Midterm #1

CMSC 412
Operating Systems
Fall 2004

October 18, 2004

Guidelines

This exam has 7 pages (including this one); make sure you have them all. Put your name on each page before starting the exam. Write your answers directly on the exam sheets, using the back of the page as necessary. Bring your exam to the front when you are finished. Please be as quiet as possible.

If you have a question, raise your hand and I will come to you. However, to minimize interruptions, you may only do this once for the entire exam. Therefore, wait until you are sure you don't have any more questions before raising your hand. Errors on the exam will be posted on the board as they are discovered. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

Use good test-taking strategy: read through the whole exam first, and answer the questions easiest for you first.

Question	Points	Score
1	20	
2	20	
3	25	
4	25	
5	10	
Total	100	

1. (Memory Protection, 20 points) Part of the job of an operating system is to protect the memory of the kernel from processes, and protect each process's memory from other processes. One way to do this is via base and limit registers, a more sophisticated version of which is provided by the i386's *segments*. This question concerns how you use segments to implement memory protection in GeekOS for project 2.

   - (2 points) What is the difference between a linear and logical address on the x86 as we used them in project 2?

     Answer:

     A linear address is a physical address, while a logical address is a user-space address. The latter will be translated to a physical address via the segmentation hardware.

   - (6 points) `User_Context` defines `entryAddr` to be the code entry point for the context (see Figure 1). Is it a linear or logical address? Why?

     Answer:

     The `entryAddr` is the address of the first instruction execute when the process starts running. It is a logical address. When creating a process, the stack is set up so that when the process is switched to, it will start running at that address. It must be a logical address because it will be used while the process is running, and thus must be legal for that process's logical memory.

   - (6 points) `User_Context` also defines `memory`, along with an associated size. Is this a linear or logical address? Why?

     Answer:

     The `memory` field is a linear address that refers to the memory allocated by the kernel to store the code, data, stack, etc. for the process. It is linear for two reasons. First, we need it to perform address translation for system calls (as when implementing `Copy_To_User`). Second, we need it to be able to free the process's memory when it terminates, by giving it back to the kernel allocator with `Free`.

   - (6 points) Explain what the `Copy_To_User` and `Copy_From_User` routines you implemented do in GeekOS. Using the terminology of linear and logical address, explain why they are needed.

     Answer:

     These routines are used to copy buffers of memory from the kernel address space to the user address space and vice versa, as part of system calls. System calls that pass pointers to buffers (like PrintString) will refer to logical addresses, and these must be translated to linear addresses to access them within the kernel. Conversely, buffers in user space into which the kernel must copy its results must be similarly translated.

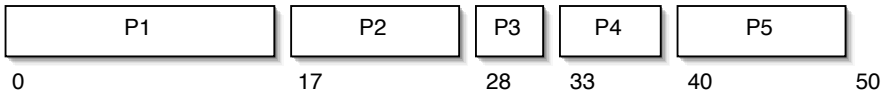
2. (Scheduling, 20 points) For this problem you are going to consider the performance of different scheduling algorithms for the following workload:

Process	Arrival time	Burst Time
p_1	0	16
p_2	0	10
p_3	6	4
p_4	7	6
p_5	8	10

Draw a Gantt chart to illustrate how these processes would be scheduled using First-Come-First-Serve (FCFS), Shortest-Job-First (SJF), and Round Robin (RR) scheduling. *Include context-switching time in your picture.* If there is a tie (you could choose either of two processes legally) pick the process with the lowest number (i.e. p_i over p_j when $i < j$). You can assume that a context-switch takes 1 time unit, and that the quantum is set to 5 time units. For each algorithm, calculate the waiting time for each process.

