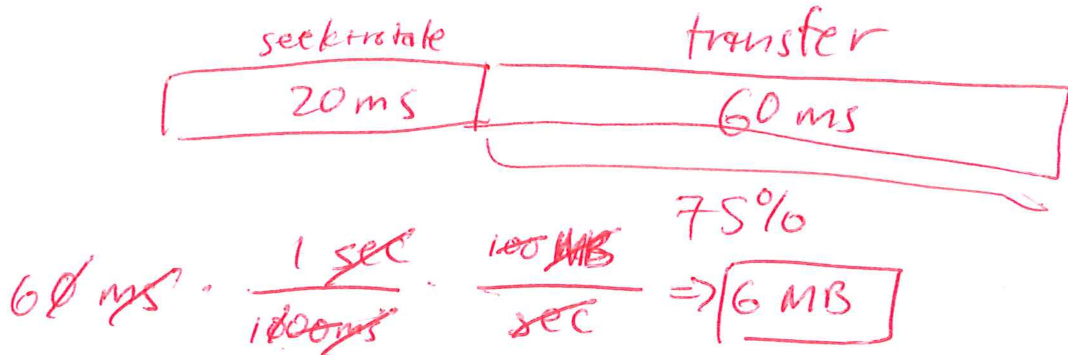
c) Draw a picture of the file system layout of FFS. How is it different than other simple file systems we've studied, such as the Very Simple File System (VSFS)?



d) FFS tries to spread large files out across the disk, by splitting them into chunks and putting each chunk in a different part of the disk. Why does FFS do this for large files?

try to avoid filling one group

e) Assuming that a disk transfers at a peak rate of 100 MB/s, and that a seek and rotation take, on average, 20 milliseconds. How big should each chunk of a large file be, so as to achieve 75% of peak transfer rate for large files when they are accessed sequentially?



2. Files That Are Large

Most file systems support pretty large files. In this question, we'll see how big of a file various file formats can support.

Assume, for all questions below, that file-system blocks are **4KB**.

a) Assume you have a really simple file system, **directfs**, where each **inode** only has 10 direct pointers, each of which can point to a single file block. Direct pointers are 32 bits in size (4 bytes). **What is the maximum file size for directfs?**

10 * 4KB => 40 KB

b) Assume a file system, called **extentfs**, has a construct called an **extent**. Extents have a pointer (base address) and a length (in blocks). Assume the length field is 8 bits (1 byte). Assuming that an inode has exactly **one** extent. **What is the maximum file size for extentfs?**

2^8 length => 256 * 4 KB blocks

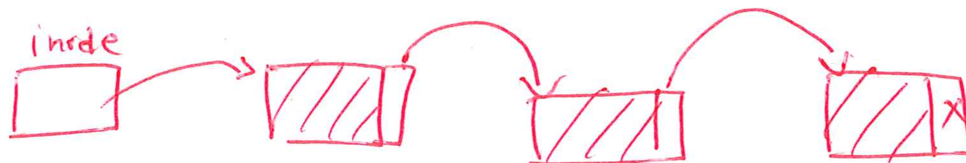
1 MB

c) A new file system uses direct pointers but also adds indirect pointers and double-indirect pointers; we call the **indirectfs**. Specifically, an inode within indirectfs has 1 direct pointer, 1 indirect pointer, and 1 double-indirect pointer field. Pointers, as before, are 4 bytes in size. **What is the maximum file size for indirectfs?** (it's OK just to show an equation here instead of the actual numeric result)

$$[(1024 \times 1024) + (1024) + 1] \times 4KB$$

~ 4 GB

d) A compact file system, called (can you guess it?) **compactfs**, tries to save as much space as possible within the inode. Thus, to point to files, it stores only a single pointer to the first block of the file. However, blocks within compactfs store 4KB of user data and a **next** field (much like a linked list), and thus can point to a subsequent block (or to NULL, indicating there is no more data). First, **draw a picture of an inode and a file that is 12KB in size.**



e) Now, a final question: what is the maximum file size for compactfs?

$2^{32} \cdot 4KB$

3. Gigantic SSDs

Flash-based SSDs are increasingly replacing hard drives, especially when performance is important. In this question, we explore Flash-based SSDs, including a focus on what happens to them when they become large.

