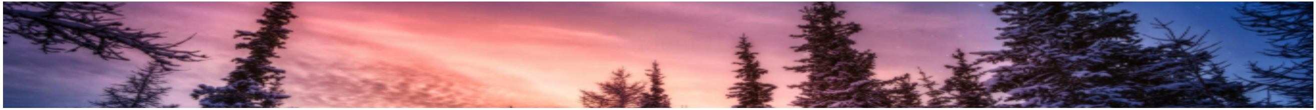
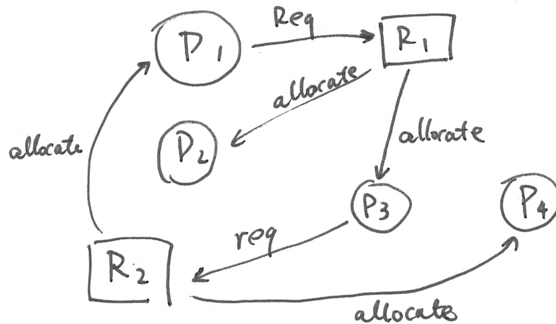
Final Examination
CS 111, Winter 2002
3/14/2022, 8 – 11am



1. **Deadlock**. Consider a system with four processes P1, P2, P3, and P4, and two resources, R1, and R2, respectively. Each resource has two instances. Furthermore:
   - P1 allocates an instance of R2, and requests an instance of R1;
   - P2 allocates an instance of R1, and doesn't need any other resource;
   - P3 allocates an instance of R1 and requires an instance of R2;
   - P4 allocates an instance of R2, and doesn't need any other resource. **(15 points, 5 points each question)**

(a) Draw the resource allocation graph



(b) Is there a cycle in the graph? If yes, name it.

Yes, there is a cycle.

P1 requset R1, R1 allocate to P3.

P3 request R2, R2 allocate to P1.

(c) Is the system in deadlock? If yes, explain why. If not, give a possible sequence of executions after which every process completes.

~~Yes~~. No.

Although There is a cycle in a graph for P1, R1, P3, R2, when P2 and P4 finishes, there will be available R1 for P1 and available R2 for P3.

Order: ① P2, P4 execute, finish ② P1, P3 execute.

2. **File system reliability**. Professor Harry writes the following program for grading students' projects. Per-project grades are stored in a set of input files, and the following program's goal is to compute a final course grade for each student and write it to file name.grade. **(3 points)**

```
main():
     remove all files ending with ".grade"
     for each student name s in alphabetical order:
             read assignment scores for student s
             calculate final grade filename = s + ".grade" fd =
             create(filename)
             write(fd, final grade, ...)
             close(fd)
             printf("finished with %s\n", s)
```

Harry uses a laptop with a journaling file system in a mode that the journal contains both file content and the metadata. He runs the following command:

program | cat

Harry sees "finished with x" for all students with names up through "p", and then his laptop crashes. Harry reboots his laptop.

Harry thinks he may have to re-run the program for some or all students. Explain what guarantees he has about which final grades will be on disk after the restart.

The printf() here occurs after the student's grade is written to disk. It 's a committed message. Since Harry sees "finished with "p" " it's garunteed that all student up to "p" is on disk.

3. Explain copy-on-write and give at least one instance when it might be useful. **(2 points)**

Copy on Write is that only making a copy of a file if it's modified. It can be applied to many places. E.g. Backup snapshot may use COW to save space. (No need to copy everyfile). E.g. fork() also use COW to save some space

4. **Process and Thread:** Consider the following C program. Assume that all system calls succeed. **(20 points)**

```
1     void* func1(void *args) {
2         printf("Yellow: %d\n",*((int*)args));
3         exit(0);
4     }
5
6     void* func2(void *args) {
7         printf("Red: %d\n", ((int*args)[0]);
8         return NULL;
9     }
10    int main(void) {
11        pid_t pid; pthread_t pthread; int status; //declaring vars
12        int fd = open("cs111.txt", O_CREAT|O_TRUNC|O_WRONLY, 0666);
13        int *subaru = (int*) calloc(1, sizeof(int));    subaru =0.
14        printf("Main: %d\n", *subaru);                   Main: 0.
15        if(pid = fork()) {    pid . pid.
16            *subaru = 1337;
17            pid = fork();
18        }
19        if(!pid) {            pid > 1
20            pthread_create(&pthread, NULL, func2,
21                           (void*) subaru);
22        } else {
23            for(int i = 0; i < 2; i++)
24                waitpid(-1, &status, 0);
25            pthread_create(&pthread, NULL, func1,
26                           (void*) subaru);
27        }
28        pthread_join(pthread, NULL);
29        if(*subaru == 1337)
30            dup2(fd, fileno(stdout));
31        printf("All done!\n");
32        return 0;
33 }
```

(a) Including the original process, how many processes are created? Including the original thread, how many threads are created? **(2 points)**

Line 15,17 creates a new processes.

