CS 159: Homework Project 2

1) A computer has 6 tape drives and N processes competing for them. Each process may need up to two drives. For which values of N is the system deadlock free? Explain. [3 pts]

2a) In regards to the attached figure 3.8, if process C requested S instead of R at step (o), would deadlock occur? Explain why or why not. [2 pts]

b) Now suppose C requested both R and S at step (o). Would this lead to a deadlock? Explain. [2 pts]

3a) In regards to the attached figure 3.12, what are the horizontal and vertical bounds (in terms of process A and B instructions) of the "all-encompassing danger region" (i.e., the region, which if entered, would eventually lead to deadlock) that should be avoided by any trajectories? [1 pt]

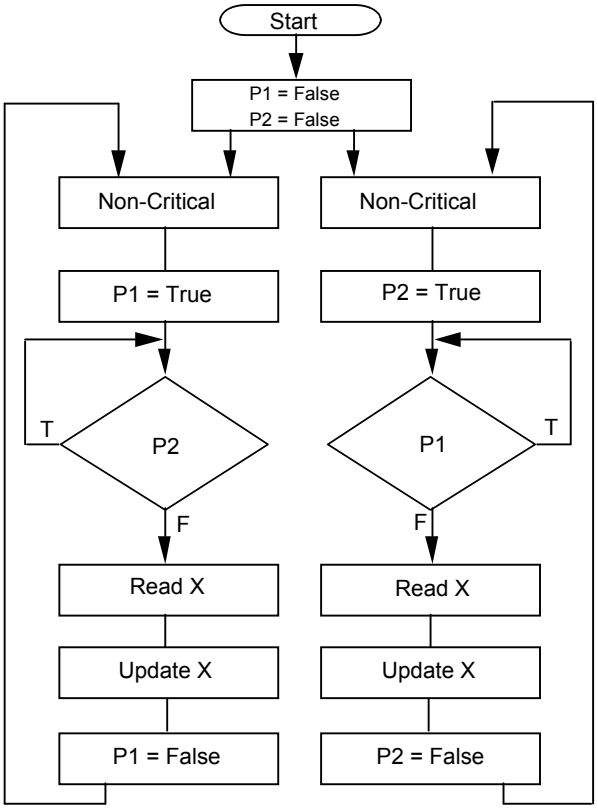
b) Is there any way for the execution trajectory to ever reach the intersection of I3 and I7? Explain why or why not. [1 pt]

c) Suppose process A requests the printer at I2 instead of I1 and then releases it at I3. If the instruction execution trajectory is currently at point t as shown in the figure, give two completion paths which will allow point u to eventually be safely reached. [1 pt]

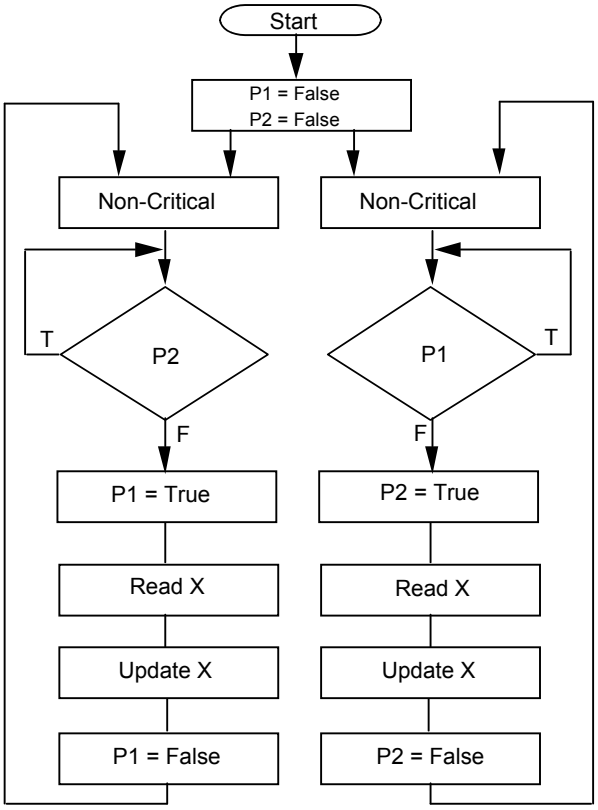
d) Suppose you saw an execution trajectory consisting of diagonal lines. On what type of hardware platform might such a trajectory be possible? [1 pt]

4) There are two flow charts below. Each is intended to provide mutual exclusion in the use of resource X by two processes executing that flowchart (process 1 executes the left half of the flowchart; process 2 executes the right half). The Boolean variables P1 and P2 are intended to be flags which provide mutually exclusive access to resource X. Both processes initially enter and start at the top block. Assume that the operations between any block (or diamond) in the flow chart are interruptible, and that the OS scheduler can swap between the two processes in any arbitrary order. Analyze both flowchart (a) and (b). Does either flow chart provide effective mutual exclusion without any other problems? If so, explain in words how it is accomplished. If not, explain why not and give an execution sequence for the two processes that would violate the mutual exclusion condition.