SSDs commonly contain a Flash Translation Layer (FTL), which maps each file system block to an underlying flash page. Assume, for this question, that **file system block size** is 4KB, and that **flash page size** is also 4KB. Assume further that the **flash block size** is 16 KB.

a) Assuming a basic page-mapped, log-structured FTL, and that the flash is initially in an INVALID state (i.e., each block needs to be erased before any page is programmed), what **sequence of flash operations** (e.g., read, erase, program) will take place when the following writes are performed by a file system above:

```
write(fsblock=1000, data='a'), write(100, 'b'), write(500,'c'),
write(1, 'd'), write(2, 'e'), write(300, 'f')?
```

erase (block=0)
prog (page=0, data="a")
prog (1, "b")
prog (2, "c")
prog (3, "d")
erase (1)
prog(4, "e")
prog(5, "f")

2

b) When this sequence of writes is complete, what will **the contents of the FTL** be? (what is the map of file-system blocks to flash pages?)

1000 → 0
100 → 1
500 → 2
1 → 3
2 → 4
300 → 5

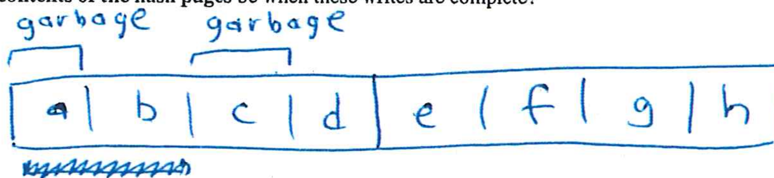
2

c) Now assume there are a few more writes: `write(500, 'g'), write(1000, 'h')`. **What will the contents of the FTL be?**

500 → 6
1000 → 7
] otherwise same as (b)

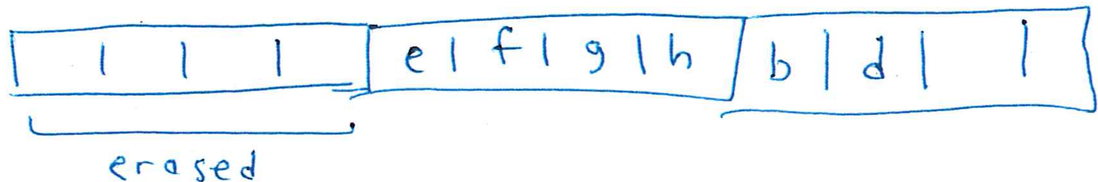
2

d) **What will the contents of the flash pages** be when these writes are complete?



2

e) Assume a garbage collector runs to clean up any dead blocks and make space available (by reading in entire blocks, determining what is live, and writing live blocks to the end of the log). **What will the final state of the flash pages be when garbage collection is finished?**



2

f) One problem with larger SSDs is that too much space is taken up by the FTL. Assuming 4 KB flash pages, and a 4 TB SSD (from the future!), **how many entries will the FTL have?**

$$\frac{4\ TB}{4\ KB} = 1\ G \text{ entries } (\sim 1 \text{ Billion})$$

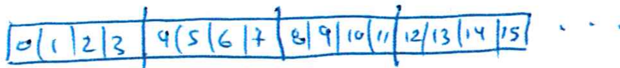
g) One method of dealing with FTLs that are too large is to swap pieces of the FTL and thus keep only pieces of it in the SSD's memory. Let's assume our FTL can only contain 16 4-KB pieces of the FTL at any given time in memory; each mapping entry uses 4 bytes. When there is memory pressure, assuming LRU replacement is used to evict pieces of the FTL from SSD memory.

Assume we have a workload where we **read through a large file sequentially; how much worse will performance be because of our swap-based FTL?** (be as quantitative as you can be)

[1024 entries for 4 KB block of FTL]

Assume contiguous placement of file...
Each 1024 reads requires extra FTL read

⇒ ~0.1%



h) Now assume we have a workload where we **write out a large file sequentially. How much worse does our swap-based FTL perform** under this workload?

Same as (g)

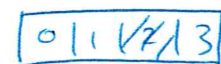
i) Now assume we have a workload where we perform a large number of **random reads** over a very large file. **How much worse does our swap-based FTL perform** under this workload?

