Solutions to Final Exam

CS 160 - Operating Systems
Drake University - Spring, 2004

Directions: This is an open book examination. Do all EIGHT of the following problems. They have equal weight. Show all work. Please work first on problems with which you are more comfortable.

Problem 1 Using pseudo-code (fake code) instead of C code, if you prefer, rewrite the shell code in Figure 1-10 (page 24) so that it checks the parameters (an array of strings = array of character arrays) to see if one of the strings is `"&"`. If so then the shell should not wait for a command to complete before looping back to get another command.

my solution:

```
while(TRUE) {
  read_command(command, parameters);
  int num_parms = number_of(parameters);
  int found_amp = 0;
  for (int i=0; i<num_parms; i++)
    if (parameters[i]=="&") found_amp = 1;
  if (fork() != 0) {
    if (!found_amp) waitpid(-1, &status, 0);
  } else {
    execve(command, parameter, 0);
  }
}
```

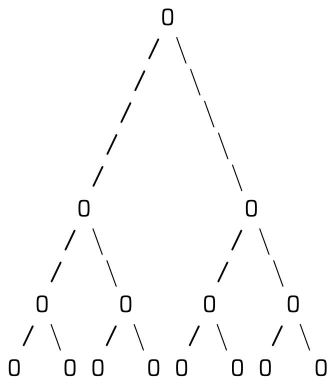
Problem 2 Clearly and completely explain what will happen when the following C code is executed. Indicate how many times each of the letters "a", "b", "c" will be displayed and why. Explain your reasoning, possibly with the help of a tree diagram. *Also,* show two of the many distinct orders in which all of the letters might be displayed, depending on preemption/scheduling issues. Defend each with an explanation.

(Technical detail: assume that if a parent process finishes before a child process, this will not affect the child process, which is sometimes true and sometimes false depending on the operating system's setup.)

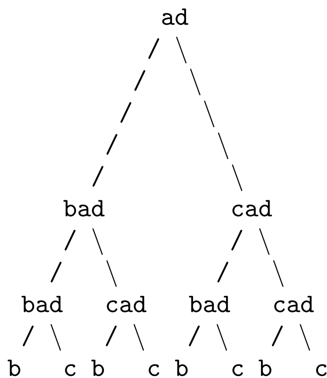
```
int i, status;

for (i = 0; i < 3; i++)
{
  if (fork() != 0)
  {
    printf("a");
    if (fork() == 0) {
      printf("c");
    } else {
      printf("d");
      exit(0);
    }
  } else {
    printf("b");
  }
}
```

solution: The original process will lead to a tree of processes as follows:



If you step through it carefully, you can see what each process will print, namely:



So seven a's, seven b's, seven c's, seven d's will be displayed. (I'm giving full credit for saying this.)

Problem 3 Rewrite the task queue portion of the `ready` function (page 609) so as to maintain the task queue as a *doubly* linked list.

my solution:

```
if (istask(rp)) {
  if (rdy_head[TASK_Q] != NIL_PROC) {
    rdy_tail[TASK_Q] -> p_nextready = rp;
    rp -> p_previousready = rdy_tail[TASK_Q]
  } else {
    proc_ptr =
    rdy_head[TASK_Q] = rp;
    rp -> p_previousready = NIL_PROC;
  }
  rdy_tail[TASK_Q] = rp;
  rp -> p_nextready = NIL_PROC;
  return;
}
```

Problem 4. *As you read the following, think about the I.H.O.P. problem on Exam Two.*

Multiple users connect via a WWW page to a web server responsible for an on-line game, where each game played requires exactly twelve players, and one player must be singled out to serve as "the coordinator", a special role. Only one game can be played at a time.

Every time a user connects to the web server, it spawns a new process to represent and accommodate this new player. (This is presumably done by means of a fork(), but you don't need to pay any attention to the creation of the new process.) These processes need to be logically "grouped together" in groups of twelve, in order to play the game. However, this just amounts to having them agree concerning a "game number" to uniquely identify an instance of the game. Use a shared global integer variable to keep track of the current game number.

Write some pseudo-code that each process can execute as soon as it is created, in order to compete for inclusion into "the next game". As mentioned, arrange for every game to be assigned a unique integer identifier ("game number"), and for some process that joins the game to be selected to be "the coordinator" of the game. When it is time to start playing the next game, its coordinator should increment the shared game number variable.

As soon as twelve processes have successfully joined the next game, and *only* then, arrange for each of these processes to call a function `play_the_game(...)` where the game number is passed to this, as well as a Boolean value used to indicate whether or not the process calling the function is or is not the coordinator for the game. Before the processes that will play the game actually call the `play_the_game(...)` function, it is OK if they are stuck in a busy-waiting loop, but no other processes can be kept in such loops. The coordinator should detect when twelve processes have joined the game, and at this point, it should take some action (perhaps by means of a shared Boolean variable) to allow the twelve processes to start playing the game.