(a) [8 pts]



(b) [8 pts]



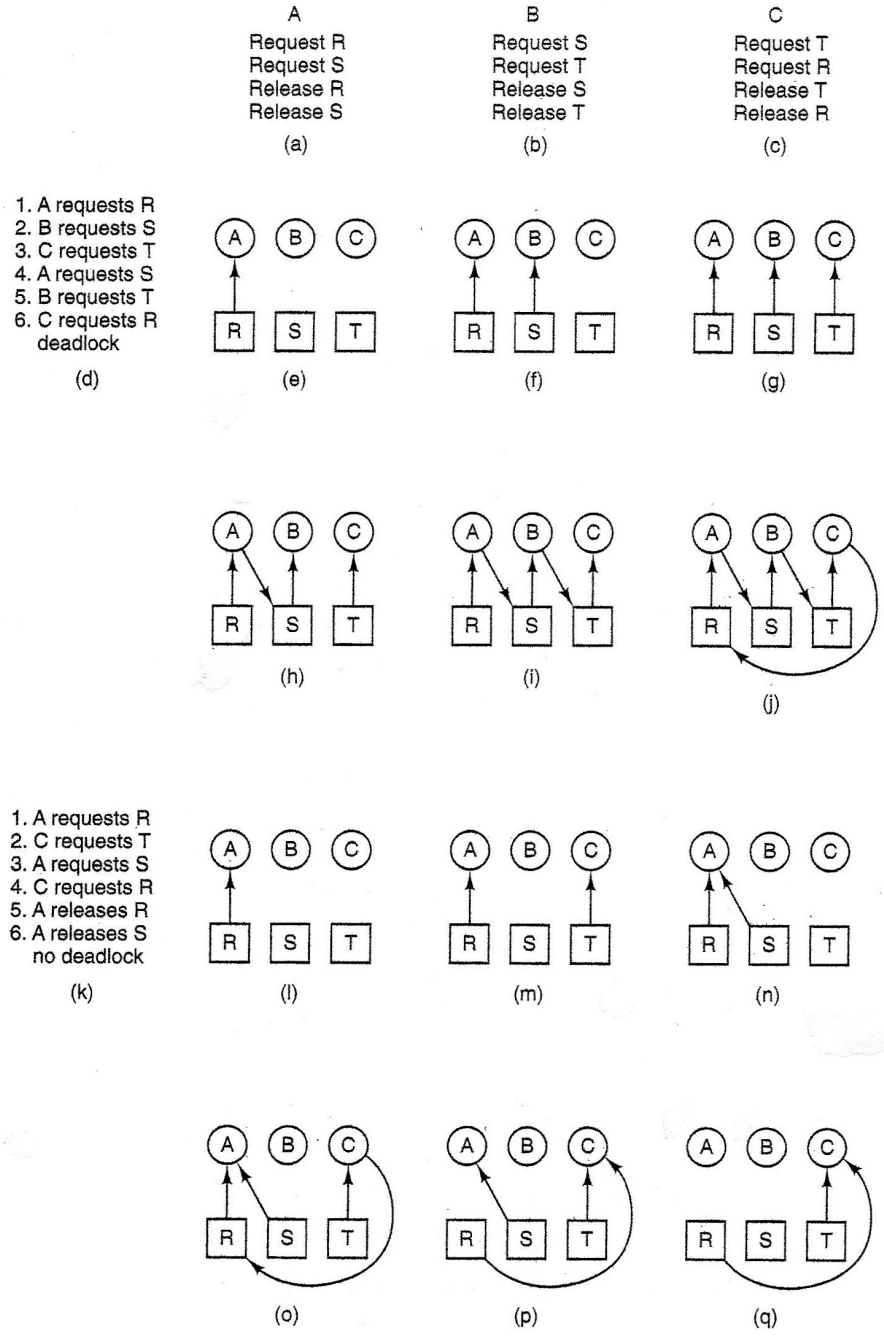


Figure 3-8. An example of how deadlock occurs and how it can be avoided.