Each data read requires FTL read

⇒ 2x

j) One other method to reduce FTL size is to try to use a single pointer per **flash block** instead of per flash page. Unfortunately, this type of approach is challenging, particularly when a smaller (page-sized) write occurs. **What performance problem arises for smaller writes** in a block-based FTL?

Must do copy-on-write (read/modify/write)



to update 2

→ read 0, 1, 3
→ erase block
→ write 0, 1, 2', 3

4. Journaling And Large Writes

Journaling, or write-ahead logging, is a technique that can be used to ensure that a file system recovers its metadata to a consistent state after a crash. In this question, we'll explore what happens when we use this technique for LARGE writes.

a) One type of journaling is called "data journaling", because all file system metadata *and* data is written to the log before being updated in place. Describe the performance of such a journaling file system, as compared to a file system that only logs metadata, for **sequential write workloads**. What throughput can you expect? (roughly)

writes all data twice
For sequential writes, most traffic is data
=> 2x slower

b) Usually write operations that occur with data journaling are committed atomically, meaning that they either happen in their entirety, or don't get committed at all (due to a crash). How does data journaling guarantee that a set of updates are updated atomically? (describe)

write everything carefully to journal (first)
Only when all is committed there
do we update FS proper

c) Now let's think about what happens for large transactions. Let's say a very large write takes place (e.g., a user calls write() with an incredibly large 10-GB buffer). Why is this problematic for journaling? How should the file system handle this write? What guarantee can it make about the atomicity of the write's completion?

→ Can't fit 10 GB write into journal
→ Must break update into N smaller (consistent) updates
→ Thus, only each smaller write is atomic

d) Because of some of the problems of data journaling, people often use metadata-only journaling. Describe the basic protocol, including which writes must happen before others.

Data is checkpointed (written) directly
TBegin + Metadata → Journal
after: TEND → Journal
after: Metadata checkpointed

e) Do writes cause any different problems for metadata-only journaling? For example, even assuming relatively small writes, do they commit atomically to disk? (hint: think about overwrites)

Because overwrites occur in place,
may start to do update and crash
Thus, overwrites are not atomic
Contrast: Append works as expected
(∴ is atomic)

5. Networking And File Systems

In this question, we'll think a little bit about Sun's Network File System (NFS) protocol, and how it is affected when network delays are large. Specifically, sometimes messages take a LONG TIME to get from one machine to another, and this can affect how a distributed system behaves.

a) The basic protocol works by sending a request to the server, and waiting for a reply. What happens, on the client, if the reply takes a LONG TIME to complete?

client times out
sends request again (retry)

b) What if replies always took a LONG TIME to complete? What might you change about the protocol or implementation?

change timer to wait longer
(avoid always timing out)

c) The server handles requests and sends replies. What happens, on the server, if the reply it sends to the client takes a LONG TIME to complete?

Nothing
(Eventually client times out and retries)

d) The server always forces updates to persistent storage before replying. Why does the server do this, even though it makes each reply take longer to complete?

if server ack'd request
and crashed, update
could be lost

e) The client waits for a long time before sending a file write to the server. Why is it OK for the client to wait before writing to the server?

OK to buffer writes (in general)
(However does cause some issues:
visibility by others)

buffering
of writes
at client
does NOT
affect correctness
of protocol ⟹ only visibility, performance

6. RAIDS That Are Big

RAID storage systems are commonly used in very large disk arrays (and even, in some modern settings, for large collections of SSDs). Let's examine the very basics of RAID before delving into some new techniques needed for especially large arrays.

a) RAID levels 4 and 5, as discussed in class, use "even parity" to store a bit of redundant information for each stripe of data. For each of the following sets of bits, calculate the parity bit:

