# 537 Final Exam,
# Old School Style

## "The Job Interview"

Name: ________________________

This exam contains 13 old-school pages.

You are on a job interview. Unfortunately, on job interviews like this one, all they really want to see is that you can write code. So on this exam, you'll have to write some code. Even worse, you'll have to write code that shows that you understand how an operating system works. Yikes!

The good news: if you answer the questions well enough, you'll get the job!

The bad news: this is not true; it is just an exam.

Good luck!

The company is old school, and hence the exam is brought to you in 100% pure old-school style.

# GRADING PAGE

1. ____

2. ____

3. ____

4. ____

5. ____

6. ____

7. ____

8. ____

9. ____

10. ____

Total: ____ / 100

What is the maximum file size on a typical Unix system that uses an inode, an indirect pointer, and a double-indirect pointer? Assume a 4KB block size, and 12 direct pointers within the inode, and disk addresses that are 32 bits. Explain your work.

$$12 \text{ direct} + \underbrace{\underset{\text{indirect}}{\left(\frac{\overset{\text{block size}}{4KB}}{\underset{\text{disk addr}}{4 \text{ bytes}}}\right)}}_{1024} + \underbrace{(1024 \times 1024)}_{\text{double indirect}}$$

$$\text{ptrs}: 12 + 1024 + (1024)^2$$

$$\text{size}: \text{ptrs} * 4KB$$

Now, write a program that figures out the maximum file size on the file system you are running. What is the most efficient program you could write to do this? You are to use the following subset of the classic file-system API, i.e., open(), read(), write(), close(), lseek().

```
// assume: file system has room for
//         a big file

int fd = open("file", O_RDWR|O_CREAT|O_TRUNC);
assert(fd >= 0);
char buf[4096];
int i = 0;
while (1) {
    int rc = write(fd, buf, 4096);
    if (rc < 0)
        break;
    i++;
}
printf("max file is %d bytes\n", i*4096);
close(fd);
```

# PROBLEM 2: JOURNAL RECOVERY

You are to write the journal-recovery code. Actually, just a small piece of it: you are to write the piece of code that replays one journal entry. Assume that for this single transaction, all relevant blocks are already in memory. Here is the structure you should use to figure out what is in the transaction.

```
--------------------------------------------------------------------------------
struct transaction {
    int numblocks;
    unsigned int destinations[MAX_BLOCKS];
    unsigned char *blockarray[MAX_BLOCKS];
};

struct transaction *t; // use this pointer to access the struct
--------------------------------------------------------------------------------
```

In it:

- 'numblocks' includes the number of blocks in the transaction.
- 'destinations' is an array that includes, for each block to be replayed, the on-disk block address to which that block should be written.
- 'blockarray' is an array of pointers to the blocks, which are all in memory right now.

Your task: Go through the transaction structure, and write each block to its final destination. You can use the following primitive to write to the disk:

- WRITE(int block, char *data);

which takes a block number and a pointer to the data and writes the data to that block number.

```
for (i=0; i< t->numblocks; i++)
    WRITE(t->destinations[i], t->blockarray[i]);
    // assume this never fails;
       if it does, retry or exit
```

## PROBLEM 3: LOG-STRUCTURED FUN

In this question, we have a highly-simplified version of the log-structured file system (LFS). You are supposed to write some code that emulates the cleaner. Specifically, go through each update in a segment and figure out whether the update has a *live inode* in it. If you find a live inode, print "LIVE", otherwise print "DEAD".

Here are some data structure definitions to help you out:

```
// the inode map: records (inumber) -> (disk address) mapping
unsigned int imap[MAX_INODES];

typedef struct __inode_t {
    int      direct[10]; // just 10 direct pointers
} inode_t;

typedef struct __update_t {
    int      inumber;    // inode number of the inode in this update
    inode_t  inode;      // the inode
    int      offset;     // offset of data block in file, from 0 ... 9
    char     data[4096]; // the data block
} update_t;

typedef struct __segment_t {
    int      disk_addr;              // disk address of this segment (in bytes)
    update_t updates[MAX_UPDATES]; // the updates in this segment
                                     // (assume all MAX_UPDATES are used)
} segment_t;

segment_t *segment; // start with this pointer to the segment in question
```

// assume this is also in bytes for simplicity

(just like this)

Assume you are given a pointer to a segment_t (called 'segment'). Then go through each update in the segment, figure out whether the inode referred to in that update is live or not, printing "LIVE" or "DEAD" as you go:

```
for (i = 0; i < MAX-UPDATES; i++) {
    update_t *u = segment -> updates[i];

    if (imap[u -> inumber] ==
        segment -> disk-addr + (i * sizeof(update_t))
                             + sizeof(int))
        printf("LIVE\n");
    else
        printf("DEAD\n");

}
```

// key: does imap point to this inode?

skip over other updates

start at disk addr of seg

skip over the inumber in the update to get to the inode

# PROBLEM 4: TLB MISSING

TLB misses can be nasty. The following code can cause a lot of TLB misses, depending on the values of STRIDE and MAX. Assume that your system has a 32-entry TLB with a 8KB page size.

```
int value = 0;
int data[MAX]; // a big array

for (int j = 0; j < 1000; j++ {
    for (int i = 0; i < MAX; i += STRIDE) {
        value = value + data[i];
    }
}
```