- FCFS:

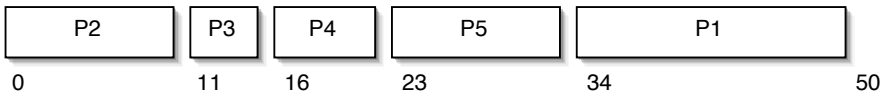
  Answer:

  The waiting times are simply the start times minus the arrival times.
  $p_1 = 0, p_2 = 17, p_3 = (28 - 6 = 22), p_4 = (33 - 7 = 26), p_5 = (40 - 8 = 32)$

- SJF:

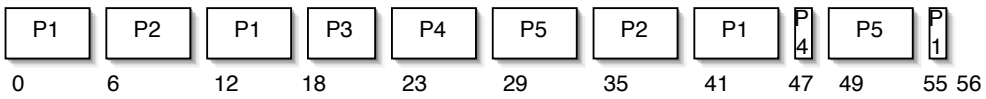
  Answer:

  The waiting times are simply the start times minus the arrival times.
  $p_1 = 34, p_2 = 0, p_3 = (11 - 6 = 5), p_4 = (16 - 7 = 9), p_5 = (23 - 8 = 15)$

- RR:

  Answer:

  The waiting times are calculated by taking the finishing time of the process, and subtracting from it the burst time and the arrival time.
  $p_1 = (56 - 16 - 0 = 40), p_2 = (40 - 10 - 0 = 30), p_3 = (22 - 4 - 6 = 12), p_4 = (48 - 6 - 7 = 35), p_5 = (54 - 10 - 8 = 36)$

3. (Synchronization, 25 points) This question concerns a number of aspects of synchronization.

- A reasonable solution to the critical section problem has three properties: mutual exclusion, progress, and bounded waiting. Explain each below (4 points each):
  - Mutual Exclusion.

    Answer:

    Only one process should be in its critical section at any given time.

  - Progress

    Answer:

    A process should not be prevented from entering its critical section if no other processes are in theirs.

  - Bounded Waiting

    Answer:

    Once a process has started to wait to enter the critical section, it should be able to get in eventually.

- (8 points) Show how semaphores can be used to solve the critical section problem, and explain why your solution satisifes the above three criteria.

  Answer:

  Define a semaphore `mutex` *initalized to* 1*. When a process wishes to enter its critical section, it does:*

  ```
  wait(&mutex);
  // CS here
  signal(&mutex);
  ```

  Since the mutex is initialized to 1*, the first process to* `wait` *on it will be able to enter the critical section, but subsequent processes will block, satisfying mutual exclusion. When a process exiting the critical section signals the mutex, it unblocks a process on the semaphore, which is then allowed to enter, satisfying progress. Finally, since processes on the semaphore's blocked queue are serviced first-come-first-serve, the waiting time is bounded.*

- (5 points) Name a datastructure in GeekOS that requires synchronization to protect it from race conditions. Explain why it needs synchronization, and indicate what kind of synchronization it uses.

  Answer:

  One possibility is the ready queue (`s_runQueue`*), since it could be accessed and possibly modified by multiple threads simultaneously, it requires synchronization. GeekOS disables interrupts to prevent CPU preemption while accessing the queue.*

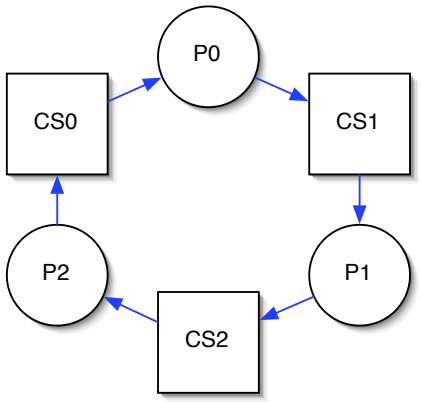
4. (Deadlock, 25 points) In the dining philosophers problem, there are n philosophers and n chopsticks. Any philosopher p_i must acquire two chopsticks before it can eat, one to its left (chopstick i) and one to its right (chopstick $i+1$ mod n). When it is finished eating, it returns the chopsticks and thinks.

