The University of Texas at Austin
CS 439 Principles of Computer Systems: Spring 2013

Midterm Exam II

- This exam is **120 minutes**. Stop writing when "time" is called. *You must turn in your exam; we will not collect it.* Do not get up or pack up between 110 and 120 minutes. The instructor will leave the room 123 minutes after the exam begins and will not accept exams outside the room.
- There are **15** problems in this booklet. Many can be answered quickly. Some may be harder than others, and some earn more points than others. You may want to skim all questions before starting.
- **This exam is closed book and notes. You may not use electronics: phones, calculators, laptops, etc.** You may refer to ONE two-sided 8.5x11" sheet with 10 point or larger Times New Roman font, 1 inch or larger margins, and a maximum of 55 lines per side. **Please leave your UT IDs out.**
- If you find a question unclear or ambiguous, be sure to write any assumptions you make.
- Follow the instructions: if they ask you to justify something, explain your reasoning and any important assumptions. **Write brief, precise answers. Rambling brain dumps will not work and will waste time.** Think before you start writing so that you can answer crisply. Be neat. If we can't understand your answer, we can't give you credit!
- There is no credit for leaving questions blank. However, to discourage unfocused responses, we will be grading the clarity of your answer. Moreover, some questions impose sentence limits.
- Don't linger. If you know the answer, give it, and move on.
- **Write your name and UT EID on this cover sheet and on the bottom of every page of the exam.**
- **Circle your TA's name and section meeting day below.**

Do not write in the boxes below.

I (xx/18)	II (xx/20)	III (xx/18)	IV (xx/24)	V (xx/20)	Total (xx/100)

TA (circle one): Sebastian (Tues.) Parth (Wed.) Navid (Thurs.)

Do not write your name on this exam

Number:

I Miscellaneous (18 points total)

1. [5 points] The code below is intended to "scan an array of [positive] `int`s [delimited with a 0] and return a pointer to the first occurrence of `val`" (Bryant & O'Halloran, Section 9.11):

```
int *search(int *p, int val)
{
    while (*p && *p != val)
        p += sizeof(int);

    return p;
}
```

But it has an error. **State the problem in the code above. Be brief (no more than two sentences).**

The pointer is being incremented by the size of an integer (4 bytes). To match the programmer's intent, the code inside the loop should be, simply, "`p++;`" or "`p = p + 1`".

2. [5 points] In class we observed that there is an analogy between paged virtual memory and indexed file systems: in both cases, the **key data structure** implements a mapping from an **abstract thing** that is visible to user-level processes to a **physical resource**.

Flesh out the analogy by filling in the table. Be specific (answers like "RAM" and "disk" will not get credit). The questions refer to the boldface terms above.

	virtual memory	indexed file system
What is the key data structure?	a tree of page tables	inode
What is the abstraction?	virtual address/page	offset in file
What is the physical resource?	physical address/page/frame	disk block/sector

3. [4 points] The Hailperin book ("Operating Systems and Middleware: Supporting Controlled Interaction") discusses *extent maps*. In what context (virtual memory? file systems? I/O? concurrency? etc.) are extent maps discussed, and what are they?

Below, state the context for extent maps, and then describe them. Be brief (no more than two sentences). To get credit, an answer must have the correct context *and* description.

Extent maps are another way of implementing files. The metadata is a list of tuples: (length, starting file block, starting disk block).

4. [4 points] What is the key on-disk data structure for enabling a system (such as a file system) to recover from crashes?

Below, state the data structure. This has a one-word answer.

Log or journal.

II Managing memory (20 points total)

5. [4 points] Bryant & O'Hallaron [CS:APP2e] give a number of ways that virtual memory (VM) can simplify or improve various system functions. Which of the following functions do they mention?

Circle ALL that apply:

A VM is a tool for caching

B VM simplifies linking

C VM simplifies loading

D VM simplifies sharing

E VM simplifies memory allocation

All of them. We also discussed many of these in class.

6. [4 points] The TLB is a cache. What does this cache hold, and on the x86 architecture, which system entity inserts entries into it?

Circle the BEST answer:

A The TLB caches the contents of recently accessed memory pages, and the CPU inserts entries.

