MIDTERM with SOLUTION
Oct 17, 2007

Operating systems, V22.0202, Yap, Fall'07

1. PLEASE READ INSTRUCTIONS CAREFULLY.
2. This is a closed book exam, but you may refer to one $8'' \times 11''$ sheets of prepared notes.
3. Please write ONLY on the right-hand side of a double page. USE THE left-hand side of the double page for scrap.
4. Attempt all questions.

PART 1. VERY SHORT QUESTIONS. (4 Points each)
One sentence answers only.

---

1. Where could you find a use for the `echo` command in unix?

SOLUTION: Inside a Makefile or in shell scripts such as .bashrc or .profile, so that you can see at run-time what the shell script is doing.
NOTES: What is the point of this question? It seems that echo is not doing anything useful, as it prints what we already know!
It has to do with WHEN echo is executed. For instance, when we start up a shell, it runs many files, and echo reminds us of this when it happens. This is useful for debugging such scripts.

2. How are the unix commands `ln` and `cp` similar, and how are they different? (Well... use 2 sentences for this question!)

SOLUTION: Similarity is that both create a new file entry in the current directory. Difference is that `ln` only creates a link (a pointer) to the original file, while `cp` creates a new copy of the file.
NOTES: You must not ignore the question about similarity.
In particular, it is NOT enough to simply describe what `ln` and `cp` does, and leave the reader to deduce the similarities (and differences)!

3. Your start out your cygwin/unix session in your home directory. You issue the command "`pushd /bin; pushd /home/usr`". If you now issue the command "`popd +2`", where would your current directory be?

SOLUTION: Answer is `/home/usr`.
NOTES: In other words, `popd +2` did not change the current directory. Most students wrongly said you would be back at your home directory.
WHAT IS GOING ON? Recall that each pushd and popd adds or removes a single directories from a "directory stack". The current directory is always at the top of this stack. To see like this stack, issue the `dirs` command. Your directory stack just before the `popd +2` command will have 3 entries, namely, `/home/usr, /bin, ~`. When you do `popd +n`, it will remove the `(n+1)`-st directory from the stack, counting from the left (i.e., top) of the stack. Thus the 3-rd directory, `~,` is removed. The directory stack becomes: `/home/usr, /bin`. Incidentally, `popd` is equivalent to `popd +0`.

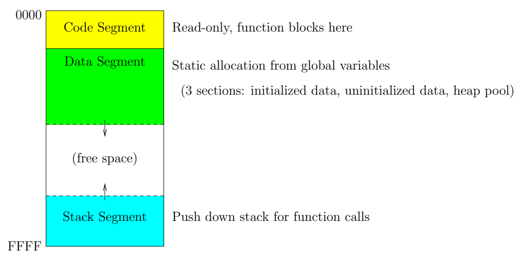
0000
Code Segment
Read-only, function blocks here
Data Segment
Static allocation from global variables
(3 sections: initialized data, uninitialized data, heap pool)
(free space)
Stack Segment
Push down stack for function calls
FFFF

Figure 1: Process image in memory

4. When you load an executable program into memory, it is laid out in a contiguous block of memory with four parts. Illustrate this layout with a diagram and label each part.

   SOLUTION: See Figure 1.

5. Name the two advantages of using shared libraries for compiling programs.

   SOLUTION: You get smaller executable program files, and at run-time, you can share library routines with other processes (thus saving main memory space).
   NOTES: Another advantage of shared libraries is that at run-time, your compiled program can access latest version of the library routines. Without shared libraries, you would have to compile your own program to get newer versions of library routines.
   GENERAL COMMENT: Many concepts in life (not just in computer science) are invented to supplant a previous concept. I will call the supplanted concept the "sup-concept". When you answer questions about a given concept, you must at least implicitly address its sup-concepts.
   The sup-concept of "shared library" is "static library". Any man-in-the-street can guess that "shared library" includes the concept of "sharing". But here, it includes the concept of dynamic loading of library routines at run-time. Your answer had better show this understanding.
   Here is an answer that the man-in-the-street could answer without special knowledge: "by using shared libraries, you can separately compile generally useful routines, and thus you do not need to compile them when you compile your own programs that use these routines. Thus you save time."
   This answer would make as much sense for the sup-concept of static libraries!

PART 2. SHORT QUESTIONS. (10 points each)
Answers must NOT be more than 3 sentences. Sentence 4 and beyond WILL BE IGNORED. So please sketch a rough solution first!

---

1. Suppose you have a file called `Makefile` in the current directory, and you type the command **make me**. Describe the actions that make program will carry out.

SOLUTION: The make program will search the Makefile for a target named ”me”, otherwise we get an error message. If found, it will see recursively make the dependencies, viewed as new ”targets”. If any dependency has a access time/date that is newer than the target's access time/date, or we recursively took some action for a dependency, then we execute the actions associated with the current target
NOTE: A target is viewed as a file of sorts. If the target is an actual file, then it has a last accessed time/date. If there is no such file (in current directory), the time/date is assumed to be ”minus infinity”. This time/date is compared to the time/date of any dependent file to decide whether the actions associated to the current target should be taken.
ADDITIONAL COMMENTS: Actually, some targets need not be explicitly specified. There could be general rules that to generate targets (e.g., how to create me.o from me.c files).

2. Consider the following goals in OS design: (1) To maximize the usage of the CPU. (2) To ensure fairness among processes. Show that these two goals are sometimes in conflict.

