Full Name: ______________________

CS 450 Fall 2010

Midterm Exam

October 13, 2010

Instructions:

- This exam is closed-book, closed-notes.
- Keep your written answers concise and to-the-point. I reserve the right to deduct points for needless verbiage.
- Write your full name on the front, and make sure that your exam is not missing any sheets.
- Good luck!

Problem 1	(/10) :	
Problem 2	(/9) :	
Problem 3	(/12) :	
Problem 4	(/6) :	
Problem 5	(/6) :	
Problem 6	(/6) :	
TOTAL	(/49) :	

Problem 1. (10 points):

Multiple choice. For each of the following multiple choice problems, choose the *single best* answer by circling (not checking, crossing, or underlining — **circling!**) its corresponding letter.

1. A valid argument against a monolithic kernel design is that, in such an architecture:

   (a) message passing between kernel modules is inefficient

   (b) a bug in one kernel module (e.g., a device driver) may crash the entire kernel

   (c) an increased number of context switches are required

   (d) system calls require hardware intervention

2. On a modern operating system that is continually running highly interactive, "bursty" processes, which of the following is arguably the *least* important scheduler metric to optimize?

   (a) response time

   (b) average wait time

   (c) process turnaround time

   (d) CPU utilization

3. A *preemptive* scheduler is best identified by its ability to carry out this state transition:

   (a) ready → running

   (b) running → blocked

   (c) blocked → running

   (d) running → ready

4. Which of the following events would *most likely* cause a multilevel feedback queue scheduler to move a given process from a lower level RR queue (q=N) to a higher level RR queue (q=M – assume $M < N$)?

   (a) the current CPU burst is preempted before it can complete

   (b) the current CPU burst is preempted multiple times before completing

   (c) the current CPU burst completes in less than N time

   (d) the current CPU burst completes in less than M time

5. Which of the following scheduling algorithms is *not* prone to starvation?

   (a) selfish round-robin

   (b) preemptive shortest job first

   (c) first come, first served

   (d) multilevel queue

Problem 2. (9 points):

The following are CPU burst durations for five processes.

Process	CPU Burst
P_0	5
P_1	2
P_2	1
P_3	4
P_4	3

Complete the following table with the individual process and average waiting times for each of the indicated scheduling policies. Assume that all processes arrive at time 0, and, for the purposes of FCFS and RR, are initially ordered as shown in the wait queue. Ignore context switch overhead.

	Wait Times					
Scheduling policy	P_0	P_1	P_2	P_3	P_4	Avg wait time
First-Come First-Served						
Non-preemptive Shortest Job First						
Round-Robin (quantum=2)						

Feel free to use the space below for your work. You do *not* need to draw a Gantt diagram for each scheduling algorithm (unless it helps you).

Problem 3. (12 points):

The following configuration file is used to set up an experiment in the UTSA scheduling simulator:

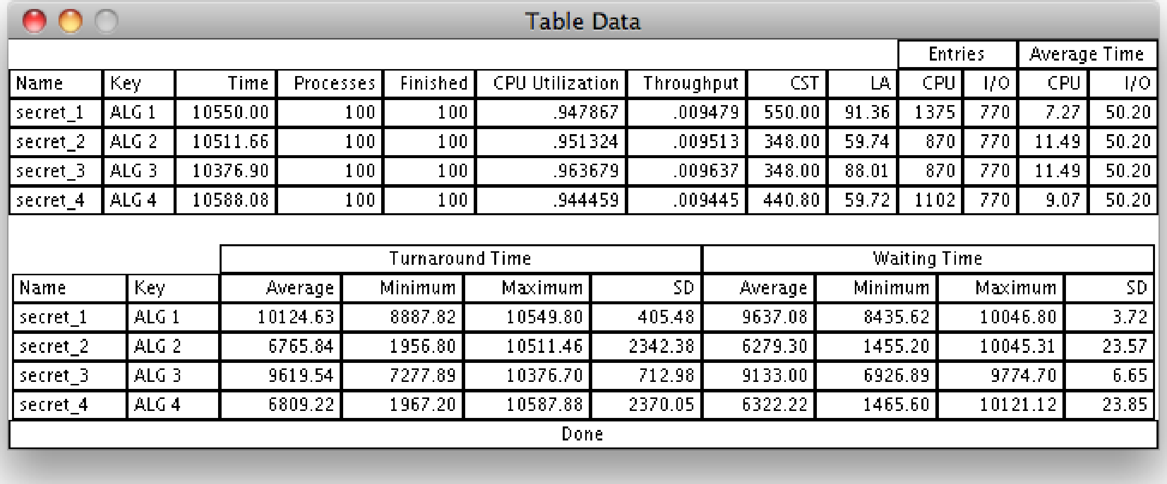
```
cstin        0.2
cstout       0.2

numprocs     100
firstarrival 0
interarrival constant 0
duration     constant 100
cpuburst     uniform 5 20
ioburst      constant 50
```