B The TLB caches recently used virtual-to-physical address mappings, and the CPU inserts entries.

C The TLB caches the contents of recently accessed memory pages, and the kernel inserts entries.

D The TLB caches recently used virtual-to-physical address mappings, and the kernel inserts entries.

B.

7. [4 points] *Heap memory management* refers to decisions about . . .

Circle the BEST answer:

A . . . which physical pages should hold the heap.

B . . . which virtual pages to swap to the disk when physical memory is full.

C . . . when to grow (meaning add pages to) the stack of a process.

D . . . what happens in the implementation of malloc() and free().

E . . . where in the network to place distributed shared memory (DSM).

D

8. [8 points] Consider a processor architecture with 32-bit virtual addresses. In this architecture, the memory management unit (MMU) expects a two-level page table structure analogous to the one used in JOS: each process has a page directory, each entry of which can point to a page table.

On this architecture, the upper 6 bits of an address determine the page directory index, the next 10 bits determine the index in the second-level page table, and the bottom 16 bits determine the offset.

Below, state how many entries are in a page directory, and explain *briefly*.

64: there are 6 upper bits, and $2^6 = 64$.

Below, state how many entries are in a second-level page table, and explain *briefly*.

1024: there are 10 middle bits, and $2^{10} = 1024$.

Below, state the page size on this machine, and explain *briefly*.

$2^{16} = 64$ KB. This is because the bottom 16 bits index into a page, so a page can be as large as 2^{16}.

Below, state the maximum number of *virtual* pages per process, and explain *briefly*.

$2^{16} = 64$K pages, since the upper 16 bits choose a virtual page.

III Disks and file systems (18 points total)

9. [9 points] Consider a disk with the following characteristics, some of which are non-standard:

- The disk rotates 15,000 times per minute.
- The disk has 100 platters.
- The disk has 2048 sectors per track (assume all tracks have the same number of sectors).
- A sector is 512 bytes.
- The disk has 8192 (2^{13}) cylinders.
- The average seek time is 5 seconds.
- Assume 0 cost for track-to-track seek time and 0 cost for changing which disk head is active.

For the questions below, you can make approximations of several percentage points.

How many milliseconds does the disk take to perform one rotation? Explain *briefly*.

4 ms. 60 sec/min * 1 min/15,000 rot * 1000 ms/sec = 4 ms.

What is the throughput of this disk, assuming a perfectly sequential access pattern? Explain *briefly*. (By "throughput", we mean number of bytes/sec that can move from the disk to memory.) Try to avoid long multiplications; work with powers-of-two whenever you can.

The track buffer and the sequential access pattern means we can fully discount seeks and extra rotations. So the question becomes "how many bytes can be read from the disk in a single rotation, and how long does a rotation take?"

In one rotation, the disk can read 512 bytes/sector * 2048 sectors/track. Above, we learned how long a rotation takes: 4ms, so we can do 250 of them per second.

Thus, we get transfer bandwidth = 2048 sectors/track * 512 bytes/sector * 250 rotations/second * 1 track/rotation = $2^{11} \cdot 2^9 \cdot 250 = 2^{20} \cdot 250$ bytes /sec = 250 MB/sec.

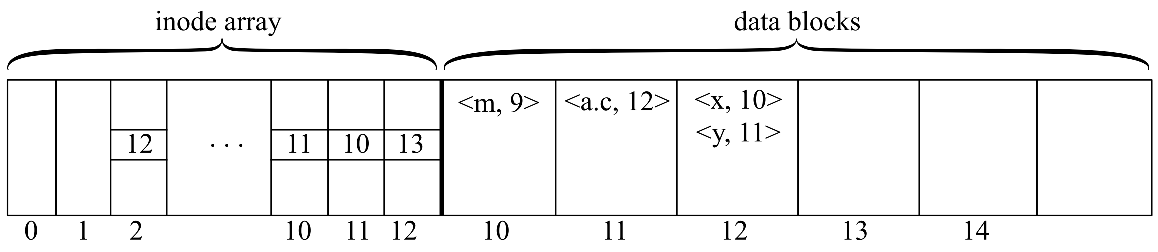
Consider the following design problem. You are writing an application that works with 2 GB of data, but you do not have that much RAM: you will have to store your data on one or more disks. You are concerned because while your application's data access pattern has some locality—the application processes data in 1 MB (2^{20} byte) chunks—the chunks themselves exhibit no locality (each 1 MB chunk could be anywhere in the 2GB of data), and the mix of reads and writes is arbitrary. This might mean many disk seeks, which are insanely expensive on this disk (see the characteristics above).

