1. Basic Concepts

(a) [10 points] Explain interrupts and traps, and provide a detailed account of the procedure that an operating system handles an interrupt.

**Answer:** An interrupt is an event that requires the attention of the operating system. These events include the completion of an I/O, a key press, the alarm clock going off, division by zero, accessing a memory area that does not belong to the running program, and so on. A trap is an interrupt generated by software.

When an interrupt occurs, the following steps will take place to handle the interrupt:

- The executing program is suspended and control is transferred to the operating system. Mode switch may be needed.
- A general routine in the operating system examines the received interrupt and calls the interrupt-specific handler.
- After the interrupt is served, a context switch transfers control back to a suspended process. Of course, mode switch may be needed.

See p. 8 of our textbook and class notes.

(b) [10 points] Define the meaning of mechanism and policy in the separation of mechanism and policy principle.

**Answer:** Mechanisms determine how to do something, and policies determine what will be done. They are usually separated for flexibility.

See Section 2.6.2 of text and class notes details.

(c) [10 points] What is the differences between A & B and A | B in Unix, where A and B are two binary executables?

**Answer:** In “A & B” programs A and B are run concurrently; however, A runs in background and B in foreground. In “A | B” programs A and B are run concurrently so that the standard output of A is connected to the standard input of B (i.e., A pipes its output to B). Note that, by default, A takes its standard input from the standard input (i.e., the keyboard) and B sends its standard output to standard output (i.e., screen).

This is a problem in a weekly reading list.

2. Fundamentals of Processes and Threads

(a) [10 points] What is a context? Provide a detail description of all activities of a context switch.

**Answer:** A process needs some system resources to run successfully. These system resources include process ID, registers, memory areas (for instructions, local and global variables, stack and so on), various tables (i.e., process table), and a program counter to indicate the next instruction to be executed. They constitute the environment or context of a process. The steps of switching process A to process B are as follows:

- Suspend A’s execution
- Transfer the control to the CPU scheduler. A CPU mode switch may be needed.
- Save A’s context to its PCB and other tables.
- Load B’s context to register, etc.
- Resume B’s execution of the instruction at B’s program counter. A CPU mode switch may be needed.

This was covered in class.
(b) [15 points] Draw the state diagram of a process from its creation to termination, including all transitions, and briefly elaborate every state and every transition.

**Answer:** The following state diagram is taken from my class note and was discussed in class. Fill in the elaboration for each state and transition by yourself.

See p. 103 of our text and class notes.

---

3. **Synchronization**

(a) [10 points] Define the meaning of a *race condition*? Answer the question first and use an execution sequence to illustrate your answer. **You will receive no credit if only an example is provided without an elaboration.**

**Answer:** A *race condition* is a situation in which more than one processes or threads access a shared resource concurrently, and the result depends on the order of execution.

The following is a simple counter updating example discussed in class. The value of `count` may be 9, 10 or 11, depending on the order of execution of the *machine instructions* of `count++` and `count--`.

```c
int count = 10;

Thread_1(...) Thread_2(...) {
  // do something // do something
  count++; count--;
}
```

The following execution sequence shows a race condition. Two threads run concurrently (condition 1). Both threads access the shared variable `count` at the same time (condition 2). Finally, the computation result depends on the order of execution of the **SAVE** instructions (condition 3). The execution sequence below shows the result being 9; however, switching the two **SAVE** instructions yields 11. Since all conditions are met, we have a race condition.

<table>
<thead>
<tr>
<th>Thread_1</th>
<th>Thread_2</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>do somthing</td>
<td>do somthing</td>
<td><code>count = 10 initially</code></td>
</tr>
<tr>
<td>LOAD count</td>
<td></td>
<td>Thread_1 executes <code>count++</code></td>
</tr>
<tr>
<td>ADD #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD count</td>
<td></td>
<td>Thread_2 executes <code>count--</code></td>
</tr>
<tr>
<td>SUB #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAVE count</td>
<td></td>
<td>count is 11 in memory</td>
</tr>
<tr>
<td>SAVE count</td>
<td></td>
<td>Now, count is 9 in memory</td>
</tr>
</tbody>
</table>

Stating that “`count++` followed by `count--`” or “`count--` followed by `count++`” would produce different results and hence a race condition is incorrect, because the threads do not access the shared variable `count at the same time (i.e., condition 2).

See p. 193 of our text and class notes.
(b) [10 points] A good solution to the critical section problem must satisfy three conditions: mutual exclusion, progress and bounded waiting. Both progress and bounded waiting involve some form of waiting. Explain and differentiate the waiting in progress and bounded waiting. You should provide a clear answer with a convincing argument. Otherwise, you will receive no credit.

Answer: A process in the enter protocol has two forms of "waiting": (a) waiting for a decision to be made to decide who can enter, and (b) waiting to enter if it is not chosen. The progress condition states that a decision to determine who can enter must take finite time, while the bounded waiting condition states that a process waits finite time before it can enter. Therefore, the major difference is that the "waiting" in progress is waiting for a decision to be made and the "waiting" in bounded waiting is waiting to enter a critical section.