(a) Bits: 0 0 1 0 Parity? 1

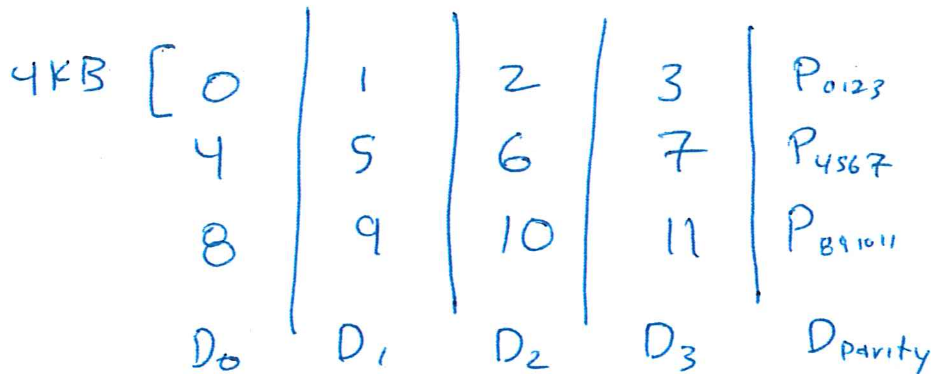
(b) Bits: 1 0 1 0 Parity? 0

(c) Bits: 0 1 0 1 Parity? 0

(d) Bits: 1 1 1 1 Parity? 0

2

b) RAID level 4 uses a parity disk to protect data. Draw a picture of a RAID-4 array, with a chunk size of 4KB, including 4 data disks and 1 parity disk.



2

c) In RAID 4, a **small write** problem exists, which occurs when we wish to update a single block of data. When updating a single block in a 5-disk RAID-4 array (4 data disks, 1 parity disk), what blocks must be read and written, and in what order? (assume you do this in the I/O minimal way, e.g., with the least amount of I/O that is possible)

2 reads Read Target data + corresponding Parity

before diff new, old data to compute new parity from old parity

2 writes Write Target + New Parity

3 −1 no parallel; extra R/W −1, −2

d) RAID-5 is quite similar but rotates the parity across disks. What is the **primary** reason that RAID-5 does this? Does it solve the small-write problem above?

Avoids parity disk bottleneck

(but still does small writes in same manner)

3

e) Larger arrays have to tolerate more than a single failure (which is what RAID-4 can tolerate). Assume we wish to build an array to tolerate 2 failures, using a similar parity-based scheme. We call this scheme RAID-DP, for RAID Double Parity. This approach includes an additional disk for what is called the **diagonal parity**. We illustrate the approach in the picture below, with 4 data disks, 1 traditional RAID-4 parity (which is the computed parity of the data disks, just as in RAID-4), and 1 diagonal parity disk.

```
Data    Data    Data    Data    Row     Diag
 0       1       2       3     Parity Parity
--------------------------------------------
[0]      1       2       3       4      [0]
 1       2       3       4      [0]      1
 2       3       4      [0]      1       2
 3       4      [0]      1       2       3
```

In the diagram, each block is labeled to indicate which **diagonal parity stripe** it is a part of. For example, the first block on data disk 0, the fourth block on data disk 2, the third block on data disk 3, the second block on the row parity disk, and the first block on the diagonal parity all form a diagonal stripe (as marked with square brackets); the diagonal parity block is just the computed even parity over all of those blocks (on disks 0, 2, 3, and row-parity) in the diagonal stripe. Similar diagonal stripes exist for 1, 2, and 3 (note there is no diagonal parity for diagonal 4; you can ignore this as it is not needed).

To understand RAID-DP better, we'll have you compute the parity for the diagonal-parity disk assuming the bits below (we use bits instead of entire blocks for simplicity, naturally). Thus, fill in a bit for each of the diagonal parity spots below, using the figure above as needed.

Hint: the first diagonal parity is just the even parity of the first bit on data disk 0, the fourth bit on data disk 2, the third bit on data disk 3, and the second bit on the row-parity disk, as the diagram above indicates.

```
Data    Data    Data    Data    Row     Diag
 0       1       2       3     Parity Parity
--------------------------------------------
 0       1       0       0       1      ____    <--- FILL THIS IN
 0       1       1       0       0      ____    <--- FILL THIS IN
 1       0       0       0       1      ____    <--- FILL THIS IN
 0       0       0       1       1      ____    <--- FILL THIS IN
```

f) The hard part with RAID-DP, naturally, is reconstruction (i.e., figuring out the missing bits when 2 disks have failed. It is possible, however, by using a step by step method in which you first reconstruct some blocks from the diagonal parity, and then the row parity, and so forth until the complete reconstruction has taken place.