But you come up with a way to avoid disk seeks. Your computer has many slots for additional disks, and you get permission from your manager to purchase a number of disks of the kind above.

How do you avoid seeks (of any kind), and how many disks do you have to buy? Explain *briefly*.

Use just one track on each platter. Each track stores 1 MB, and each disk has 100 platters, leading to 100 MB/disk. This means you need 20 disks.

Using only one track per platter is a hack, but it exaggerates an actual design approach: some applications indeed do not use all of their disks, preferring instead to use the outer stripes of multiple disks.



10. [5 points] Consider the simple file system depicted above, with the following characteristics:

- Each inode contains a single data block pointer, depicted inside the inode structure.
- Note that inodes and data blocks are numbered separately. (The confusion is intentional; we are asking you to think about what the different numbers mean.)
- As mentioned in class, inode 2 contains the inode for the root directory of the file system.

State the contents of the file system in terms of the full path names of files.

/x/a.c, /y/m

11. [4 points] Assume a file system that contains directory `/a` and file `/a/b.txt`.

For each of the four operations below, state how many inodes it creates. We describe the operation both in terms of the command that invokes the operation and with a comment (written as `#...`) that states what the operation is doing. You can read either or both.

A. `$ ln /a/b.txt /a/c.txt`
# use hard link to make `/a/c.txt` a synonym for `/a/b.txt`
Number of inodes created: ________________0

B. `$ ln -s /a/b.txt /a/d.txt`
# use soft (symbolic) link to make `/a/d.txt` refer to `/a/b.txt`
Number of inodes created: ________________1

C. `$ mkdir /a/e`
# create an empty directory `e` inside `/a`
Number of inodes created: ________________1

D. `$ touch /a/f.txt`
# create new file with name `f.txt` inside directory `/a`
Number of inodes created: ________________1

IV JOS (24 points total)

12. [6 points] This question asks about system call handling. Recall the `SETGATE` macro:

```
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level
#define SETGATE(gate, istrap, sel, off, dpl)
```

To enable system call handling, you wrote, in lab 5b, a line of code like the following:

```
SETGATE(idt[T_SYSCALL], XXX, YYY, ZZZ, 3)
```

Notice the last parameter: a 3. (The parameters `XXX`, `YYY`, and `ZZZ` are unimportant for this question.) **Why, exactly, does the code require a 3 above?** You do not need a long answer, but your answer should precisely refer to the requirements of the system.

In the same lab, you also arranged that, when control returns from the kernel to the calling environment, (a) the environment's `%eax` register contains the system call's return value, and (b) the rest of the environment's registers are unmodified.

How, exactly, did you arrange for (a) and (b) together? Be brief. You don't need a long answer.

13. [15 points] Recall that JOS is an exokernel: it exposes details of the hardware to user-level processes (known as *environments*). It also exposes data structures that the kernel itself maintains, including the `pages` array and the `envs` array. Recall that `pages` is an array of `struct Pages`, and `envs` is an array of `struct Envs`, which are like PCBs (process control blocks).

What is the purpose of a `struct Page`? Explain *briefly* (no more than two sentences).

It holds metadata for a physical page of memory (whether it's allocated or not, for example).

In lab 5, you arranged for `envs` to appear not only at virtual (linear) address `envs`, where it is readable and writable by the kernel, but also at a virtual (linear) address, `UENVS` (which is defined to be `0xeec00000`), where environments can read this data structure but not modify it. Recalling that the number of elements in the `envs` array is `NENV`, what lines of code did you write to make `envs` appear at virtual address `UENVS`?

Write the lines below, and note that the definitions below the question may be helpful:

`pde_t* kern_pgdir`**:** Holds the virtual address of the page directory that is "under construction".