What should you set MAX and STRIDE to so that you can achieve a TLB miss (but *not* a page fault) upon pretty much every access to the array 'data'? (describe)

STRIDE must be ≥ PAGESIZE (to TLB miss every time)

⇒ but, incrementing by sizeof(int) each time

$\Rightarrow \frac{PAGESIZE}{sizeof(int)} = \frac{8K}{4} = \boxed{2K}$

MAX must include > 32 pages ⇒ $\boxed{STRIDE * 33}$

What happens if MAX is too high? Too low?

| | | | |
|---|---|---|---|
| MAX < STRIDE | MAX ≤ STRIDE*32 | MAX ≥ STRIDE*33 and MAX < MEMORY SIZE | MAX ≥ MEMORY SIZE |
| TLB hits (just access data[0]) | still just TLB hits | TLB misses | Page Faults |

What happens if STRIDE is too high? Too low?

| | |
|---|---|
| STRIDE < PAGESIZE | STRIDE ≥ PAGESIZE |
| some TLB hits + some TLB misses<br>e.g. if STRIDE = PAGESIZE/4,<br>(TLB miss TLB hit, TLB hit, TLB hit) repeat | TLB miss every time |

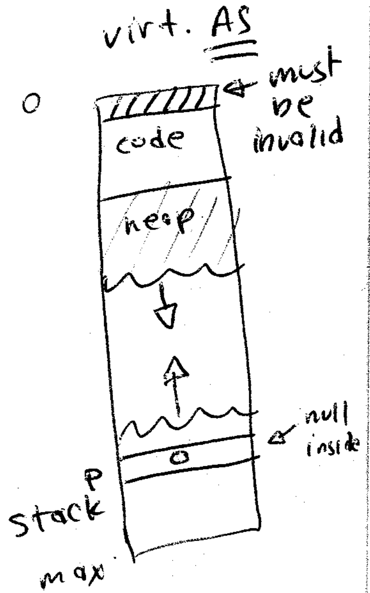
# PROBLEM 5: SEG FAULTIN' FUN

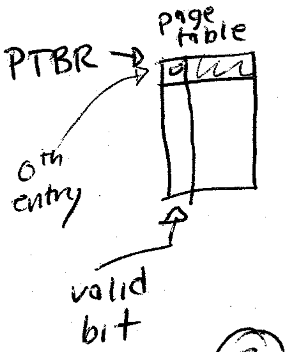
Sometimes badly-written C code dereferences a null-pointer, causing the program in question to crash. Write some code that does this; be brief!

```
int *p = NULL;      // assign p to NULL
*p = 10; //         // deref p to attempt to
                       set it to 10
```

Then, explain, in as much detail as needed, what the OS does in reaction to the null-pointer dereference, i.e., why it causes a program to fault, and how all the machinery behind it works. Be as detailed as you need to be!

Assume a system with linear page tables and 4KB pages.





① p is a valid memory location somewhere on the stack

inside p we have stored an address, which is set to "NULL" or 0.

② when we dereference "p" (e.g. *p = 10)

we are first loading the contents of p (which works, giving us a "0") and then accessing the virtual address within p or VA = 0 in this case

③ when VA = 0 is accessed, the H/W splits it into $\underbrace{VPN}_{20\,bits} \mid \underbrace{offset}_{12\,bits}$

which here is $\underbrace{\underbrace{0}_{VPN} \mid \underbrace{0}_{off}}$ (null VA)

④ the H/W checks VPN = 0 in TLB, which misses

⑤ the H/W or S/W checks the Page Table for VPN = 0 (the first entry), and finds it invalid (valid = 0)

⑥ an invalid access exception is raised; OS kills offending process

7) KEY: first page of AS must be invalid for null deref to always seg fault!

## PROBLEM 6: RAID

Some RAID code has been lost. You have to write it!

Assume you have a RAID-4 (parity-based RAID + a single parity disk), with a 4KB chunk size, and 5 disks total as follows:

| DISK-0 | DISK-1 | DISK-2 | DISK-3 | DISK-4 |
|---|---|---|---|---|
| block0 | block1 | block2 | block3 | parity(0..3) |
| block4 | block5 | block6 | block7 | parity(4..7) |
| ...... | ...... | ...... | ...... | ...... |

Fill in the routine SMALLWRITE() below:

```
// SMALLWRITE()
//
// This routine takes a logical block number 'block' and writes
// the single block of 4KB referred to by 'data' to it.
//
// It may have to use the underlying primitives:
//   READ(int disk, int offset, char *data)
//   WRITE(int disk, int offset, char *data)
//   XOR(char *d1, char *d2) (xors one block with another)

void SMALLWRITE(int block, char *data) {
     int disk = block % 4;
     int off = block / 4;
     char olddata [4096], oldparity [4096];
     READ (disk, off, olddata);
     READ (4, off, oldparity);
     char *newparity = XOR (oldparity, XOR(olddata, data));
     WRITE (disk, off, data);
     WRITE (4, off, newparity);



}
```

could do in parallel

read old data + parity

① use this to see what data has changed

② then flip bits in old parity to get new parity

could write new data + new parity in parallel too

PROBLEM 7: DEADLOCK? DEADLOCK.