In this final question, data disks 1 and 3 have failed. Your task is to reconstruct them, using the information on the working data disks, the row parity (which is computed as the even parity of the data disks), and the diagonal parity, which is computed as described above.

```
Data    Data    Data    Data    Row     Diag
 0       1       2       3     Parity Parity
--------------------------------------------
 0      ____     1      ____     0       1      <--- FILL THESE TWO BLANKS ON DISKS 1 and 3
 1      ____     0      ____     0       0      <--- FILL THESE TWO BLANKS ON DISKS 1 and 3
 0      ____     1      ____     1       0      <--- FILL THESE TWO BLANKS ON DISKS 1 and 3
 1      ____     0      ____     1       1      <--- FILL THESE TWO BLANKS ON DISKS 1 and 3
```

7. Schedules Anyone

Disk scheduling can become quite interesting, especially as the disk queue gets larger and more options become available. In this question, we explore disk scheduling for a simplified disk we have created known as a **matrix disk**.

The matrix disk looks like this, organizing data into rows and **columns**:

```
[0]  1   2   3   4   5   6   7
 8   9  10  11  12  13  14  15
16  17  18  19  20  21  22  23
...
```

The **columns** all move to the left every R time steps, with wrap around. For example, if $R = 1$, every time unit the columns move left; after 2 time units, our matrix disk would look like this:

```
[2]  3   4   5   6   7   0   1
10  11  12  13  14  15   8   9
18  19  20  21  22  23  16  17
...
```

The square brackets represent the disk **read-write head.** It can read or write the block it is positioned over (block=0 in the first diagram above; block=2 in the second). The disk reads or writes a block in R time steps, matching the column-shifting speed described above.

Thus, in the current configuration, just by waiting, our matrix disk can read blocks 0 ... 7. But what about the other blocks? To read/write other blocks, the matrix disk must engage in a **seek**, to move to a different row. Seeking to another row takes S time units.

Assume that $S = 1$. Assuming that $R = 1$ (as above), and that we started seeking when block=2 was under the read-write head (as shown in the second diagram), a short seek down to the next row would end up with:

```
3      4   5   6   7   0   1   2
[11]  12  13  14  15   8   9  10
19    20  21  22  23  16  17  18
...
```

The rest of this question is about the performance of various schedulers on the matrix disk, especially with a LARGE number of requests. OK, LARGE here may mean "more than one."

IMPORTANT: Assume the following starting position for each of the following questions.

```
[0]  1   2   3   4   5   6   7
 8   9  10  11  12  13  14  15
16  17  18  19  20  21  22  23
...
```

OK

5 (6) a) Assume $R = 1$ and $S = 1$. What is the minimum time it will take to read block 21? [2 seek + 3 wait + 1 read

20, 21 (22) b) Assume $R = 2$ and $S = 1$. How long will it take to read blocks 6 and 18, if we read 6 before 18? [12 wait + 2 read + 2 seek + 4 wait + 2 read

10, 12, (14) c) Assume $R = 2$ and $S = 1$. How long will it take to read blocks 6 and 18, if we read 18 before 6? [2 seek + 2 wait + 2 read 2 seek + 4 wait + 2 read

d) Assuming $R = 1$, for what values of S should the scheduler read 18 before 6? $S \leq 1$

e) Assume a scheduler is being built, called **shortest-seek-time-first (SSTF)**. You are to write a function that determines how long it will take to seek given the current row position (**curr_row**) and the target block **block**, assuming the 8-column matrix disk as above, and that you know the value of S. We've filled in some of the code to help you out:

```
int how_long(block, curr_row, S) {
  dest_row = block / 8;                          // how to compute destination row?

  seek_dist = abs(dest-row - curr-row);          // function of dest_row and curr_row and ...

  seek_time = seek-dist * S;                     // easy to compute ... yes?
  return seek_time;
}
```

8. Threads Are Part Of This Exam

So, you have threads. LOTS of threads. In this question, you use LOTS of THREADS to solve some problem in parallel on a computer with LOTS of processors. So that kind of fits the theme, right?