   The text presents the following solution to this problem. We represent each chopstick as a semaphore, initialized to 1, storing all chopsticks in the array `chopstick` of size n. The code for philosopher p_i is as follows (recall that % in C represents the modulus operator):

```
while (1) {
  wait(chopstick[i]);
  wait(chopstick[(i+1) % n]);
  ... eat ...
  signal(chopstick[i]);
  signal(chopstick[(i+1) % n]);
  ... think ...
}
```

   - (10 points) It is asserted in the text that this solution could lead to deadlock. Draw a resource allocation graph that demonstrates a deadlocked state, for $n = 3$.

     Answer:



   - There are four necessary conditions for deadlock. *Deadlock prevention* is a technique that prevents one of these necessary conditions from ever manifesting, and thus prevents the possibility of deadlock.
     - (5 points) Pick one of these conditions, and explain how preventing it would disallow the graph you have drawn above.

       Answer:

       Two possible answers.

       (a) Hold and Wait. The problem is that a philosopher can be holding one chopstick while it is waiting for another. Without this condition, we wouldn't be able to have an edge going into and out of any process p_i; edges would either go in or go out, and thus prevent a cycle.

       (b) Circular Waiting. Avoiding circular waiting would prevent the cycle, since resources would have to be acquired in order. No process could acquire a resource earlier in the order if it holds one later in the order.

(over)

– (10 points) Give a new solution to the problem that is deadlock-free using the condition you have described above. If you must use semaphore primitives other than `wait` and `signal`, define the primitives and briefly justify that you could implement them.

Answer:

(a) Hold and Wait: acquire both resources at the same time. To do this, we have to use `trywait`*, which returns* 1 *if the caller would block were the semaphore to be decremented.*

```
while (1) {
  done = 0;
  while (!done) {
    wait(chopstick[i]);
    if (trywait(chopstick[(i+1) % n])) {
      signal(chopstick[i]);
    } else { done = 1; }
  }
  ... eat ...
  signal(chopstick[i]);
  signal(chopstick[(i+1) % n]);
  ... think ...
}
```

(b) Circular Waiting: force the resources to be acquired in the order they appear in the array.

```
while (1) {
  if (i == (n-1)) {
    wait(chopstick[0]);
    wait(chopstick[i]);
  }
  else {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % n]);
  }
  ... eat ...
  signal(chopstick[i]);
  signal(chopstick[(i+1) % n]);
  ... think ...
}
```

5. (Miscellaneous, 10 points)

- (5 points) What is a buffer overflow? How does Cyclone prevent buffer overflows?

  Answer:

  A buffer overflow occurs when a program writes outside the bounds of an array ("overflows" the bounds). This can lead to security holes. Cyclone prevents this by using a combination of dynamic and static checks. For example, writing to a "fat" pointer checks at run time that the pointer is within the bounds of a legal array.

- (5 points) What is the difference between a process being in the *running* state and in the *ready* state? What would cause a process to go from *running* to *ready*, and what would cause it to go from *ready* to *running*?

  Answer:

  A process that is running is using the CPU, while one that is ready is waiting to be scheduled to use it (it is in the ready queue*). A process goes from running to ready when it is* preempted*, when its quantum expires. A process goes from ready to running when the OS removes it from the ready queue and schedules it to run.*

```
struct User_Context {
    /* We need one LDT entry each for user code and data segments. */
#define NUM_USER_LDT_ENTRIES 2

    /*
     * Each user context contains a local descriptor table with
     * just enough room for one code and one data segment
     * describing the process's memory.
     */
    struct Segment_Descriptor ldt[NUM_USER_LDT_ENTRIES];
    struct Segment_Descriptor* ldtDescriptor;

    /* The memory space used by the process. */
    char* memory;
    ulong_t size;

    /* Selector for the LDT's descriptor in the GDT */
    ushort_t ldtSelector;

    /*
     * Selectors for the user context's code and data segments
     * (which reside in its LDT)
     */
    ushort_t csSelector;
    ushort_t dsSelector;

    /* Code entry point */
    ulong_t entryAddr;

    /* Address of argument block in user memory */
    ulong_t argBlockAddr;

    /* Initial stack pointer */
    ulong_t stackPointerAddr;
};
```

Figure 1: `User_Context` structure in GeekOS