The run file (not shown) specifies the following four different scheduling algorithms – listed out of order – to be used with the configuration above:

- Shortest Job First
- Preemptive Shortest Job First
- Round Robin (q=10)
- Round Robin (q=20)

The tabulated output of the simulation is shown below:



									Entries		Average Time	
Name	Key	Time	Processes	Finished	CPU Utilization	Throughput	CST	LA	CPU	I/O	CPU	I/O
secret_1	ALG 1	10550.00	100	100	.947867	.009479	550.00	91.36	1375	770	7.27	50.20
secret_2	ALG 2	10511.66	100	100	.951324	.009513	348.00	59.74	870	770	11.49	50.20
secret_3	ALG 3	10376.90	100	100	.963679	.009637	348.00	88.01	870	770	11.49	50.20
secret_4	ALG 4	10588.08	100	100	.944459	.009445	440.80	59.72	1102	770	9.07	50.20

		Turnaround Time				Waiting Time			
Name	Key	Average	Minimum	Maximum	SD	Average	Minimum	Maximum	SD
secret_1	ALG 1	10124.63	8887.82	10549.80	405.48	9637.08	8435.62	10046.80	3.72
secret_2	ALG 2	6765.84	1956.80	10511.46	2342.38	6279.30	1455.20	10045.31	23.57
secret_3	ALG 3	9619.54	7277.89	10376.70	712.98	9133.00	6926.89	9774.70	6.65
secret_4	ALG 4	6809.22	1967.20	10587.88	2370.05	6322.22	1465.60	10121.12	23.85

Your job is to solve the riddle: which algorithms correspond to ALG 1, 2, 3, and 4?

Circle your guesses on the following page. For full credit, you must justify your answers.

- ALG 1: `SJF` / `PSJF` / `RR(10)` / `RR(20)`

  Explanation:

- ALG 2: `SJF` / `PSJF` / `RR(10)` / `RR(20)`

  Explanation:

- ALG 3: `SJF` / `PSJF` / `RR(10)` / `RR(20)`

  Explanation:

- ALG 4: `SJF` / `PSJF` / `RR(10)` / `RR(20)`

  Explanation:

Problem 4. (6 points):

Consider the following lines taken from the `retu` assembly function:

```
0743     mov (sp),KISA6
0744     mov $_u,r0
0745  1:
0746     mov (r0)+,sp
0747     mov (r0)+,r5
```

What is the purpose of line **0743**? In particular, how is the kernel virtual address space affected?

What is the combined effect of lines **0744-0747**? Why is it crucial for these lines to be executed after line **0743** (e.g., what would happen if they weren't)?

Problem 5. (6 points):

Consider the following v6 snippet:

```
n = -1;
for (rp = &proc[0]; rp < &proc[NPROC]; rp++)
    if (rp->p_stat == SRUN && (rp->p_flag&SLOAD) == 0 && rp->p_time > n) {
        p1 = rp;
        n = rp->p_time;
    }

if (n == -1) {
    runout++;
    sleep(&runout, PSWP);
    goto loop;
}
```

What is the loop searching for? And more specifically, what type of scheduling is being performed?

What does the kernel do after exiting the loop, upon first encountering this code during the startup sequence? Why?

Problem 6. (6 points):

We first encounter a call to `savu` in the `newproc` function – the context is shown below:

```
savu(u.u_rsav);
rpp = p;
u.u_procp = rpp;
rip = up;
n = rip->p_size;
a1 = rip->p_addr;
rpp->p_size = n;
a2 = malloc(coremap, n);
```

Note that `up` refers to the "parent" process, while `p` refers to the new process being created.

Why is it so important that `savu` be called here, within the `newproc` function, and *before* the call to `malloc` (and the subsequent initialization of the new process's user data area)? Explain.