OS SAMPLE Midterm Exam
Prof. Golden G. Richard III

Name: ____________________________

ANSWER 4 OF THE 6 QUESTIONS. PUT A LARGE "X" THROUGH THE QUESTION YOU DO NOT WANT GRADED.

[25, Round Robin Process Scheduling]
1.

Assume that an average process runs for time *T* before blocking for I/O (that is, the CPU burst is *T*). Processes are scheduled using a round-robin strategy with quantum *Q*. The context switch time is *S*, which is pure overhead. The quantum *Q* does not include this overhead.

CPU efficiency is defined as follows:

$$\textbf{efficiency} = \frac{\textbf{(CPU time provided to processes)}}{\textbf{(CPU time provided to processes + overhead).}}$$

Note: in the following, assume that the ready queue is never empty.

For each of the following scenarios, give a formula for CPU efficiency in terms of *Q, S,* and *T*. Be sure to simplify each formula as much as possible.

(1) $Q = \infty$

(2) $S < Q < T$

(3) $Q = S$

[25, Efficient fork()-ing]
2.

What is fork()?

What is copy on write?

Describe how copy on write can be used to efficiently implement fork().

[25, Semaphores]
3.

Making Tofu Sandwiches

Consider a system with 3 agents (threads), each of which makes tofu sandwiches, and one controller (also a thread), which provides some of the ingredients needed to make sandwiches.

Here are some basic rules:

1. To make a tofu sandwich, bread, tofu, and mustard are required.
2. One of the sandwich agents possesses an infinite supply of tofu. Another possesses an infinite supply of bread, and the third possesses an infinite supply of mustard. Thus each agent is missing two of the ingredients necessary to make a sandwich.
3. The controller possesses an infinite supply of all ingredients.

The threads behave like this:

1. The controller places two of the necessary ingredients, chosen at random, on a table.
2. The sandwich maker who has the remaining ingredient then makes a tofu sandwich and signals the controller that the sandwich is complete.
3. The procedure is repeated from step 1.

Synchronization is required for this system to behave properly. The following is an initial attempt to solve the problem using semaphores:

```
ingredient : array[0..2] of semaphore = (0,0,0); // initially all 0
controller : semaphore = 1;

controller                        sandwich guy needing r and s (0<=r,s<=2)
repeat                             repeat
  randomly pick unique i, j          wait(ingredient[r]);
  wait(contoller);                   wait(ingredient[s]);
  signal(ingredient[i]);             // make a sandwich
  signal(ingredient[j]);             signal(controller);
forever;                           forever;
```

Does this solution work? If so, provide a convincing argument that it works. If not, state why and then write a correct solution. Put your answer on the following page.

[Space for answer for 3]

[25, A Monitor Variation]
4.

Imagine a monitor implemention that replaces condition variables (and the associated operations wait() and signal()) with a single operation ***wait_for(B)***, where *B* is an arbitrary Boolean expression based on the variables declared globally in the monitor.

Example: ***wait_for(number_of_readers = 0)***

Using this new type of monitor, write a solution to the producer-consumer problem, where a producer thread and a consumer thread store and remove items from a buffer of length *n*.

[25, Traditional Monitors]
5.

Write a solution to the *sleeping barber's* problem using a monitor. The rules are:

- There is a single barber in a shop, capable of providing one haircut at a time.
- There are *n* chairs in the barbershop, where customers can wait on the barber.
- If there are no customers present, the barber goes to sleep.
- When a customer enters the shop:
    - If the barber is asleep, he wakes up the barber and gets a haircut.
    - If the barber is busy and there is an available chair, then he sits and waits for a haircut.
    - If all chairs are full, the customer leaves and does not get a haircut—in this case, the customer does <u>not</u> block!

The barber executes the following loop, where PrepareToCut is a monitor procedure:

```
loop
    PrepareToCut();     // blocks if there are no customers!
    // give a haircut
forever;
```

Each customer thread will execute the following loop, where TryToGetAHaircut is a monitor procedure:

```
loop
    // the following returns immediately if no available chairs, otherwise blocks
    // until barber becomes available
    TryToGetAHaircut();
    // …
    // do some other things, don't want to get a haircut too often!
    // …
forever;
```

[25, Process Scheduling: SJF, SRTF]
6.