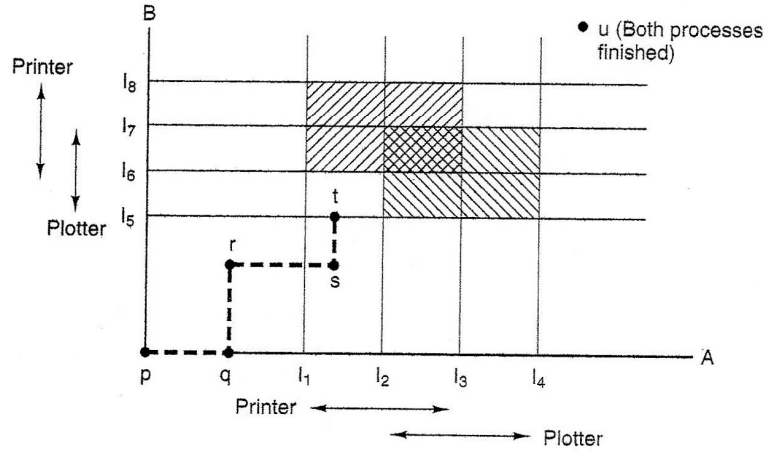


Figure 3-12. Two process resource trajectories.

5) In class, an example of two people trying to cross a river by stepping on stones which are mutually exclusive was presented. Deadlock occurs if two people cross from opposite directions and meet somewhere in the middle. Consider a situation where each person agrees to obey the following protocol:

> Upon arrival at the river bank, check to see if there is someone on the opposite side who wants to cross or is currently crossing. If so, wait until he is done before proceeding.

Will this protocol be effective in circumventing deadlock while effectively utilizing resources ? If so, explain why. If not, explain and give a scenario in which the protocol is ineffective. [3 pts]

6) The four conditions necessary for deadlock to occur are listed below. Explain what each one means. [4 pts]

a) Mutual Exclusion

b) Hold and Wait

c) No Preemption

d) Circular Wait

7) Draw the resource allocation graph for the following sequence of instructions. Assume no jobs are initially in the system. Is the system in a deadlocked situation after the instruction sequence ? Explain why or why not. [3 pts]

Process A requests Resource R
Process B requests Resource S
Process B requests Resource R
Process A requests Resource S

As illustrated in the last project, the operating system contains interface calls that enable user programs to communicate with it – to fork processes and to enable processes to get their own and their parent's process IDs. There are many more OS system interface calls.

In a multiprogrammed system where several processes can be active concurrently, kernel functions are needed to enable user processes to: 1) create (fork) other processes, 2) coordinate (via wait and sleep) their execution sequences, and 3) communicate with (signal) each other. Processes may also need to be aware of time, and react to time-based events.

The role of interprocess communication (IPC) is twofold. First, it transmits information which may be necessary between processes. Second, it provides a capability for process synchronization. Although processes can always pass information through named files, facilities are provided to enable the processes to communicate more directly. In this project, we examine additional functions that can be used to accomplish IPC.

For the following programs: Hand in your source code listing and a printout of the execution trace showing that the program performs correctly for various test conditions. Mark and label your printout using a pen/pencil/highlighter to identify the output of your program for given inputs. Note that part of this assignment involves seeking out information on your own from various sources. This is a significant change from Project 1 where most of the information needed regarding the system calls, including sample code demonstrating their usage and functionality, was provided to you in the handout. Although brief information is provided below regarding the various system calls used for this project, you will probably need to consult other reference documentation such as that contained on-line or in other operating systems and systems programming books.

8) Signals provide a method for transmitting software interrupts to processes. Once a receiving process has detected a signal, it can then act on the basis of that signal. Signals are used by the kernel to deal with certain kinds of error conditions (e.g., floating point overflow). Signals also enable the user to interactively abort processes running in the foreground of their command shell by typing control-c. If a signal is sent to a process that has not explicitly declared a method to deal with that type of signal, the process is simply aborted. Processes can specify protocols for handling a signal by providing a function to be called upon receipt of that signal. After a signal has been caught, it is necessary to re-enable the signal catching; otherwise, if another signal of the same type arrives before the signal catching is re-enabled, the process is aborted. A process can also choose to ignore a certain type of signal.

a) Write a program that simply has five successive sleep(1) statements in it. Before each call to sleep, print out a message such as "Sleep #1", "Sleep #2", etc. After the final sleep, print out "Program exiting". Run the program interactively and press control-c at various points in its execution. [5 pts]

b) Use **signal(SIGINT, SIG_IGN)** to augment the program above such that it ignores control-c. Include the header file <signal.h>. [5 pts]

c) Now write an interrupt handler function that will catch the signal. Put a print statement inside the interrupt handler (e.g., "Jumped to interrupt handler"). Run the program and press control-c at various points. Verify that the interrupt handler functions correctly. Now, try typing two control-c's during one execution and explain what happens. [5 pts]

d) Augment program (c) above to handle multiple control-c interrupts within the same execution. Therefore, the very first thing the signal handler should do, after having caught a signal, is re-enable signal catching. Run the program to verify that the interrupt handler re-enables itself and can indeed catch multiple control-c's entered by the user. [5 pts]