You are given the following code, which adds two vectors together, and does so in a multithread-safe way.

```
void
vector_add(vector *v1, vector *v2) {          ← 0
    mutex_lock(v1->lock);
    mutex_lock(v2->lock);                     ← 1
    for (i = 0; i < v1->size; i++) {
        v1[i] = v1[i] + v2[i];
    }
    mutex_unlock(v1->lock);
    mutex_unlock(v2->lock);
}
```

Then you are told that two different concurrently-executing threads, 1 and 2, call this code as follows:

| Thread 1: | Thread 2: |
|---|---|
| vector_add(&vectorA, &vectorB); | vector_add(&vectorB, &vectorA); |

Unfortunately, this can lead to a DEADLOCK*, in which the program gets stuck, with each thread waiting for the other to make progress.

1)- Why does this happen? (describe, or show with a picture)

2)- How could you write vector_add() so that this deadlock never happens?

1) Thread 1:

mutex-lock(&vectorA→lock);

interrupt

Thread 2:

mutex-lock(&vectorB→lock);

//now this thread tries to grab A's lock but can't

//and same over here

2) Lots of solutions possible:

1) general lock around lock acquisition (avoid hold+wait)

@ 0 above: mutex-lock(&generalLock);

@ 1 above: mutex-unlock(&generalLock);

2) (avoid cycle) order locks, always acquire in same order

e.g.

```
if (v1>v2) {
    //grab v2, then v1
}else{
    //grab v1, then v2
}
```

---

* Sorry, a deadlock question. But it should not be too hard**, should it?

** Probably not true.

## PROBLEM 8: OUT OF CONDITION

Your co-worker implements the following code for condition variables (and specifically, the cond_wait() and cond_signal() routines) using *semaphores*:

```
typedef struct __cond_t {
    sem_t s;
} cond_t;

void cond_init(cond_t *c) {
    sem_init(&c->s, 0);
}

// cond_wait(): assumes that the lock 'm' is held
void cond_wait(cond_t *c, mutex_t *m) {
    mutex_unlock(&m); // release lock and go to sleep
    sem_wait(&c->s);
    mutex_lock(&m);   // grab lock before returning
}

void cond_signal(cond_t *c) {
    sem_post(&c->s);  // wake up one sleeping waiter (if there is one)
}
```

Unfortunately, it is buggy. Why doesn't this code work properly?

10

imagine a cond-signal before a cond-wait.

→ w/ "real" conditions, the cond-wait would still get stuck waiting (cond-signal only wakes a waiter if one is already waiting)

→ here, a sem-post would happen on cond-signal, and a subsequent cond-wait would not get stuck waiting

the problem is that the semaphore has state, whereas a condition var. does not

# PROBLEM 9: FETCH, FIDO, FETCH

You are given a new atomic primitive, called FetchAndSubtract(). It executes as a single atomic instruction, and is defined as follows:

```
int FetchAndSubtract(int *location) {
    int value = *location; // read the value pointed to by location
    *location = value - 1; // decrement it, and store result back
    return value;          // return old value
}
```

You are given the task: write the lock_init(), lock(), and unlock() routines (and define a lock_t structure) that use FetchAndSubtract() to implement a working lock.

```
typedef struct _lock_t {
      int ticket;
      int turn;
 } lock_t;

 lock_init (lock_t *lock) {                    // ok to
       lock->ticket = lock->turn = 0;             start @
 }                                                0 and
                                                  go negative
                                                  right away

  lock (lock_t *lock) {
       int myturn = FAS (&lock->ticket);
       while (myturn != lock->turn)
           ; // spin
  }

  unlock (lock_t *lock) {
       FAS (&lock->turn);
  }
```

10

// this is a ticket lock -> simpler solutions here ok too

// e.g. init lock to something   acquire by while (FAS(lock) < something) ; //spin   release by setting back to something

## PROBLEM 10: DISTRIBUTED FILE SYSTEMS

NFS and AFS are two famous distributed file systems, yet they each have cases where one performs noticeably different than the other.

Assuming you can only access a single file (and using only the limited API: open(), read(), write(), close(), and lseek() calls), write a program that runs MUCH MUCH faster when run upon NFS than AFS.

many answers possible!

```
open ("file"); <= very fast on NFS //
on AFS, fetches WHOLE file
(even if file isn't accessed, or only some of it was read)
```

⑤

Using the same assumptions as above, write a program that runs MUCH MUCH faster when run upon AFS than NFS.

```
fd=open ("file2", O_RDONLY);
  while (1) // well, not forever, but a loop would do
      int rc= read (fd, ...);
      if (rc<0)
          break;

  close (fd)
```

⑤

// assuming file > local mem but < local disk [and (not needed) file has been accessed before]

key: if so, AFS -> local, NFS -> goes to server