The Shortest Job First and Preemptive Shortest Job First (also called Shortest Remaining Time First) scheduling algorithms have a very favorable property. What is it? They also have an unfavorable property that causes them not to be used in general purpose systems. What is it?

SOLUTION STARTS ON NEXT PAGE!

OS SAMPLE Midterm Exam SOLUTION
Prof. Golden G. Richard III

- Please read the questions carefully! "Answering" the wrong question does no good!
- Don't hesitate to ask for clarification if you're confused about the intent of a question.
- Total points = 100

ANSWER 4 OF THE 6 QUESTIONS. PUT A LARGE "X" THROUGH THE QUESTION YOU DO NOT WANT GRADED.

[25, Round Robin Process Scheduling]
1.

Assume that an average process runs for time *T* before blocking for I/O (that is, the CPU burst is *T*). Processes are scheduled using a round-robin strategy with quantum *Q*. The context switch time is *S*, which is pure overhead. The quantum *Q* does not include this overhead.

CPU efficiency is defined as follows:

$$\textbf{efficiency} = \frac{\textbf{(CPU time provided to processes)}}{\textbf{(CPU time provided to processes + overhead).}}$$

Note: in the following, assume that the ready queue is never empty.

For each of the following scenarios, give a formula for CPU efficiency in terms of *Q, S,* and *T*. Be sure to simplify each formula as much as possible.

(2) $Q = \infty$

A process can execute for as long as it needs the CPU—it is never preempted. Thus for each process execution, only one context switch is required—at the point where the process relinquishes the CPU. Thus:

$$efficiency = \frac{T}{T+S}$$

(4) $S < Q < T$

In this case, a process needs more CPU time in each burst than is allowed by the quantum *Q;* this means there will be multiple context switches per CPU burst. There are two ways of looking at this one. One is to count the number of context switches per CPU burst:

$$\boldsymbol{efficiency = \frac{T}{S\lceil T/Q \rceil + T}}$$

Another (simpler) way is to notice that for each quantum (or fractional quantum), you pay one context switch. In this case,

$$efficiency = \frac{Q}{Q+S}$$

(5) $Q = S$

$$efficiency = \frac{T}{T + S\lceil T/S \rceil}$$

Your intuition is probably than in the Q=S case, that efficiency is about 50%. This is true, and the equation above simplifies to ½ if S divides T evenly. Or use Q / 2Q !

[25, Efficient fork()-ing]

2.

What is fork()?

See your notes. It's the Unix mechanism for process creation, accessed through the fork() system call.

What is copy on write?

Copy on write is a mechanism for page sharing. The intention is to share pages of memory between processes which to the processes must appear private—that is, changes made to the page by one process should not be visible to the others. As long as no process modifies the page, nothing special needs to be done. When one process modifies the page, a private copy of the page is made and the change is applied only to that copy.

Describe how copy on write can be used to efficiently implement fork().

Copy on write allows process creation without duplicating the entire address space. Instead, creation of additional copies of pages owned by the parent process can be deferred until modifications are made (by either the parent <u>or</u> the child).

[25, Semaphores]
3.

Making Tofu Sandwiches

Consider a system with 3 agents (threads), each of which makes tofu sandwiches, and one controller (also a thread), which provides some of the ingredients needed to make sandwiches.

Here are some basic rules:

- To make a tofu sandwich, bread, tofu, and mustard are required.
- One of the sandwich agents possesses an infinite supply of tofu. Another possesses an infinite supply of bread, and the third possesses an infinite supply of mustard. Thus each agent is missing two of the ingredients necessary to make a sandwich.
- The controller possesses an infinite supply of all ingredients.

The threads behave like this:

1. The controller places two of the necessary ingredients, chosen at random, on a table.
2. The sandwich maker who has the remaining ingredient then makes a tofu sandwich and signals the controller that the sandwich is complete.
3. The procedure is repeated from step 1.