Including the original one, 2+1 =3 processes.

Each process create one new thread,

Including original one, 3+1 = 4 Threads.

(b) Provide all possible outputs in standard output. If there are multiple possibilities, put each in its own box. You may not need all the boxes. **(10 points)**

Possibility 1:	Possibility 2:	Possibility 3:	Possibility 4:
Main: 0 Red: 0. Red: 1337 ~~All done!~~ Yellow: 1337 All done!	Main: 0 Red: 1337 Red: 0 Yellow: 1337 All done!		

(c) Provide all possible contents of cs111.txt. If there are multiple possibilities, put each in its own box. You may not need all the boxes. (2 points)

Possibility 1:	Possibility 2:	Possibility 3:	Possibility 4:
All done! All done!			

(d) Suppose we deleted line 29, would the contents of cs111.txt change? If they do, how? **(3 points)**

All done
All done
All done.

Yes. it would. suberabu is NOT 1337 for one process, In that case, the process will move its last stdout to cs111.txt as well. The last stdout will be mixed with cs111.txt 3 "All done

(e) What if, in addition to doing the change in part (d), we also move line 12 (where we open the file descriptor) between lines 18 and 19? What would cs111.txt look like then? **(3 points)**

In this case, the ~~st~~ cs111.txt will be opened by each process individually. There are more context switch and may get slower, but the file should roughly be the same with 3 all done.

5. **TLB.** Consider the following piece of code which multiplies two matrices:

```
int a[1024][1024], b[1024][1024], c[1024][1024];

multiply()
{
     unsigned i, j, k;
     for(i = 0; i < 1024; i++)
          for(j = 0; j < 1024; j++)
               for(k = 0; k < 1024; k++)
                    c[i][j] += a[i,k] * b[k,j];
}
```

Assume that the binary for executing this function fits in one page, and the stack also fits in one page. Assume further that an integer requires 4 bytes for storage. Compute the number of TLB misses if the page size is 4096 and the TLB has 8 entries with a replacement policy consisting of LRU. **(10 points)**

For each page. 4096 Byte / 4 Byte = 1024 integers. is stores. Therefore. when ~~ope~~ executing,

a[i,k] and c[i,j] will be ~~on the same pa~~ on it's page unless i is changed.

b[k,j] will be a miss every time, since k is ~~chaging~~ changing.

c[i,j] : miss 1024 times.

a[i,k] : miss 1024 times.

b[k,j] : miss $(1024)^3$ times.

$2048 + (1024)^3$ times.

6. **Monitor and Condition Variable**. In this problem, you will implement a monitor to help a set of drivers (modeled as threads) synchronize access to a set of five keys and a set of ten cars. Here is the problem setup:

   – The relationship between the keys and the cars is that key 0 operates cars 0 and 1, key 1 operates cars 2 and 3, etc. That is, key i works for cars 2i and 2i + 1.

   – If a key is being used to operate one car, it cannot be used to operate the other.

   – A driver requests a particular car (which implies that the driver needs a particular key). However, there may be many more drivers than cars. If a driver wants to go driving but cannot get his desired car or that car's key, it waits until the car and key become available. When a driver finishes driving, it returns his key and notifies any drivers waiting for that key that it is now free.

– You must allow multiple drivers to be out driving at once, and you must not have busy waiting or spin loops.

– Note: there could be many, many instances of `driver()` running, each of which you can assume is in its own thread, and all of which use the same monitor, `mon`.

