Project 2: Multi-programming and Inter-process Communication

Introduction to Operating Systems **CSE4/521**

Due: 11/10/2005,11.59PM

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 an 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 among threads. You will first design the solution before coding. Here are the details:

1. Add a case in exception handler so that non-system call exceptions can finish (currentTthread->Finish()) the thread. 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 (thread Finish()).

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 memory in terms of pages. 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 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 whomever is doing a join. The exit code is 0 if a program successfully completes, 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.

5. Implement the **int CreateSemaphore(char *name, int semval)** system call. You 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 upto 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. **Make sure you follow the wait and signal as the mnemonics for these two and not down and up or P and V.** 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. Implement a simple **barrier** primitive using the primitive thread routines currently available (eg. Thread::Sleep()). There is barrier.h file in the threads directory which will define the method signatures for your implementation. The class consists of four items. The Barrier constructor will allow you create a barrier of a particular capacity (size). The Barrier::barrierSynch method will cause each thread invoking the barrier to block until the thread capacity is met. For example, if a barrier is created with a size of 3 elements, the first two threads that call the barrierSynch() method will block, and the third will release the first two and continue. The order for release should be the same as the order for arrival. A Barrier::print() method prints list of threads blocked within the barrier as well as the capacity information. You will also define a destructor, which will release any blocked thread back into the system. You will define the necessary private data and implement all methods contained within the barrier.cc file. Provide the appropriate tests in order to demonstrate the success of your simple barrier.

9. Implement system calls **int CreateBarrier(char *barrierName, int capacity); int BarrierSynch(char *barrierName);** Both system calls return 0 on success and –1 on failure. Failure can occur if the user gives an illegal barrier name (one that has not been created).

10. Implement a test program that uses the user-level barriers: Use barriers to coordinates three types of processes in the problem described Sant-Elf-Reindeer IPC problem. Santa Claus sleeps in his shop at the North pole and can only be awakened by either (1) all 2 reindeer being back from their vacation, or (2) by some of the elves having difficulty in making toys. Elves can wake Santa up when three of them have problems. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for the three elves to return. ``Three elves waiting'' has higher priority than all reindeer getting back. That is because without any toys Santa cannot go on his gift_giving spree. All the elves' difficulty should be solved before Santa leaves.

11. Documentation (10%) Includes internal documentation, and external documentation as described in Project1.