It is required that while the next game is being set up, if too many processes try to join, all but twelve of them will be blocked (suspended) until after this game is over, *i.e.* until after the twelve processes have returned from their calls to `play_the_game(...)`. At this moment, competition for a new game should begin. A Semaphore (or semaphores) should be used to accomplish this.

Hint: Except for the game number, do NOT think in terms of creating any data, *e.g.* an object, to represent an instance of the game. Simply treat playing the game as being analogous to taking a ride together in the I.H.O.P. problem from Exam Two. Also, you don't need to do anything special to "group" processes together to play a game. In practice, it would be necessary to hook up some interprocess communication (*e.g.* "sockets"), but we'll ignore this here.

my solution:

```
semaphore spots_left = 12;
semaphore mutex = 1;
shared int game_number = 0;
shared int number_players = 0;
main() {
  int i_am_coordinator = 0;
  down spots_left;
  down mutex;
    if (number_players == 0) {
      game_number++;
      i_am_coordinator = 1;
    }
    number_players++;
  up mutex;
  while (number_players < 12); // busy waiting
  play_the_game(game_number, i_am_coordinator);
  if (i_am_coordinator) number_players = 0;
  while (number_players == 12); // busy waiting
  up spots_left;
}
```

(This is a bit cleaner than my original solution, and I deviated slightly from the specs in the problem, but I stayed true to the spirit of the thing. Any correct solution is acceptable.)

Problem 5. The following table reflects the current usage of four types of resources among four processes. The notation m/n in the table means that a particular process might at some moment require up to n instances of a particular type of resource (*i.e.* have a need for this many, at a maximum), and that it is currently holding m of these.

	$R1$	$R2$	$R3$	$R4$
$P1$	1/5	1/3	2/3	1/8
$P2$	2/5	3/4	0/4	1/2
$P3$	1/3	2/7	2/3	0/6
$P4$	2/3	3/5	1/3	4/6

Assume also that at the present time there are two instances of each type of resource still available. If $P1$ requests another instance of $R1$, will granting this request be safe, or could it result in deadlock. Explain your answer in detail.

solution: Safe. Here is why. If you grant the request, the number of resources still available will be 1, 2, 2, 2 (respectively). Now, the OS can suspend everybody except P4 (if necessary). Notice that P4 will not make too many requests, and can be allowed to run to completion. When it releases its resources, the number of available resources will be 3, 5, 3, 6. This will allow P3 to continue since it won't make too many requests. When it is done, the number of available resources will be 4, 7, 5, 6. Now P2 can run, and when it finishes the number of available resources will be 6, 10, 5, 7. This will allow P1 to continue.

Problem 6. A 128-byte computer uses 128 physical addresses to reference each of its individual bytes. The current contents of this memory are described in the following table, where the notation $nn : mm$ means that address nn contains the byte mm.

00 : 07	10 : 04	20 : 09	30 : 07	40 : 0C	50 : 0A	60 : 04	70 : 02
01 : 04	11 : 07	21 : 0B	31 : 0D	41 : 0C	51 : 03	61 : 02	71 : 07
02 : 02	12 : 09	22 : 04	32 : 00	42 : 00	52 : 0A	62 : 02	72 : 05
03 : 07	13 : 0B	23 : 07	33 : 03	43 : 02	53 : 02	63 : 01	73 : 05
04 : 00	14 : 04	24 : 02	34 : 06	44 : 0D	54 : 02	64 : 09	74 : 0D
05 : 09	15 : 03	25 : 08	35 : 07	45 : 03	55 : 00	65 : 08	75 : 03
06 : 0B	16 : 02	26 : 0F	36 : 04	46 : 09	56 : 01	66 : 08	76 : 04
07 : 01	17 : 0C	27 : 01	37 : 01	47 : 00	57 : 07	67 : 0B	77 : 00
08 : 03	18 : 01	28 : 00	38 : 01	48 : 00	58 : 06	68 : 04	78 : 00
09 : 05	19 : 00	29 : 05	39 : 07	49 : 05	59 : 04	69 : 04	79 : 0A
0A : 00	1A : 07	2A : 05	3A : 03	4A : 07	5A : 04	6A : 05	7A : 06
0B : 0F	1B : 0E	2B : 03	3B : 02	4B : 02	5B : 03	6B : 04	7B : 02
0C : 06	1C : 09	2C : 00	3C : 0C	4C : 08	5C : 0E	6C : 06	7C : 00
0D : 04	1D : 04	2D : 09	3D : 06	4D : 08	5D : 02	6D : 09	7D : 09
0E : 0A	1E : 00	2E : 0F	3E : 00	4E : 0B	5E : 07	6E : 0A	7E : 04
0F : 04	1F : 02	2F : 00	3F : 03	4F : 01	5F : 0B	6F : 0A	7F : 00