On the next page, fill in the monitor's remaining variable(s) and implement the monitor's `take_key()` and `return_key()` methods. **(20 points)**

```
typedef enum {FREE, IN_USE} key_status;

class Monitor {
    public:
        Monitor() { memset(&keys, FREE, sizeof(key_status)*5); }
        ~Monitor() {}
        void take_key(int desired_car);
        void return_key(int desired_car);
    private:
        Mutex mutex;
        key_status keys[5];
        /* YOU MUST ADD MATERIAL BELOW THIS LINE */
        queue_t *q[5];

        int guard;
};

void driver(thread_id tid, Monitor* mon, int desired_car) {
     /* you should not modify this function */
     mon->take_key(desired_car);
     drive();
     mon->return_key(desired_car);
}

void Monitor::take_key(int desired_car) {
     /* YOU MUST FILL IN THIS FUNCTION. Note that the argument refers
        to the desired car. */
     key = desired_car /2;   while (test-andset (this.guard)) { }
     this.mutex.lock();
     if ( keys[key] == IN_USE){
         enqueue(this->q[key], self);  this->guard=0;  this.mutex.unlock(); yield();
     } else {
      this.keys[key] = IN_USE;  this->guard=0;
     }
     this->mutex.unlock();
}

void Monitor::return_key(int desired_car) {
     /* YOU MUST FILL IN THIS FUNCTION. Note that the argument refers
        to the desired car. */
     k = desired_car /2;
     While (test_and_set(this.guard)) {}
     this.mutex.lock()
     if (!queue_empty(this->q[k])) {
        wake up (dequeue (this->q[k]));
     } else {
        keys[k] = FREE;
     }

     this.guard = 0;
     this.mutex.unlock();
}
```

7. **Disk**. Consider a disk with the following characteristics:

– The disk rotates at 12,000 RPM (rotations per minute)

– The disk has 10 platters

– Each sector is 512 bytes

– There are 1024 sectors per track (we are ignoring the fact that the number of sectors per track varies on a real disk)

– There are 4096 tracks per platter

– The track-to-track seek time is 0 milliseconds

– If the disk head has to seek more than a single track, the seek time is given by 1 + t * .003 milliseconds, where t is the number of tracks that the disk is seeking over.

– Ignore the time to transfer the bits from the disk to memory; that is, once the disk head is positioned over the sector, the transfer happens instantaneously

(a) What is the storage capacity of the disk in bytes or gigabytes? Explain briefly. **(5 points)**

$1024 \frac{\text{sector}}{\text{track}} \cdot 4096 \frac{\text{track}}{\text{platter}} \cdot 10 \text{ platter/disk} \cdot 512 \text{ bytes/sector}.$

(b) What is the sequential transfer bandwidth, expressed in bytes/second or megabytes/second? Explain briefly. **(5 points)**

$12000 \text{ RPM} \cdot 1024 \text{ sector/track} \cdot 512 \text{ byte/sector} \cdot \frac{1 \text{ Min}}{60 s}$

8. **File System**. As discussed in our lectures, creating a new file in a directory needs to update 4 blocks under Linux ext2: the inode bitmap, the file inode, the directory inode,

and the directory's data block. Assume the directory inode and the file inode are in different on-disk blocks. **(20 points, 4 points for each question)**

Part I: Assume we perform neither journaling nor FSCK. What would happen if a crash occurs after only updating the following block(s)? Choose from the following answers and explain the reasons:

- no inconsistency
- wasted data block/inode
- multiple file paths may point to the same inode
- point to garbage data
- multiple problems listed above.

(a) Bitmap

Inconsistency: Bitmap says inode ~~it's~~ allocated, but other places are not.

(b) Directory inode and Directory data

Points to Garbage data.
Inconsisteny.

If we trust Directory Data ~~Inode~~ points to unchanged Inode, we get garbage. It's incosistent with Bitmap too.

Part II:

Let us add a simple implementation of data (including data and metadata) journaling to our file system and perform the same file creation as Part I. Assume each transaction on the journal starts with a header block and finishes at a commit block. If the system crashes after the following number of blocks have been synchronously written to disk (including the journal and real FS), what state will the FS be in after the reboot? What can we do to recover?

(a) 1 disk write (hint: just the transaction header block is written to the journal)

It's only written to Journal but ~~NOT in~~ ~~may~~ NOT in actual file system. We need to recover everything. Because even the Journal is ~~incomplete~~ uncommitted

FS will be at it's initial state untouched.

Since Journal writes and Joural Commit occus

before FS writes. We need to re-run the whole.

program to recover, the joinal here isn't useful

for recover.

It's in old status

(b) 8 disk writes (hint: transaction header, plus 4 blocks, plus commit block to journal, plus two data blocks to the real FS)

Since Journal contains commit block, it's a committed Journ

However, writing to the actual FS is not completed.

~~We~~ Because the transaction is not cleared.

We may follow the Journal and recover using the

committed message. It's in old status.

(c) 11 disk writes (hint: you also have to clear the transaction after writes to the real FS finish)

Transection Header + 4 blocks + commit block

+ 4 data blocks + Clearing transaction >= 11.

Since the Journaling has entered the stage of

journal clear, the FS write must be completed.

There is nothing needed for recovery.

It's in New status.