

Name: \_\_\_\_\_

Work alone. Open book and notes.

1. Given the page reference string in the first row of each table below, show the page number occupying each page frame just AFTER the new page is loaded. The memory holds 3 page frames, each represented by a blank table cell below each page number in the bold-type reference string. Assume the replacement policy used is: When no page fault occurs, you need not show the memory contents. Assume the entire reference string repeats after the last page number shown.

a. **OPT [10 pts]:**

<b>1</b>	<b>2</b>	<b>1</b>	<b>3</b>	<b>4</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>4</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>1</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>
1	1		1	1			1				2			2				2
	2		2	2			3				3			3				4
			3	4			4				4			1				1

b. **LRU [10 pts]:**

<b>1</b>	<b>2</b>	<b>1</b>	<b>3</b>	<b>4</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>4</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>1</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>
1	1		1	1		1	1		1		1	1	4	4		4	3	3
	2		2	4		4	3		3		2	2	2	1		1	1	4
			3	3		2	2		4		3	3	3	3		2	2	2

Name: \_\_\_\_\_

2. 3 jobs are submitted at time 0. Each job requires total CPU time  $T$ , and requires  $X$  units of I/O delay every  $Y$  units of CPU time, where the values of  $T$ ,  $X$ , and  $Y$  for each job are given in the table below:

Job	T	X	Y
A	500	0	--
B	50	50	5
C	100	50	10

- a. Suppose each job is run alone (sequential execution). How long will each job run? [10 pts]

A: 500 B: 550 C: 600

- b. If the jobs are scheduled so they run sequentially, in an order that minimizes average response time, in what order will they run, and what average response time results?

Job order: A, B, C [5 pts]

Average response time:  $3200/3=1066.67$  [5 pts]

- c. Find a schedule that minimizes the finish-time of the entire set of jobs. The single CPU can switch from one job to another, at any time your schedule deems correct, with no time lost. You may assume that multiple jobs can perform I/O at the same time.

1. Schedule, or scheduling rule: [10 pts]

Run A on the CPU at lower priority than either B or C. A possible CPU schedule would be: (C5 B10 A50) repeat 10 times. B does its I/O while A is running, and C's I/O overlaps B's computation, and some of B's I/O.

2. Finish time: 650 [5 pts]

3. How do you know this is the smallest possible finish time? [5 pts]

The total CPU time all 3 jobs need is 650 units, so no total run time that is smaller can exist. This schedule achieves this runtime

Name: \_\_\_\_\_

3. A central resource manager controls allocation of a pool of 10 units of some resource R. The manager offers the following methods to requesting processes:
- bool IsOK(n): Returns 1 if it is OK to grant a request for n units to this process, and 0 otherwise;
  - alloc(n): Actually allocates n units of R to this process; Violates an ASSERT() if this is not possible.
  - release(): Releases all units of R owned by this process.

Processes that use this system never need more than 5 units of R, total. Each process issues a single release() call when it finishes.

- a. How should isOK(n) determine if allowing the new request is OK, in the sense that if the request is granted, no deadlock on R can result, no matter what the future sequence of requests for R are (consistent with what was said above)? [10 pts]

isOK must use a simplified version of the Banker's Algorithm. It pretends to grant the request, and then checks to see if, after granting the request, there is at least one process that can get the remainder of the 5 maximum units it will ever ask for.. If such a process exists, it can eventually finish. In this case, this will leave at least 5 units free, and this is enough to allow any other job to finish. (In the full Banker's Algorithm, you must check to see that there is a sequence of process-completions that eventually allow ALL processes to complete.)

- b. If isOK(n) grants a request whenever there are currently at least n units of R available, give an example showing how this can lead to deadlock. [10 pts]

Suppose 2 processes arrive, and each ask for 3 units. According to the rule, these requests will all be granted, leaving one free unit. But each of these jobs now needs 2 more units, and none of them can get this many. Since NONE of the processes can proceed they are deadlocked.

Name: \_\_\_\_\_

- c. Outline or write a method `grab(n)` which uses the methods described above to get `n` units of `R` allocated to the current process, waiting until this is possible before proceeding. Carefully show the needed synchronization operations, using semaphores, and possibly ordinary variables. The methods of the resource manager are all safe to run in parallel. Specify the routine “`quit()`” which calls `release()` in some detail, also. [20 pts]

```
Semaphore *Sleep = new Semaphore(0); //Part of Thread class
```

```
Semaphore *MemLock = new Semaphore(1); // global
```

```
List MemWait = new List(); // global
```

```
Void grab(int n) {
    MemLock->P();
    While (!isOK(n)) {
        MemLock->V();
        [MemWait->append(currentThread.Sleep);
        currentThread.Sleep->P();]
        MemLock->P();
    }
    alloc(n);
    MemLock->V();
}
```

```
[Void quit() {
    MemLock->P();
    Release();
    Semaphore *t = MemWait->remove();
    If (t != NULL) t->V();
    MemLock->V();
}]
```

The `MemLock` critical section is needed, because `isOK()` isn't specified to change any internal state of the memory manager (it should not ASSUME that because it granted a request, the process will actually allocate that number of units). If the state of the memory manager is allowed to change between the time an `isOK()` responds TRUE, and the time the process that issued the `isOK()` performs the corresponding `alloc()`, some other process could make a request, get a TRUE `isOK()`, and `alloc()` some of the units the first process needs. I'm also using `MemLock` to protect the `MemWait` list. Use of `MemLock` inside `quit()` is probably needed, to ensure that `release` can't be interrupted by an `isOK()` or `alloc()`, although I think you could lock the innards of these routines instead.

The loop inside `grab()` is a busy waiting loop, modified here to put the issuing process to sleep on a semaphore `Sleep`, “owned” by the issuing thread. Only that thread does `P()`

Name: \_\_\_\_\_

operations on its Sleep.. The address of the semaphore is first added to the end of MemWait list. As each process complete, one semaphore is removed from the front of MemWait, and V()'d. Hopefully, this gives each waiting process a turn at trying to run.

Correctly maintaining a lock surrounding BOTH the isOK() call and extending through the subsequent alloc(),is worth 10 points. A simple busy-waiting solution, which omits the stuff inside [..],is worth another 10 points. This solution, which sleeps instead of looping, is worth 25-30 points total.