9) Signals can also be pre-programmed to occur after a specified amount of time. The function **alarm(t)** arranges for a SIGALRM type signal to be sent to the process after t seconds. Since alarm calls are not stacked, if alarm is called with t = 0, any previously set alarm is disabled. Write a program that will prompt the user for some input, allow the user a certain amount of time to respond (e.g., 10 seconds), and if no response is provided within the allotted time, an appropriate timeout message printed on the screen. Note that the alarm has to be disabled if the user responds properly within the allotted time. [10 pts]

10) Unlike a signal, which conveys only the occurrence of a particular event and contains no information content, a pipe can be thought of as a scratch file created by a system call. It can be used as a communications channel between concurrently running processes. The interface call to a pipe is similar to that for any file. In fact, the process reads and writes to a pipe just like any file. Unlike files, however, pipes do not represent actual devices or areas on disk. Instead, they are transient areas in memory. Just as a pipe is used by the shell to pass information on the command line, a program pipe is used to pass information from one process to another. A pipe's main advantage is that it can provide much higher bandwidths than if the processes read and write from a physically moving media (as would be the case with ordinary disk files).

Pipes have a finite capacity and relay their information on a first-in, first-out (FIFO) basis. The system blocks a process until the read/write can be satisfied. That is, if the reader gets ahead of the writer, the reader just waits for more data. If the writer gets too far ahead of the reader and manages to completely fill the pipe (the size of which is system dependent), it sleeps until the reader has a chance to catch up. Also, once information is read from the pipe, it is erased, thereby freeing up pipe space.

It should be noted that pipes do have some disadvantages. One of the more notable ones is the fact that the processes communicating over the pipe must be related, typically as a parent and child, or as two siblings. The reason for this is because one process has to tell the other what the file (pipe) descriptor is; however, the descriptor

is only valid in the context of a particular execution environment. Therefore, two unrelated processes generally cannot share a pipe. This can be a constraint for many applications where the potential reader and writer of the pipe originate from different, unrelated processes. On the other hand, if a pipe is established in a process before creating (forking) another process, the new process will inherit the pipe file descriptor thereby making it valid in both execution environments.

A sample initialization sequence for a pipe that will enable messages of 15 characters to be exchanged between processes is of the form:

```
#include <stdio.h>
#define MSGSIZE 16 /* This figure should include room for a terminating null */

int  pfd[2], retval;
char msg [MSGSIZE], buffer[MSGSIZE];

retval = pipe(pfd);
write (pfd[1], msg, MSGSIZE);
read (pfd[0], buffer, MSGSIZE);
```

The call creates a pipe represented by two file descriptors that are returned in the **pfd** array. Writing to pfd[1] puts data in the pipe; reading from pfd[0] gets data out.

a) Write a single program which will create a pipe, write at least four messages down it, and then read them back. Although this particular exercise does not demonstrate IPC (since there is only one process), it demonstrates how pipes can be used as a scratchpad FIFO buffer within a process. [9 pts]

b) Write a program which will spawn a child process and establish a one-way communication pipe from the parent to the child. The parent process should write at least four messages down the pipe. The child process should read all the messages from the pipe. Insert process ID statements so that the progress of the two processes can be traced. [9 pts] For example:

"This is the parent process. Writing first message into pipe."
"This is the child process. Reading first message from pipe. Contents is:"

c) Using the timer capability of one of the previous problems, design a program which will determine the size limit of a pipe on the system you are using. Only a certain number of bytes can be put in the pipe without it being read. Knowing this limit is important. Normally, if there is room in the pipe, the write function will return immediately. However, if a write is attempted that would overfill the pipe, process execution is suspended until room is made by another process which can read from the pipe.

Create a program which will input data into a pipe one character at a time. Count the number of characters as they are written into the pipe. Print out a message after every 1K (1024) characters are written to the pipe. If the process does not read from the pipe, eventually the pipe will fill and the next attempted write to the pipe will block the process. Use the **alarm** function to prevent the process from hanging indefinitely after the pipe gets full, but be sure to set the time limit large enough so that you can be sure the pipe is completely filled. In the interrupt handler function, print out a message showing the final count of characters that can be input to the pipe (i.e., the maximum size of pipes on your system). [15 pts]

So, the printout will appear something like:

1024 characters in pipe
2048 characters in pipe
xxxx characters in pipe
Write blocked after yyyy characters.