SOLUTION: To maximize usage of the CPU, we could simply run each process to completion (unless they are waiting for I/O). But this might not be fair to the short processes waiting on the long processes. So we schedule them for time quantums, but this wastes CPU time in doing context-switching to pre-empt processes.
NOTE: Notice that our answers show that satisfying (1) may violate (2), AND satisfying (2) may violate (1). Some of your answers only show one of these two conflicts.

3. You are designing a new editor program that will have some pretty amazing features – it will continuously search the web even as you type, looking for words, facts and other information related to the topic that you are currently writing. This information will be presented to to you in pop-up windows for your consideration. Briefly explain how you would design this feature.

SOLUTION: We use multithreading. In addition to the usual “reader thread” to respond to editing commands, and the “display thread” to update the display, we will spawning one “web thread” for each “interesting” word that you type. The web thread will call up Google search engine, and after processing the search results, it will send this information to the display thread.
NOTE: You should not “fork” and use processes for doing the various tasks since it is too inefficient to sharing of data among the processes.

4. List the typical 5 levels of the memory hierarchy. What is the main tradeoff in this hierarchy? Name two considerations that are not part of this tradeoff.

SOLUTION: 1. Registers, Cache, Main Memory, Disk, Tape.
2. The trade-off is between speed and capacity of the storage medium.
3. Other considerations are price per byte, physical size of the storage, weight of the device, power consumption, technological characteristics (e.g., dynamic RAM will not be useful for stable memory).

5. Draw the state transition diagram for process states. Give the proper labels for each state and each transition. There should be 5 states.

SOLUTION: See Figure 2.

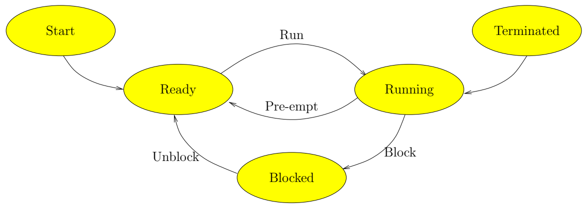

Figure 2: Process States and their transitions

PART 3. LONG QUESTIONS. (15 Points each)
You may use more than 3 sentences to explain, and ought to consider the issues in greater depth.

---

1. (10 points) Write a C program that accepts an arbitary sequence of command line arguments, and prints out the same sequence in reverse order, one argument per line. DO NOT print the name of the executable program itself.

   Correctness of your C syntax will be graded – your code should compile without error. Do not forget any necessary "includes". Three lines of C statements should suffice in the main body.

```
#include <stdio.h>
int main(int argc, char * argv[])
{
    int i;
    for (i = argc-1; i>0; i--)
        printf("
    return(0);
}
```

   Note the include line, that variable i goes from argc-1 to 1, that the printf format string has a newline character at the end, and we return an integer.

2. Joe Smart says: *Yeah, multiprogramming is cool. But it is useless to introduce it in a single-user environment like my laptop computer with a single CPU.*

   Please enlighten Joe Smart, addressing the single-user issue as well as the single-CPU issue.

   SOLUTION: Even a single user environment will use many processes. E.g., an editor can have several processes running simultaneously to do the windowing GUI, to do spell check, to do automatic backup, etc.
   While multiple processors will clearly benefit from multiprogramming, it is not true that uni-processors cannot benefit from multiprogramming. The reason has to do with complexity of writing optimal uni-processor code for multiple tasks. Since these tasks have different CPU-bursts or I/O bursts characteristics, it is impossible to solve the scheduling problem for them on a case by case basis. Instead, we leave this task to the OS scheduler to do it for us automatically, thus improving CPU efficiency.

3. Jane Smart says: *In trying to solve the MUTEX problem, it is silly to use a busy waiting solution such as Petersen's. We should always use semaphores.*

   Explain all the terms that Jane used (MUTEX problem, busy waiting, Petersen's solution, semaphors). Say why she is down on busy waiting, but counter her argument.

SOLUTION: 1. MUTEX problem is the problem of ensuring that no 2 processes enter a critical section of their code at the same time. Besides mutual exclusion, there are 3 requirements expected of an acceptable solution.
2. Busy waiting is a method where a process keeps checking for a condition to become true before continuing.
3. Petersen's solution is one of the simplest solution but it uses busy waiting.
4. Jane did not like wasting CPU cycles by busy waiting.
5. She would like to semaphore primitives such as P(S) and V(S) (equivalently, these are called acquire(S) and release(S), or signal(S) and wait(S)). The semaphore variable S is a kernel resource and typically there are process queues associated with S.
6. Although semaphors does not do busy waiting, they use kernel resources and these are not cheap to use. In particular, you will need to force context switching, and to have special queues associated with semaphore variables. These are relatively expensive compared to simple busy-waiting! Hence if the expected waiting for a critical section is brief, it may be better to use a busy waiting solution.

4. In process scheduling, we need to estimate the CPU burst time of processes. Explain a method based on using history of the process, and a formula for doing this.

SOLUTION: The method is to use previous history of actual burst times to predict future burst times. We must associate with each process an estimated burst time, and keep track of the last burst time. If the previous estimate is τ and the previous burst time was t, then we estimate the next burst time to be

$$\tau' = \alpha t + (1 - \alpha)\tau$$

where α is a constant between 0 and 1.
NOTE: some of you propose the average of previous burst times. This is not as good as the exponential decay method in this solution.