`boot_map_region()`**:** Function with the following signature:

```
void boot_map_region(pde_t *pgdir, uintptr_t va, size_t size,
                     physaddr_t pa, int perm)
```

The next question asks how the lines of code that you wrote above modify the page directory and relevant page tables. The following equations may help (the symbol $\ll$ denotes the C operator <<):

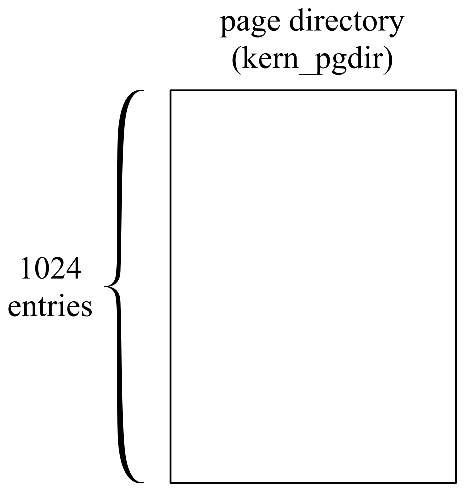
$$\begin{aligned} \text{0xeec00000} &= \text{0x2ef} \cdot 4 \cdot 2^{20} \\ \text{0xeec00000} &= (\text{0x2ef} \ll 22) \\ \text{0x2ef} &= 751 \end{aligned}$$

In the figure below, depict the entries that are created, both their indexes and their contents:

The question was ill-posed. We made a mistake. Above, it should have said:

$$\begin{aligned} \text{0xeec00000} &= \text{0x3bb} \cdot 4 \cdot 2^{20} \\ \text{0xeec00000} &= (\text{0x3bb} \ll 22) \\ \text{0x3bb} &= 955 \end{aligned}$$

Given this, the correct answer should have been that the 955th entry in the page directory has the physical page number of a page table. That page table has some of its entries filled in, with the permissions equal to present and user-accessible (but not writable). Since we messed up the arithmetic above, we threw out the question; everyone got credit.



Now, assume that a JOS environment is running (as in lab 5); its page structures include the mapping for `UENVS` that you set up above. Consider the code below, which is legal C:

```
int* p;

/* ... */

p = (int*)UENVS;
*p = 5;
```

If a JOS environment executes this code, does that result in a fault? If so, why? If not, why not? Be *brief* (no more than two sentences). Only answers with explanations will get credit.

Yes, it does, because the environment is trying to store to memory through a virtual address that is read-only.

Let `P` equal the physical memory address that `UENVS` maps to. Consider the following incorrect attempt to read the contents of `P` in kernel mode:

```
int* q = (int *)P;
int val;

val = *q;
```

Why does the code fail to read `P`'s contents into `val`? Explain *briefly*.

Because `P` is translated to a virtual address, and that virtual address is not the one that we want.

14. [3 points] This is to gather feedback. Any answer, except a blank one, will get full credit.

Please state the topic or topics in this class, since midterm 1, that have been least clear to you.

Many students wrote MCS spinlocks

Please state the topic or topics in this class, since midterm 1, that have been most clear to you.

V Multithreaded programming (20 points total)

15. [20 points] In this problem, you will synchronize access to a dishwashing machine (which we call *D*) that is shared by multiple roommates. *D* can hold `MAX_DISHES` dishes. *D* supports three actions: a roommate can *get* dishes (presumably clean ones) from *D*, a roommate can *put* dishes (presumably dirty ones) in *D*, and a roommate can *try-to-run* *D*, to attempt clean the dirty dishes that it is holding. *D* itself can be in one of three states:

- `RECEIVING_DIRTY`: putting dishes in *D* is acceptable, but getting dishes from *D* is not.
- `CLEANING`: in this state, it is not permissible to put dishes in *D* or to get dishes from *D*.
- `HOLDING_CLEAN`: in this state, it is permissible to get dishes (which are presumably clean) from *D* but not to put dishes into *D*.

