MIDTERM with SOLUTION
Oct 18, 2006

Operating systems, V22.0202, Yap, Fall'06

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.

GENERAL COMMENTS ABOUT YOUR ANSWERS:

- *Question 1 asked for "brief justifications" but many*

PART 1. SHORT QUESTIONS. EACH SUB-QUESTION IS INTENDED TO BE ANSWERED USING LESS THAN 3 SENTENCES. (5 Points part)

1. Name the three major functions of an Operating System. Briefly explain each function.

   ANSWER: The 3 major functions are: (a) Providing multi-programming: manage and isolate processes from each other, and also provide synchronization mechanisms.

   (b) Providing an extended interface to the hardware: this amounts to convenience services.

   (c) Managing resources: this requires both optimal use of resources as well and manage interactions among resources.

2. What do the abbreviations PC, SP, PSW have in common?

   ANSWER: They are names of registers which are associated with a process.

   COMMENTS: many people also said that this is the information associated with each process in the process table.

3. Explain the differences between Kernel Threads and User Threads.

   ANSWER: Kernel threads are known to the kernel and thus schedulable on the CPU, while user threads are unknown to kernel and not directly schedulable. But kernel threads are more expensive to create and may be limited in number, in contrast to user threads.

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

   ANSWER:

   The method is to use previous history of actual burst times to estimate future burst times. If the previous estimate is τ and the previous burst time was t, then we estimate the next burst time to be

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

   Here α is a constant between 0 and 1.

5. Consider the following set of processes and their data: all times are in milliseconds.

Process	Burst Time	Priority	Arrival Time
A	9	4	0
B	3	3	8
C	1	2	7
D	6	1	3

Draw the Gantt Charts for the following schedulers:

(a) First-Come-First-Serve (FCFS) method (no priority)

(b) Priority Scheduling method (with preemption) NOTE: Smaller priority number is more important.

(c) Round Robin (RR) method (with quantum time of 2, using priority to select, instead of the usual first-come-first-serve selection)

ANSWER: Gantt Charts:

(a) FCFS: [0 (A) 9 (D) 15 (C) 16 (B) 19]

(b) Priority: [0 (A) 3 (D) 9 (C) 10 (B) 13 (A) 19]

(c) RR: [0 (A) 2 (A) 4 (D) 6 (D) 8 (D) 10 (C) 11 (B) 13 (B) 14 (A) 16 (A) 18 (A) 19]

COMMENT: (c) was confusing to some students who used the FCFS method to choose next process for RR.

6. Using the same data as the previous question, what is the average turn around time under each of the methods? Show working.

   ANSWER:

   Turn around time is the time from first arrival to completion of job.

   (a) FCFS: A+B+C+D $= 9 + 11 + 9 + 12 = 41$. So average $= 41/4 = 10.25$.

   (b) Priority: A+B+C+D $= 9 + 7 + 3 + 6 = 33$. So average $= 33/4 = 8.25$.

   (c) RR: A+B+C+D $= 19 + 6 + 4 + 7 = 36$. So average $= 36/4 = 9$.

7. What is odd about the options used in this command line?

```
% gcc myprog.c  -lm -c
```

   ANSWER: "-lm" means the compilier should search the libm.a library for linking. "-c" means do not link, just produce the object file. Thus they are contradictory.

   REMARK: Logically, "-c" may come before myprog.c, but "-lm" is normally considered as coming after myprog.c. But in fact, the two gcc compilers I use (on solaris and on cygwin) do not seem to care whether you place "-lm" before or after the file name.

8. Criticize this snippet of code found in Joe Student's program:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%
int pid;
...
pid = fork();
if (pid>0)
   parent_routine();
else
   child_routine();
...
%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

   ANSWER: pid may be -1, indicating error. Some students also noted that, strictly speaking, fork() returns a value of type `pid_t`, which may not be `int`.

9. Here is Peterson's solution for 2 processes:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Initially P1wants=P2wants=false and turn=1

   P1:                                  |  P2:
    Loop forever                        |   Loop forever
                                        |
P:     P1wants <-- true                 |      P2wants <-- true
       turn <-- 2                       |      turn <-- 1
       while (P2wants and turn=2) {}    |      while (P1wants and turn=1) {}
                                        |
C:     critical-section                 |      critical-section
                                        |
V:     P1wants <-- false                |      P2wants <-- false
                                        |
D:     non-critical-section             |      non-critical-section
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

What can go wrong if we do not initialize both P1wants and P2wants to false? Why is this "wrong"?

ANSWER: If P1wants is true, but P1wants is not actually interested in entering its CS, then P2 might be stuck waiting for P1wants to become false. This is "wrong" because it violates the "Progress criteria" in the text.

REMARK: some students claim if P1wants and P2wants are initialized to true, then P1 and P2 could enter the C.S. simultaneously. This is impossible.

10. (10 points) Write a C program that reads a sequence of numbers from the command line, and spawns one child process for each number. Each child process simply prints the square of its number, and exits. SYNTAX is being graded – make sure your code can compile without error. Be sure to put any needed "includes".