Now, assume that a two-tiered (two-level) virtual memory paging system is used to convert twelve-bit virtual addresses into seven-bit physical addresses, based on sixteen byte pages. The first four (high-order) bits of a virtual address are the primary table field, the next four are the secondary table field, and the last four (low-order) bits are an offset. Assume that the primary table for this system begins at memory location 00. Assume that each table entry is a single byte, and that the three lowest order bits of each table entry give a frame number, and the next higher bit is set to one if the page sought is currently loaded in physical memory.

(a) Do the tables currently contain enough information to convert the virtual address $E6F$ (hexadecimal) to a physical address, or will a page fault occur. Explain.

solution: The E in E6F leads us to location 0E (in primary page table). Here find 0A, which in binary is 00001010. Entry is valid, and leads us to frame 2 (010), which is part of the secondary page table. Specifically, go to location 26 (because of 6 in E6F). Here find 0F, which is 00001111. This is valid and takes us to frame 7 (111). Specifically, go to location 7F.

(b) Suppose that a transfer look-aside (TLB) buffer is used with this system, and suppose that each lookup in the TLB takes 10 nanoseconds. Also assume that *EACH* main memory access takes 100 nanoseconds. Assume too that on a TLB miss, the only significant time required to update the TLB is the time required to access memory, but that after doing so, the TLB must be read again in order to make the virtual-to-physical address conversion. What does the hit ratio for the TLB need to be in order to ensure an average time of 20 nanoseconds for converting a virtual address to a physical address?

solution: $10h + (10 + 100 + 100 + 10)(1 - h) = 10h + 220(1 - h) = 220 - 210h$. Set this to 20 and solve to get $h = 200/210 \approx 0.95 = 95\%$. (Notice why a miss costs 220 nsec.)

Problem 7. Assume that the operating system maintains a linked list of available blocks of free memory, and that currently this begins with a 12K block followed by a 8K block followed by a 2K block followed by an 11K block. Assume that requests are made for a 8K block, followed by a 4K block, followed by an 6K block. Show exactly how the link list will change as these requests are fulfilled assuming the following allocation strategies:

(a) Best-Fit
solution:

```
12K, 8K, 2K, 11K
12K, 2K, 11K
12K, 2K, 7K
12K, 2K, 1K
```

(b) Worst-Fit
solution:

```
12K, 8K, 2K, 11K
4K, 8K, 2K, 11K
4K, 8K, 2K, 7K
4K, 2K, 2K, 7K
```

(c) Next-Fit
solution:

```
12K, 8K, 2K, 11K
4K, 8K, 2K, 11K
4K, 4K, 2K, 11K
4K, 4K, 2K, 5K
```

Problem 8. Assume that each disk block contains 512 bytes. Assume that a 4-byte integer is used to identify each block (the "block number"). Assume that each i-node in a Unix system occupies one block, and that half of it is used to maintain file attribute information, followed by forty data block numbers, followed by sixteen block numbers used as single indirect pointers, followed by four block numbers used as double indirect pointers, followed by four block numbers used as triple indirect pointers.

(a) How big can a file be (in bytes) using this system? (I am just referring to the data here, not the overhead blocks. Be exact.)

solution: A block consisting entirely of pointers can only contain 128 such since $512/4 = 256$. The number of data blocks in the file is therefore $40+16\cdot 128+4\cdot 128^2+4\cdot 128^3 = 40+2048+65536+8388608 = 8456232$. Multiply this by 512 to get 4,329,590,784 bytes (about 4G bytes).

(b) Show the details of doing a random access into the file so as to locate the byte in logical address (byte position) 10,000,000 (decimal). That is, show how to chase the pointers to find this byte in the file.

solution: $10,000,000 = 19531 \cdot 512 + 128$, so need to find block number 19531 in the file. Skip over first 40 blocks: 19531 - 40 = 19491. Skip over next 2048 blocks: 19491 - 2048 = 17443. Now, $128^2 = 16384$, so skip over first doubly indirect pointer: 17443 - 16384 = 1059. Use second doubly indirect pointer ("doubly indirect pointer number one"). Follow this to a block of 128 singly indirect pointers. Now, $1059 = 8 \cdot 128 + 35$. So skip over the first eight pointers and use "pointer number eight" in this block. It points to a block containing 128 direct pointers. Skip over the first 35 of these and use "pointer number 35" in this block. This leads straight to the block of data in the file sought. The desired byte is "byte number 128" in this block because of $10,000,000 = 19531 \cdot 512 + 128$.