The state transitions are as follows. *D* transitions from `RECEIVING_DIRTY` to `CLEANING` when someone runs the machine; meanwhile, running the machine is permissible only when *D* holds `MAX_DISHES` dishes. From the `CLEANING` state, *D* transitions (after some time) to `HOLDING_CLEAN`. Finally, *D* must transition from `HOLDING_CLEAN` to `RECEIVING_DIRTY` when the last dish has been removed from it.

The actions have the following constraints: a *get* must wait, unless *D* is in the `HOLDING_CLEAN` state; a *put* must wait, unless *D* is in the `RECEIVING_DIRTY` state and the dishwasher is not yet full; and a *run* should happen only when *D* is full (a *try-to-run* can happen at any time).

Dishes are stored in one of two places: in *D* or in a sink, which we model as having infinite capacity. As is often the case, no one stores dishes in cabinets. To simplify, we assume that there are two different types of roommates: those who selfishly use dishes for eating, and those who altruistically move dishes from the sink to *D*. We model each as threads, with the following pseudocode:

```
// globals
Dishwasher D;
Sink sink;

void eating_roommate() {

    Dish *dish = NULL;

    while (1) {

        <live; get hungry>

        dish = D.Get();

        <eat>

        sink.Put(dish);

    }
}
```

```
void cleaning_roommate() {

    Dish* dish = NULL;

    while (1) {

        D.TryToRun();

        dish = sink.Get();
        D.Put(dish);
    }
}
```

Implement the `Dishwasher` **monitor: write down the state variables, the synchronization objects, and implement the methods.** Be certain to follow the 439 coding conventions. Assume that the `Sink` monitor is implemented correctly (so `sink.Get()` waits until it can return a dirty dish, etc.). You may indicate activities other than synchronization and mutual exclusion using angle brackets (for example, ⟨store dish⟩ and ⟨run dishwasher⟩).

```
class Dishwasher {

    public:

        // Initialize the state and synchronization variables
        Dishwasher();

        // See the constraints outlined above
        void Put(Dish* dish);

        // See the constraints outlined above
        Dish* Get();

        // This method checks whether running the dishwasher is permissible.
        // If not, the method immediately exits. If so, the method runs the
        // dishwasher, treating <run dishwasher> as blocking I/O.
        void TryToRun();

    private:
        // FILL THIS IN









};
```

```
// Here and on the next page, give the implementations of
//     Dishwasher::Dishwasher() [this should initialize all of the state],
//     Dishwasher::Put(),
//     Dishwasher::Get(),
//     Dishwasher::TryToRun()
```

Space for code and/or scratch paper

Data members:

```
    Mutex m;

    // we'll use only one cv (and broadcast). you could use more CVs and
    //   signal. Note that multiple CVs is a  performance
    //   optimization; correctness is ensured by state variables and
    //   checking predicates.

    Cond cv;

    typedef enum {RECEIVING_DIRTY, CLEANING, HOLDING_CLEAN} state_t;

    state_t state;
    int num_dishes;
```

Methods:

```
Dishwasher::Dishwasher() :
    state(RECEIVING_DIRTY),
    num_dishes(0)
{
    m.init();
    cv.init();
}

void
Dishwasher::Put(Dish* dish)
{
    M.acquire();

    while (state != RECEIVING_DIRTY || num_dishes >= MAX_DISHES) {
        cv.wait(&m);
    }

    <store dish>

    ++num_dishes;

    // could do ...
    // if (++num_dishes == MAX_DISHES) {
    //     cv.broadcast(&m)
    // }
    // ... but because the caller of TryToRun never waits, we don't
    // actually need to wake it up.
    M.release();
}

Dish *
Dishwasher::Get()
{
    Dish* dish;

    M.acquire();

    while (state != HOLDING_CLEAN || num_dishes == 0)
        cv.wait(&m);

    dish = <remove dish>

    if (--num_dishes == 0) {
        state = RECEIVING_DIRTY;
        cv.broadcast(&m);
    }

    M.release();

    return dish;
}

Dishwasher::TryToRun()
{
    M.acquire();

    if (state == RECEIVING_DIRTY && num_dishes == MAX_DISHES) {
        state = CLEANING;
        <run dishwasher>
        state = HOLDING_CLEAN;
        cv.broadcast(&m);
    }

    M.release();
}
```

Space for code and/or scratch paper

End of Midterm