    ANSWER:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
  int i, n, pid, status;

  for (i=1; i<argc; i++){
    n=atoi(argv[i]);
    pid = fork();
    if (pid<0){
      perror("error");
      return -1;
    } else if (pid==0) { // child
      printf("%d-th child answer=%d\n", i, n*n);
      return (i);
    }//else if
  }//for

  // You need not do this for your answer:
  for (i=1; i<argc; i++) {
     wait(&status); // wait for the children to die
     printf("%d-th child to exit with status=%d\n", i, status);
  }//for
```

```
  return 0;
}//main
```

Remark: a common mistake is that the first for-loop has the form "`for (i=1; i<=argc; i++)`.

PART 2. SOME SMART IDEAS. (50 Points)

1. Student Smart says "Busy waiting is dumb. There is a smarter method." *Explain what Smart has in mind. You must explain all the elements necessary to implement the smarter method. Also explain why the "smarter method" may not always be smarter.*

   ANSWER:

   Smart did not like wasting CPU cycles by busy waiting. He would like to semaphores (in the sense of Section 7.5 (p.226) of the text). The semaphore is a system-wide variable S, and a process can access it using P(S) and V(S), equivalently, using acquire(S) and release(S). There is a queue associated with S. Let us assume S is a counting semaphore that has the initial value 1.

   P(S) will first decrement S, and if (S< 0), it will add the current process to the queue (and then block).

   V(S) will first increment S, and if (S<= 0), it will wake up some blocked process on the queue.

   The process of blocking and waking up a process is expensive. Hence if the wait on a critical section is expected to be short, busy waiting might be better than the "smart" solution.

   REMARK: we asked you to describe how to implement the non-busy waiting solution.

2. Student Smart says: "There is never a reason to use pipes. We can always use files to communicate among processes. In fact, files are more general than pipes. Sometimes files can win big over pipes." *FIRST explain what Smart has in mind when he says files can replace pipes. NEXT explain what Smart means by "files are more general". GIVE a scenario which justifies Smart's remark about files winning big. FINALLY, give some reasons why pipes might be better than files.*

   ANSWER: FIRST, a pipes is really a buffer. We could therefore use a file to simulute this buffer. If you need a pipe to send information from process 1 to 2, we can use a file called "1to2". (We assume that processes know their own indices) When process 1 wants to send a message to process 2, it opens "1to2" and writes its message there. When process 2 wants to read from process 1, it just opens the file "1to2" and reads. However, there is the problem of synchronization among the processes trying to read/write to a file.

   This method is more general because there is no reason that the file "1to2" be read by only process 2. In fact, process 1 can broadcast its messages to all the other processes this way: we just use a file named "1toAll" for this purpose.

   The file method can win big when we have many processes that must broadcast messages to other processes: using pipes, we need $n(n-1)$ pipes for n processes, but using files, we just need $2n$ files. E.g., $n = 100$ means that we use 200 files as opposed to 9900 pipes.

   The difficulty with using files is, as noted, the issue of synchronization. Otherwise, there could be a race condition. For instance, when 2 processes try to write into a file, one process could clobber the output of the other process. On the other hand, this synchronization is automatically handled in pipe communication. We must therefore implement some mutex variables.

   Files are an overkill if you need just a two-way communication. Files might be less secure because it is exposed to the outside world and other processes.

3. Smart says: "Here is how to solve the MUTEX problem using using files AND pipes. Every process has an index (i=0,1,2, etc) and every process can communicate with every other process via pipes. We have two files called "Wants" and "It". Wants contains the indices of the processes that wants to enter the C.S., and "It" contains the index of the process that has permission to go to the C.S. After the "It" process has gone through its C.S., it will pick the next process to be "It", by pipe communication."

   EITHER give the details of how to achieve this Smart solution, OR point out the difficulties of achieving this solution. We do not want C code, but with enough details so that one can program it up using your description. E.g., describe any global variables you need.

   ANSWER:

   The problem is in ensuring that the updating of the Wants files is correctly done. Suppose Process A wants to enter the C.S., and it reads the Wants file, and append its index to the list, and writes the contents back to the Wants file. The problem is that when another Process B also does the same thing at the same time, it would might clobber the list containing A's index. This is a race condition (as noted in previous solution).

   If you could solve this MUTEX problem, then files would be a possible alternative to pipes (other considerations aside).

   Here is one way to solve MUTEX: Before trying to access any shared file (like "Wants"), a process must first try to open a file named MUTEX. This is the equivalent of P(MUTEX). If it succeeds, then it checks if "It" is currently occupied. If not, it puts its own index into "It", perform V(MUTEX), and proceed to the C.S. Note that V(MUTEX) amounts to deleting the file MUTEX.

   If some other process is "It", the process adds its index to the "Wants" file. The process next perform a V(MUTEX) and then blocks on some queue (alternatively, do non-blocking reads on a pipe).

   Meanwhile, the "It" process, after it has finished its C.S., and it also attempts P(MUTEX). When it succeeds, it can check if "Wants" has any process. If so, it transfers a process from "Wants" to "It" and then wakes up the "It" process. It could also wake up the "It" process by sending a message on a pipe. If "Wants" is empty, it just performs V(MUTEX).