Here is some code:

```
void *func(void *arg) {
     int arg_int = (int) arg;   // treats argument as integer
     int x = ...;               // computes something based on 'arg_int', puts result in 'x'
     return (void *) x;         // returns integer x (pretending it is a "void *")
}
```

a) First, write some code to directly call `func()`, passing it as an argument the integer value 100, and putting its return value in the integer named `foo`. Make sure to cast things properly to enable a warning-free compilation.

```
int foo = (int) func ((void *) 100);
```

Now recall how to use `pthread_create`. This is what the prototype looks like for that useful thread-creating function:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *restrict arg);
```

You can assume, here, that the `attr` (attributes) are set to NULL.

b) Let's say you wish to call `pthread_create()` to create exactly one thread, which calls `func()` above and passes it the argument integer value 100. What does that code look like? Make sure to declare any thing that you need to for the code to compile, as well as to cast properly.

```
pthread_t p;
int rc = pthread_create(&p, NULL, func, (void*) 100);
```

c) Now let's try to wait for this thread to complete, and to get its return value. Use `pthread_join` to do so; here is its prototype:

```
int pthread_join(pthread_t thread, void **value_ptr);
```

For this code snippet, call join to wait for the single thread to complete, and then print out the return value (which is an integer).

```
int result;
pthread_join(p, (void **) &result);
printf("%d\n", result);
```

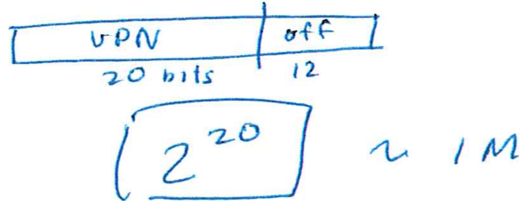
e) Finally, let's create a large number of threads and add up all of the results. Specifically, write code to create 100 threads, pass each of them a different argument, from 1 through 100 (inclusive), and then sum up all of the results the threads return; finally, print out the sum.

```
pthread_t p[100];    int sum = 0;
for (int i = 0; i < 100; i++)
    pthread_create(&p[i], NULL, func, (void*) i+1);
for (int i = 0; i < 100; i++) {
    int result;
    pthread_join(p[i], (void **) &result);
    sum += result;
}
```

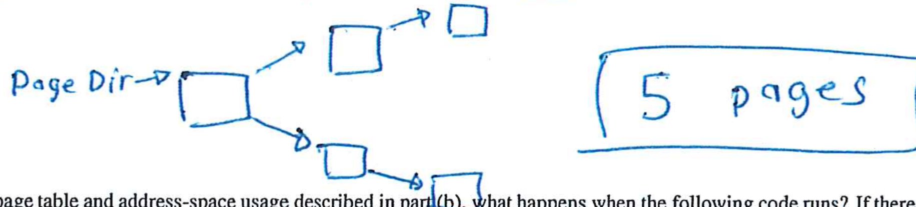
9. Virtual Memory Gets Big Too

Ah, you thought you could forget about virtual memory. But not yet! Here, we assume your memory has more capacity, and thus can handle this abuse. Specifically, we examine here what happens when a system uses really large page sizes. However, we first review a few concepts; hopefully they are familiar!

a) Assume a system has 32-bit virtual addresses and small, 4-KB pages. How many page table entries does this require in a standard linear (array-based) page table?



b) Now assume that we use a multi-level page table. We use a three-level table, such that a single root page directory points to pieces of mid-level page directory pieces which in turn point to pieces of the page table itself. Assuming that just two pages in an address space are valid – the very first page of the address space, and the very last – how many pages are used by this specific page table?



c) Assuming the page table and address-space usage described in part (b), what happens when the following code runs? If there is a problem, indicate at which line the problem occurs.

```
int main(int argc, char *argv[]) {
    int *p;
    p = 0;
    printf("value at p is %d\n", *p);
}
```

VPN 0 is VALID (first page of AS)

will print out whatever is in first 4 bytes of AS => code? 0?

d) Now we get to the large-page part of the question. Assume instead of 4-KB pages, we use large, 4-MB pages. First, list at least two major benefits of using large pages.

→ Smaller Page Tables → fewer entries

→ Makes TLBs work better → faster TLB miss lookup

e) Now, list at least two major disadvantages of large pages.

→ Internal Fragmentation (waste)

→ Can hurt (improve?) I/O performance (moving large pages to/from disk)

→ fills cache w/ fewer, large pages