

### Second Midterm Exam

- This test contains 9 questions worth a total of 73 points.
- Question 9 requires programming and is worth 20 points.
- You have 50 minutes to complete this exam.
- You may **not** use your text, notes, or any other reference material.
- Do not turn this page until instructed to do so.

### Test Taking Advice

- If you can't answer a question, move on and come back to it later. I often hear comments like this, "I spent 40 minutes working on this 10 point problem and left 30 points worth of problems blank."
- If you have time left over, use it to review your answers. Students who turn tests in early often make trivial mistakes that they would catch if they went back over their answers. Some of my questions are difficult, go back and make sure you understood all the questions.
- Some questions may have **multiple parts**. Students often lose points because they don't answer all the parts of a question.

1. (6 points) In a system that uses paging, describe what happens to cause a segmentation fault and when the OS recognizes it has occurred.

A program accesses a memory location that is not w/in its logical address space. The OS recognizes this at run time when the CPU requests the out of bounds address.

2. (6 points) What is a page fault? Describe what happens when there is a page fault.

A page fault occurs when a the CPU requests to access a memory location on a page that is not currently in memory.

The page is found on the secondary storage and then loaded into an empty frame in memory. The page table is updated to point to the frame that contains the page. Now the page is ready to be accessed.

3. (6 points) Describe the memory fragmentation in a contiguous memory allocation scheme. Describe the memory fragmentation in a paging memory management scheme.

Contiguous memory allocation causes external fragmentation. The entire memory for the process (called the segment) must be loaded into memory for the process to run. Since each process has a different sized segment, it is hard to pack segments into memory. Often there is lots of unused space in memory but none of it large enough to hold a new process and thus the new process cannot be loaded.

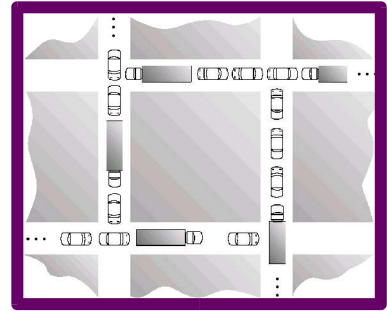
In paging memory management the memory needed by a process is broken into chunks called pages. All pages are the same size. Since each process is a different size, there is usually an unused portion of a process's last page. For example, N pages is too small for the process and N+1 pages is too large. So N+1 pages are used and the last page is only partially used. This is called internal fragmentation.

4. (6 points) How many page-faults will LRU page-replacement algorithms produce for the following string of page requests, assuming 3 page frames. Remember that all frames are initially empty. You must show your work to get any credit.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		4	4	4	0			1		1		1		
	0	0	0		0		0	0	3	3			3		0		0		
		1	1		3		2	2	2	2			2		2		9		

LRU = 12 page-faults

5. (8 points) Consider the traffic gridlock depicted below. List the four necessary conditions for deadlock and explain why each holds in this example.



1. Hold and wait: Cars are holding spaces in the road while waiting for a new space.
  2. Circular wait: There is a cycle of waiting cars.
  3. No preemption: Cars don't just disappear giving up their resource (that is, their space on the road).
  4. Mutual exclusion: Only one car can occupy any space on the road
6. (6 points) Give some code that could cause a race condition. What has to happen for this to be a race condition? What is the bad outcome?

P<sub>0</sub>:  
i++;

P<sub>1</sub>:  
i++;

The increment operator performs the following:

get value  
add one to value  
set variable to new value

If either P<sub>0</sub> or P<sub>1</sub> gets interrupted (is removed from executing and placed on the ready queue) after the get value and before the set, and then the other process execute the set, the value of i will be incorrect.

7. (6 points) Solutions to the critical section problem must satisfy which requirements? Briefly explain each.

**Mutual Exclusion:** If a process is executing in its critical section, then no other processes can be execution in the same critical section.

**Progress:** If no process is executing in its critical section and some process wants to enter, it should not be postponed indefinitely.

**Bounded Waiting:** There exists a bound on the numbers of times a process will be kept waiting while other process *cut* in front of it.

8. (9 points) Describe the three different types of address binding: compile time, load time, and run time.

**Compile time:** The compiler inserts physical memory address in its output. This requires that the compiler know where a program will reside in memory at compile time.

**Load time:** The compiler generate code that does not contain physical memory address. Instead this code contains relocatable addresses. When the program is loaded into memory, the relocatable addresses are replaced with physical addresses.

**Run time:** The compiler generates code with relocatable address. When the program executes, the relocatable addresses are converted into physical addresses. Most general purpose operating systems use this method.

9. (20 points) The Chico Speedway has a program to attract new fans to its destruction derby (this is a car race where the cars try to disable the other cars by crashing into them). After each car race fans are allowed to run onto the track and collect parts that have fallen off the cars. This program poses a serious safety problem: if fans and race cars are allowed on the track at the same time, fans might get hurt. Write pseudo code to implement a solution to this problem, specifically write the code that fans and the cars will execute. Don't worry about scheduling the races, just write the code that will make sure the fans don't get run over. Assume that fans can look for parts before the first race and that fans might make multiple trips to the track between races. Use semaphores for synchronization and don't provide any error checking.

```
sem_t sem_track = 1;
sem_t sem_cars = 1;
sem_t sem_fans = 1;
sem_t sem_print_mutex = 1;

int num_cars = 0;
int num_fans = 0;

void get_print_mutex()
{
    int result = sem_wait(&sem_print_mutex);
    assert(!result);
}

void release_print_mutex()
{
    int result = sem_post(&sem_print_mutex);
    assert(!result);
}

void *
fan(void *arg)
{
    int fan_number = *((int *) arg);

    while (true)
    {
        int result = sem_wait(&sem_fans);
        assert(!result);
        num_fans++;
        if (num_fans == 1)
        {
            result = sem_wait(&sem_track);
```

```

        assert(!result);
    }

    get_print_mutex();
    cout << "fan " << fan_number << " entered track. "
         << num_fans << " fans on track." << endl;
    release_print_mutex();

    result = sem_post(&sem_fans);
    assert(!result);

    sleep(1 + random() % 3); // simulate collecting parts

    result = sem_wait(&sem_fans);
    assert(!result);
    num_fans--;
    if (num_fans == 0)
    {
        result = sem_post(&sem_track);
        assert(!result);
    }
    get_print_mutex();
    cout << "fan " << fan_number << " exited track. "
         << num_fans << " fans still on track." << endl;
    release_print_mutex();

    result = sem_post(&sem_fans);
    assert(!result);

    sleep(15); // simulate watching the race
}
}

void *
car(void *arg)
{
    int car_number = *((int *) arg);

    for (int race = 0; race < 5; race++)
    {
        int result = sem_wait(&sem_cars);
        assert(!result);
        num_cars++;
        if (num_cars == 1)
        {
            result = sem_wait(&sem_track);
            assert(!result);
        }

        get_print_mutex();
        cout << "car " << car_number << " entered track. "
             << num_cars << " cars on track." << endl;
        release_print_mutex();

        result = sem_post(&sem_cars);
        assert(!result);

        sleep(2 + random() % 2); // simulate the race

        result = sem_wait(&sem_cars);
        assert(!result);
        num_cars--;
    }
}

```

```
    if (num_cars == 0)
    {
        result = sem_post(&sem_track);
        assert(!result);
    }
    get_print_mutex();
    cout << "car " << car_number << " exited track. "
         << num_cars << " cars still on track." << endl;
    release_print_mutex();

    result = sem_post(&sem_cars);
    assert(!result);

    sleep(10); // simulate waiting for next race
}
}
```