4. [25 points] Problem Solving:

(a) [25 points] Consider the following solution to the mutual exclusion problem for two processes $P_0$ and $P_1$. A process can be making a request REQUESTING, executing in the critical section IN_CS, or having nothing to do with the critical section OUT_CS. This status information, which is represented by an int, is saved in flag[i] of process $P_i$. Moreover, variable turn is initialized elsewhere to be 0 or 1. Note that flag[] and turn are global variables shared by both $P_0$ and $P_1$.

```c
int flag[2]; // global flags
int turn; // global turn variable, initialized to 0 or 1

Process i (i = 0 or 1)
```

// Enter Protocol
```c
repeat // repeat the following
    flag[i] = REQUESTING; // making a request to enter
    while (turn != i && flag[j] != OUT_CS) // as long as it is not my turn and
        ; // the other is not out, wait
    flag[i] = IN_CS; // OK, I am in (well, maybe); but,
    until flag[j] != IN_CS; // must wait until the other is not in
    turn = i; // the other is out and it is my turn!
```

// critical section
```c
// Exit Protocol
    turn = j; // yield the CS to the other
    flag[i] = OUT_CS // I am out of the CS
```

Prove that this solution does implement mutual exclusion correctly. That is, there is at most one process can be in the critical section. You will receive zero point if you provide a vague answer without a convincing argument and/or you prove this with prove-by-example. Hint: Show that the condition in the while loop cannot be true for both processes.

Answer: If process $P_0$ is in the critical section, it had executed flag[0] = IN_CS followed by seeing flag[1] != IN_CS. By the same reason, if process $P_1$ is in the critical section, it had executed flag[1] = IN_CS followed by seeing flag[0] != IN_CS. Therefore, if $P_0$ and $P_1$ are both in the critical section, we have flag[0] = IN_CS and flag[1] != IN_CS (from $P_0$’s point) and flag[1] = IN_CS and flag[0] != IN_CS (from $P_1$’s point). As a result, flag[0] and flag[1] are equal to IN_CS and not equal to IN_CS at the same time. This is impossible, and the mutual exclusion condition holds.

If the above proof is good enough, what is the purpose for the hint? Its purpose is to help you think deeper and perhaps prove mutual exclusion directly rather than going for prove-by-contradiction. Consider $P_i$’s while loop while (turn != i && flag[j] != OUT_CS). This implies that $P_i$ waits as long as it is not $P_i$’s turn and $P_j$ is not out of the critical section. This condition fails if turn == i or flag[i] == OUT_CS. Since turn i can only hold one value (i.e., either 0 or 1), either turn == 0 for $P_0$ or turn == 1 for $P_1$ holds but not both. Consequently, only one process can pass its while loop and set its flag[] to IN_CS. Note that setting flag[i] to IN_CS does not
imply that $P_i$ can enter the critical section, because there is one more test in the until part (i.e., flag[$j$] != IN_CS). Therefore, $P_i$ sets flag[$i$] to IN_CS to make a claim to enter. If $P_j$ is in the critical section or has made a claim to enter, $P_i$ waits by returning to the top of the repeat-until loop and try again!

Since a process can have three possible states: REQ, IN and OUT, all possible state combinations of $P_0$ and $P_1$ are REQ-REQ, REQ-IN, REQ-OUT, IN-IN, IN-OUT, and OUT-OUT. Note that only six are listed rather than nine because the code is symmetric, and switching the roles of $P_0$ and $P_1$ would get one of the listed six (i.e., IN-OUT is equivalent to OUT-IN).

Any state with OUT will not affect our discussion because (1) the process with OUT is not interested in entering the critical section, and because (2) the process with OUT will eventually become REQ which makes the states to be REQ-REQ, REQ-IN and IN-IN. Therefore, we actually have only three cases to consider: REQ-REQ, REQ-IN and IN-IN.

- **Case:** REQ-REQ
  If both $P_0$ and $P_1$ are in the REQ state, they must be executing the while loop. As a result, only one of them can pass through due to the value of turn. If $P_0$ has its turn, $P_0$ sets flag[0] to IN, passes through the until (because $P_1$ is in REQ), and enters the critical section.

- **Case:** REQ-IN
  Without loss of generality, let $P_0$ and $P_1$ be in REQ and IN, respectively. If $P_1$ is in IN, which means $P_1$ could be before until and after flag[1] = IN_CS, or $P_1$ actually is in its critical section. In both cases, flag[1] != OUT_CS holds in $P_0$.
  - If $P_1$ is in its critical section, $P_1$ sets turn to 1 at the end of its enter protocol. As a result, $P_0$ stays in while and cannot enter.
  - If $P_1$ is in the enter protocol right after flag[1] = IN_CS and before until. In this case, flag[1] != OUT_CS still holds. Depending on the value of turn, $P_0$ may stay in while or proceed to execute until. However, since flag[1] is IN_CS, $P_0$ loops back and cannot enter. Of course, $P_1$ will enter eventually. Once this happens, we have the REQ-IN case.

- **Case: IN-IN
  Assume that $P_0$ is between flag[0] = IN_CS and until. Then, $P_1$ may be at the same spot or $P_1$ may be in its critical section.
  - Both $P_0$ and $P_1$ are between flag[0] = IN_CS and until:
    They both see the condition in until holds and they both loop back. As a result, we have the earlier REQ-REQ case.
  - $P_0$ is between flag[0] = IN_CS and until but $P_1$ is in its critical section:
    In this case $P_0$ loops back to become REQ. Again, it is the earlier RED-IN case!

Since $P_0$ and $P_1$ cannot be in the critical section at the same time in all possible cases, mutual exclusion holds.

Note that this “direct” observation, although a bit lengthy, actually can help you understand more about this algorithm. This solution is the two-process version of a general $n$-process version due to Donald Knuth. ■