Synchronization is required for this system to behave properly. The following is an initial attempt to solve the problem using semaphores:

```
ingredient : array[0..2] of semaphore = (0,0,0); // initially all 0
controller : semaphore = 1;

controller                         sandwich guy needing r and s (0<=r,s<=2)
repeat                              repeat
  randomly pick unique i, j           wait(ingredient[r]);
  wait(controller);                   wait(ingredient[s]);
  signal(ingredient[i]);              // make a sandwich
  signal(ingredient[j]);              signal(controller);
forever;                            forever;
```

Does this solution work? If so, provide a convincing argument that it works. If not, state why and then write a correct solution. Put your answer on the following page.

[Space for answer for 3]

Semaphore controller=1, havemustard=0, havetofu=0, havebread=0;

CONTROLLER:

```
loop
  R = random(3);
  if (R == 1)
     // deposit tofu, mustard
     signal(havebread);
  else if (R == 2)
     // deposit tofu, bread
     signal(havemustard);
  else if (R == 3)
     // deposit bread, mustard
     signal(havetofu);
  end if
  wait(controller);
end loop;
```

HAVEMUSTARD: **(others similar…)**

```
loop
     wait(havemustard);
     // create sandwich
     signal(controller);
end loop;
```

[25, A Monitor Variation]
4.

Imagine a monitor implemention that replaces condition variables (and the associated operations wait() and signal()) with a single operation ***wait_for(B)***, where *B* is an arbitrary Boolean expression based on the variables declared globally in the monitor.

Example: ***wait_for(number_of_readers = 0)***

Using this new type of monitor, write a solution to the producer-consumer problem, where a producer thread and a consumer thread store and remove items from a buffer of length *n*.

```
Private data:   buffer(1..n) of some type;


entry produce(item) {
    wait_for(numitems < n);
    add_to_buffer(item);
    numitems++;
}

entry consume {
    wait_for(num_items > 0)
    item = remove_from_buffer();
    numitems--;
    return item;
}
```

That's it.

[25, Traditional Monitors]
5.

Write a solution to the *sleeping barber's* problem using a monitor. The rules are:

- There is a single barber in a shop, capable of providing one haircut at a time.
- There are *n* chairs in the barbershop, where customers can wait on the barber.
- If there are no customers present, the barber goes to sleep.
- When a customer enters the shop:
    - If the barber is asleep, he wakes up the barber and gets a haircut.
    - If the barber is busy and there is an available chair, then he sits and waits for a haircut.
    - If all chairs are full, the customer leaves and does not get a haircut—in this case, the customer does <u>not</u> block!

The barber executes the following loop, where PrepareToCut is a monitor procedure:

```
loop
    PrepareToCut();     // blocks if there are no customers!
    // give a haircut
forever;
```

Each customer thread will execute the following loop, where TryToGetAHaircut is a monitor procedure:

```
loop
    // the following returns immediately if no available chairs, otherwise blocks
    // until barber becomes available
    TryToGetAHaircut();
    // …
    // do some other things, don't want to get a haircut too often!
    // …
forever;
```

```
Monitor {
        Barber, Customer : condition;
        chairsavail: int = n;

        PrepareToCut() {
                if (chairsavail == n) {
                        Barber.wait();
                }
                chairsavail++;
                Customer.signal();
        }

        TryToGetAHaircut() {
                if (chairsavail == 0) {
                        // no room, just leave
                }
                else {
                        chairsavail--;
                        Barber.signal();  // wake up barber
                        Customer.wait();   // have to wait for the barber's attention
                }
        }
```

[25, Process Scheduling: SJF, SRTF]
6.

The Shortest Job First and Preemptive Shortest Job First (also called Shortest Remaining Time First) scheduling algorithms have a very favorable property. What is it? They also have an unfavorable property that causes them not to be used in general purpose systems. What is it?

Favorable:

Potentially better response time for interactive processes, particularly for the preemptive version.

Unfavorable:

Besides the fact that SJF (the non-preemptive variant) is susceptible to large processes hogging the CPU, there's a major problem with both algorithms.   They require knowledge about the length of CPU bursts.   This knowledge is difficult to obtain (and more difficult as more accuracy is required).