Project 2: Multi-programming

Introduction to Operating Systems CSE421
Assigned: 10/13/2003 Due: 11/13/2003,11.59PM
Be warned: No extension of the due date will be given for this project.

In the second project you will design and implement appropriate support for multiprogramming. You will extend the system calls to handle process management and inter-process communication primitives. You will add this to the coded first project. Make sure you correct all the deficiencies in your first project before starting the second project. This solution for project1 will be covered as part of next week's recitation.

Nachos is currently a uni-programming environment. We will have to alter Nachos so that each process is maintained in its own system thread. We will have to take care of memory allocation and de-allocation. We will also consider all the data and synchronization dependencies between threads. You will first design the solution before coding. Here are the details:

1. Alter your general exceptions (non-system call exceptions) to finish the thread instead of halting the system. This will be important, as a run time exception should not cause the operating system to shut down. You will most likely have to revisit this code several times before your project is complete. There are several synchronization issues you will have to handle during thread exit.

2. Implement multiprogramming. The code we have given you is restricted to running one user program at a time. You will need to make some changes to addrspace.h, and addrspace.cc in order to convert the system from uniprogramming to multiprogramming. You will need to:
   a. Come up with a way of allocating physical memory frames so that multiple programs can be loaded into memory at once.
   b. Provide a way of copying data to/from the kernel from/to the user's virtual address space.
   c. Properly handling freeing address space when a user program finishes.
   d. It is very important to alter the user program loader algorithm such that it handles frames of information. Currently, memory space allocation assumes that a process is loaded into a contiguous section of memory. Once multiprogramming is active, memory will no longer appear contiguous in nature. If you do not correct the routine, it is most likely that loading another user program will corrupt the operating system.

3. Implement the **SpaceID Exec(char *name)** system call. Exec starts a new user program specified in the parameter "name", running within a new system thread. You will need to examine the "StartProcess" function in progtest.cc in order to figure out how to set up user space inside a system thread. Exec should return –1 on failure, else it should return the "Process SpaceID" of the user level program it just created. (Note: SpaceIDs can be kept track of in a similar manner to OpenFileIDs of your project 1, except that you will want to keep track of them outside the thread.)

4. Implement the **int Join(SpaceID id)** and **void Exit(int exitCode)** system calls. Join will wait and block on a "Process SpaceID" as noted in its parameter. Exit returns an exit code to whoever is doing a join. The exit code is 0 if a program successfully completes and it is another value if there is an error. The exit code parameter is set via the **exitcode** parameter. Join returns the exit code for the process it is blocking on, -1 if the join fails. A user program can only join to processes that are directly created by **the Exec system call**. You can not join to other processes or to yourself. You will have to use semaphores inside your system calls to coordinate Joining and Exiting user processes. You will observe that this can be modeled as sleeping barber IPC (inter process communication).

5. Implement the **int CreateSemaphore(char *name, int semval)** system call. From the remove system call that you implemented in Project1 you would have realized that we will have to update start.s and syscall.h to add the new system call signatures. You will create a container at the system level that can hold up to 10 named semaphores. The CreateSemaphore system call will return 0 on success and –1 on failure. The CreateSemaphore system call will fail if there are not enough free spots in the container, the name is null, or the initial semaphore value is less than 0.

6. Implement **int wait(char *name)** and **int signal(char *name)** system calls. The name parameter is the name of the semaphore. Both system calls return 0 on success and –1 on failure. Failure can occur if the user gives an illegal semaphore name (one that has not been created).

7. Implement a simple shell program to test your new system calls implemented as above. The shell should take a command at a time, and run the appropriate user program. The shell should "Join" on each program "Exec"ed, waiting for the program to exit. On return from the Join, print the exit code if it is anything other than 0 (normal execution). Also, design the shell such that it can run program in the background. Any command starting with character (&) should run in the background. (Ex: &create will run the test program create program in the background.)

8. Create a user level solution to the producer/consumer problem. The solution should consist of three programs: a main level start program, producers and consumers. The main level start program should minimally prompt the user for the number of producers to "exec" and the number of consumers to "exec" as well as an iteration count for the producers and consumers and the size of the bounded buffer. The main start program will then Exec the producers and the consumers and wait until all the processes are complete. Each producer should produce a unique character payload (ex: Producer 1 produces "a" and producer 2 produces "b" etc.). Producers and consumers should print their iterations in a clear and concise format. Note that exec does not allow command line arguments. You may use "file" from project1 as the shared buffer to take care of this problem.

9. Design two user programs other than the items 7, and 8 above that can be run inside the shell to demonstrate the robustness of your overall project solution.

10. Documentation (10%) Includes internal documentation, and external documentation as described in Project1. Create a READE file and place it the code directory. Tar your code and submit it as one file. Follow the directions given in Project1. We plan to run automatic plagiarism detection software to detect any copied projects. The consequences are quite unpleasant for academic dishonesty. So work on your own and not in groups. **This is not a group project. No late projects will be accepted. Submit the external documentation containing the design represented by a class diagram online on the due date. Please take a hardcopy with you to the TA when you do the demo of the project.**