

Operating Systems

Exam II

Name: _____

4 November 1999

1. President Williams wants to diversify Ithaca College's income portfolio and Stan Seltzer, the chair of the Math/Computer Science department, has convinced her that we could make a ton of money by developing an alternative operating system for PCs. This OS, actually just an improvement on UNIX, will be called ICnix (pronounced eye-see-nicks).

Stan has decided that we need to diversify the device driver manager of UNIX. In addition to character devices and block devices, we'll have *page* devices. A page device is a very large storage device that we can access with a *page_read* or a *page_write* system call. An application issues a *page_read* or a *page_write* command and the system will start reading in blocks of data and storing them in an OS buffer (it will store data from the device if its a read and from the application if its a write). After the first block is finished, the application can resume processing. The OS must continue to read in blocks of data, however, and store them in its OS buffer. Thereafter, when the application makes a *page_read* or a *page_write* call, the OS should check it's buffer for that application and, if the buffer contains information, return immediately. If the buffer does not contain data, the normal process should be followed.

a. Modify the UNIX device management process (as described in the chapter 8 handout) to add page devices. Draw a **diagram** and **describe** the steps necessary to complete the *page_read* and *page_write*. Start from the initial system call and continue through the final actions by the OS. Include all data structures that are used by the OS. Provide code fragments if you find it useful to explain what is happening.

b. Describe how you could use the **strategy** entry point in your device.

c. The purpose of this new device class is to speed up reads and writes to/from large storage devices. Will this new procedure provide a speed-up? Justify your answer.

2. Scheduling algorithms. Show the gantt chart and give the average turnaround and wait times for the series of processes below. The chart lists each process, its priority and its service time (for the first, second and third cpu burst and for its first and second I/O burst). Assume that all processes are in the ready list when you start. Ignore context switching time.

Assume that the scheduler uses a multilevel feedback queue. There will be two levels, one for CPU intensive foreground jobs (priority 1), one for I/O intensive foreground jobs (priority 2). When a process is blocked for more than 20 units for I/O, it is moved to the priority 2 queue. When it is subsequently blocked for less than 20 units, it is moved back to the priority 1 queue. Every process starts in the priority 1 queue. The priority 1 queue is scheduled using SJN. Priority 2 queue is scheduled by its outside priority (lower numbers mean higher priority). All algorithms are nonpreemptive. All jobs in the priority 1 queue must be done before any job in the priority 2 queue is run.

P_i	$\tau_1(p_i)$ priority	I/O ₁	$\tau_2(p_i)$	I/O ₂	$\tau_3(p_i)$	
0	80	10	120	15	70	3
1	50	25	30	15	30	1
2	35	30	40	35	35	4
3	70	5	65	25	80	2

3. Assume that you have a system with **user threads**, ie threads that are scheduled by a user-level program associated with the process (see page 146 of Nutt). The process itself will schedule its own threads and will have control over the threads. Extend the process state diagram (**figure 6.12 on page 148** of Nutt) to include user threads. Your diagram will show the various states that a thread can be in in addition to the various states that a process can be in. **Explain your diagram and justify any assumptions.**

4. Consider the “in the hanger” example from page 299 that shows an example of the compile-link-load process. The example is expanded below. On the attached pages, show how the various code segments produced by the compiler, linker, and loader would change. You can make up relative addresses for the code if you need to.

```
/* some variable declarations */
static int gVar;
static int *gpPtr;

/* some various instructions */
int proc B(int x, int *y )
{
    int i;

    /* other declarations and instructions */

    i = *y;
}
proc A( )
{
    int i;
    /* this is a pointer to a function. The variable name is
fpPtr, and it points to a function that takes an int and a
*int as parameters and returns an int */
    int (*fpPtr)(int, *int);
    /* more declarations and instructions */
    gVar = 7;
    i = gVar;
    gpPtr = malloc(sizeof(int));
    *gpPtr = i + 10;
    fpPtr = D; /* fpPtr points to function D which is defined
externally */
    putRec(gVar);
    proc C(gpPtr, fpPtr);
}
/* procedure C receives a pointer to an int and a pointer to a
function */
proc C (int *inVar, int (* inFnc)( int, *int))
{
    int i;
    i = 3;
    /* some random variable declarations and instructions */

    inFnc(i,inVar);
}
/* main program for this process */
main( )
{
    int i;
    proc A;
}
}
```