Name ______________________ Student No. ______________________

No aids allowed. Answer all questions on test paper. Use backs of sheets if necessary.

Total Marks: 65

[5] 1. What is the main advantage of *multiprogramming* ?

Multiprogramming makes efficient use of the CPU by overlapping the demands for the CPU and its I/O devices from various users. It attempts to increase CPU utilization by always having something for the CPU to execute.

[10] 2. Define the essential properties of the following three types of operating systems: *Batch*, *Time sharing*, and *Distributed*.

Batch: Jobs with similar needs are batched together and run through the computer as a group by an operator or automatic job sequencer. Performance is increased by attempting to keep CPU and I/O devices busy at all times through buffering, off-line operations, spooling, and multiprogramming. Batch is good for executing large jobs that need little interaction.

Time sharing: The system uses CPU scheduling and multiprogramming to provide economical interactive use of a system. The CPU switches rapidly from one user to another.

Distributed: The system distributes computations among several physical processors. The processors do not share memory or a clock. Instead, each processor has its own local memory. They communicate with each other through various communication lines, such as high-speed bus.

[5] 3. What are the differences between a *trap* and an *interrupt*? What is the use of each function?

An interrupt is a hardware-generated change of "flow" within the system. An interrupt handler is summoned to deal with the cause of the interrupt; control is then returned to the interrupted context and instruction. A trap is a software-generated interrupt. An interrupt can be used to signal the completion of an I/O to obviate the need for device polling. A trap can be used to call operating system routines or to catch arithmetic errors.

[10] 4. Give two reasons why *caches* are useful. What problems do they solve? What problems do they cause? If a cache can be made as large as the device for which it is caching (e.g., a cache as large as a hard disk), why not make it that large and eliminate the device altogether?

Caches are useful when two or more components need to exchange data, and the components perform transfers at different speeds. Caches solve the transfer problem by providing a buffer of intermediate speed between the components. If the fast device finds the data it needs in the cache, it need not wait for the slower device.

The data in the cache must be kept consistent with the data in the component. If a component has a data value change, and the datum is also in the cache, the cache must also be updated. This is especially a problem on multiprocessor systems where more than one process may be accessing a datum.

A component may be eliminated by an equal size cache, but only if: (a) the cache and the component have equivalent state-saving capacity (if the component can retain data after a power down, so must the cache), (b) they are similarly priced (faster memory tends to be more expensive).

[10] 5. List three important services provided by the operating system and describe them briefly. Point out for which services (from among the three you listed) it would not be possible for user-level programs to provide them, and why.

Program execution: The OS loads the contents (or sections) of a file into memory and begins its execution. A user-level program could not be trusted to properly allocate the CPU.

I/O operations: Disks, tapes, serial lines, and other services must be communicated with at a very low level. The user need only specify the device and the operation to perform on it, while the system converts that request into device- or controller-specific commands. User-level programs cannot be trusted to only access devices they should have access to and to only access them when they are otherwise unused.

File-system manipulation: There are many details in file creation, deletion, allocation, and naming that users should not have to perform. Blocks of disk space are used by files and must be tracked. Deleting a file requires removing the name file information and freeing the allocated blocks. Protections must be checked to assure proper file access. User programs could neither ensure adherence to protection methods nor be trusted to allocate only free blocks and deallocate blocks on file deletion.

[5] 6. Describe the actions a kernel takes to *context switch* between processes.

In general, the OS must save the state of the currently running process, and restore the state of the process scheduled to be run next. Saving the state of a process typically includes the values of all the CPU registers in addition to memory allocation. Context switches must also perform many architecture-specific operations, including flushing data and instruction caches.

[10] 7. Recall the bounded-buffer solution for the producer-consumer IPC problem:

```
10  #define BUFFER_SIZE 10
20  typedef struct { ... } item;
30  item buffer[BUFFER_SIZE];
40  int in = 0;
50  int out = 0;

60  while (1) {
70          while (((in + 1) % BUFFER_SIZE) == out)
80                  ; /* do nothing */
90          buffer[in] = nextProduced;
100         in = (in + 1) % BUFFER_SIZE;

110 while (1) {
120         while (in == out)
130                 ; /* do nothing */
140         nextConsumed = buffer[out];
150         out = (out + 1) % BUFFER_SIZE;
```

This scheme allows at most BUFFER_SIZE−1 items in the buffer at the same time. How would you modify the above "program" so that BUFFER_SIZE items can be in the buffer at the same time?

A simple solution is two add two boolean flags, EMPTY and FULL, initialize them to true, and false, respectively. Lines 70 and 120 would become:

```
while (FULL)
while (EMPTY)
```

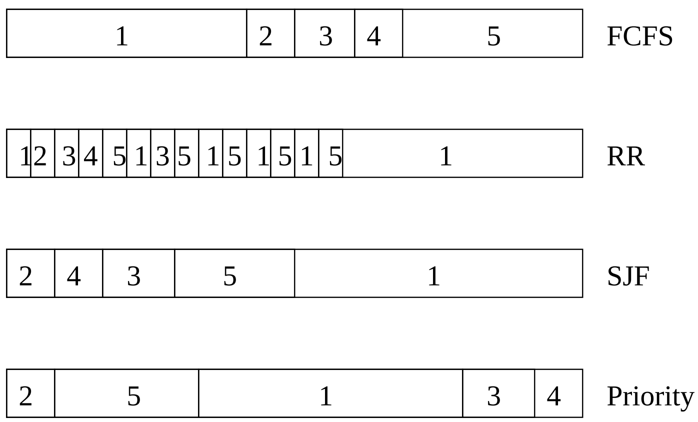
respectively. We would also add the following four lines:

```
105         EMPTY=false;
106         if (in == out) FULL=true;
155         FULL=false;
156         if (in == out) EMPTY=true;
```

[10] 8. Consider the following set of processes:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a nonpreemptive priority (a smaller priority number implies a higher priority), and RR (time slice = 1) scheduling. Explain briefly what FCFS, SJF, and RR are.



FCFS is a First-Come-First-Serve CPU scheduling algorithm. Incoming processes are put at the tail of the ready-queue; when a process terminates or makes an I/O request, it is relinquishes the CPU, and the next process is scheduled.

SJF is a Shortest-Job-First CPU scheduling algorithm. Processes are chosen from the ready-queue on the basis of the next shortest CPU burst.

RR is a Round-Robin CPU scheduling algorithm. Each process is given a time-slice, and the ready-queue is implemented as a circular-queue. After the time-slice of a process expires, it is put back in the ready-queue, and the next process is selected.