Project #2 – Thread Synchronization, Scheduling and Inter-Process Communication (IPC)
Project created by C.Egert and modified by B. Ramamurthy

Introduction to Operating Systems **CSE421**
Assigned: October 7, 2002 **Due: November 6, 2002 11:59:59 PM**

The first project was designed to further your understanding of the relationship between the operating system and user programs. In this project we will explore the process synchronization (thread synchronization in the case of Nachos) and inter-process communication (IPC) models.

Phase 1: [10%] Understand the Code and test a simple IPC solution

The first step is to read and understand the part of the thread system that is available in the current distribution. Study the thread.h, thread.cc and threadtest.cc programs. Study the kernel level semaphores provided for synchronization. Implement a simple Producer/Consumer solution with limited buffer size. Allow the user to input the number of producers and consumers and the number of iterations of the producer and consumer cycle.

Phase 2: [15%] Implement Priority Scheduling and Simple Aging

Modify the thread scheduler to always return highest priority thread. You will have to create another parameter – the priority level of the thread represented by an integer value; leave the old constructor as such to allow for backward compatibility. For your implementation, set a constant to indicate the highest-level thread allowed. Allow priorities from 0 (lowest) to 6 (highest). Provide the appropriate tests in order to demonstrate the success of your priority scheduling system.

Most priority scheduling solutions will starve out a low priority thread. After you complete and test the above part, implement a simple aging system to take care of the starvation problem. You will need to create a new constructor for the Thread to take both a priority level and whether aging will occur or not. 0 for no aging, 1 for aging with priority increases of one unit for every *x* times the thread is ready to run. That is for every x thread switches from ready to run, increment the priority of the low priority threads by 1. Specify x as a constant in your program and make appropriate size to see an effect. Add thread ("t") debug statements to display the trace of the aging algorithm. Provide the appropriate tests in order to demonstrate the success of your aging system.

Phase 3: [20 + 20 =40%] Synchronization through Barriers

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.

Chris Egert has designed a new primitive called a N-way barrier. The N-Way barrier is different from a conventional barrier in that the N-way barrier supports multiple queues. Each one of these queues will hold threads of a particular 'type'. Types are numbered from 0 to N-1. The N-way barrier releases when the barrier reaches capacity AND when there are enough threads of each type within the barrier. This lab will allow students to implement a simpler version of the N-way barrier, in that all queues from type 1 to type N-1 only require a single thread, and the type 0 queue will contain the remainder. For example, consider a 3-way barrier of capacity/size 7. In this case, the barrier releases when we have 1 entry of type 2, 1 entry of type 1, and the remaining 5 of type 0. If, however, one type of components is missing, it is conceivable that the additional threads could block within the N-way barrier. For example, 0 entries of type 2, 1 entry of type 1 and 10 entries of type 0 is a valid state, as the N-way barrier is still waiting for a type 2 entry. Upon release, only the capacity number of thread should be released, and should correspond to the arrival order within each of the queues. Any additional threads must wait for the barrier to re-release. You are to use the additional method signatures within barrier.h for your implementation. Note the barrier constructor that has three parameters: char* name, int ways, int size, where ways indicates the value of N and size is the total capacity of the barrier before it releases. The method Barrier::barrierSynch(int type) will cause each thread utilizing the barrier to synchronize. The formal parameter indicates the thread type. The method barrierSynch(int type) will return an integer as to the success or failure of the method call. A value of 0 means that the synch succeeded and that the code resumed after the barrier. A value of –1 means that the call failed due to an improperly constructed N-way barrier. 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.

Phase 4: [25%] Putting it all together

Jurassic park IPC problem:

See the enclosed diagram of a model Jurassic Park.

- You can visit the dinosaur exhibit only by a tour bus. Tour bus has a driver and tour guide. There are two types of visitors to the park: season ticket holders and one-time visitors. They arrive with equal probability but season tickets holders normally have higher priority. But in order to avoid denial of service to one-time visitor their priority is raised periodically according to a certain policy. The policy is that allow a one-time visitor (if one is waiting) for every two season ticket holders. That is, in case there are too many season ticket visitors, one-time visitors may not get a chance because of priority. To avoid this 1 one-time visitor in the wait queue are allowed for every 2 season ticket visitors. (Hint: use scheduling with priority and aging for this.)
    - Implementation details: Use threads for the bus, guide, driver, and visitors. Use priority scheduling with aging between two types of visitors. Use priority 3 to season ticket holder, 1 to one-time visitor, 5 to the bus and 4 to driver and conductor.
- There is a holding room, which holds 20 visitors. Visitors enter this room to wait for the bus. When the bus arrives a bell rings and the conductor and the driver lounging around arrive at the bus. Then the visitors from the holding room board the bus. When the bus, guide, driver and visitors are ready, the bus door closes and the bus is ready to begin the tour. When the holding room gets full people wander around the other parts of the park. (Hint: Use semaphore to block the entry into the holding room, which has limited capacity.)
    - Use N-way barrier to synchronize the various participants. Avoid barrier overflow using proper synchronization primitives.

- Bus has a capacity of 22 (20 visitors, 1 driver and 1 guide.) For simplicity assume that a bus never leaves without full load. If visitors don't fill the bus, park officials fill it up before it leaves. Bus could also be modeled as one of the synchronizing threads.

Timing considerations and test data: You must choose proper random timing for the various threads to illustrate the various conditions.

Implement a solution for this problem and run a suitable test suite to show it is working properly.

Phase 5: [10%] Documentation

This includes internal documentation (comments) and a ***BRIEF, BUT COMPLETE*** external document (read as: paper) describing what you did to the code and why you made your choices. **DO NOT PARAPHRASE THIS LAB DESCRIPTION AND DO NOT TURN IN A PRINTOUT OF YOUR CODE AS THE EXTERNAL DOCUMENTATION.**

Deliverables and grading

When you complete your project, remove all executables and object files. If you want me to read a message with your code, create a README.NOW file and place it in the nachos *code* directory. Tar and compress the code, and submit the file using the online submission system. It is important that you follow the design guidelines presented for the system calls. I will be running my own shells and test programs against your code to determine how accurately you designed your lab, and how robust your answers are. Grading for the implementation portion will depend on how robust and how accurate your solution is. Remember, the user should not be able to do anything to corrupt the system, and system calls should trap as many error conditions as possible.

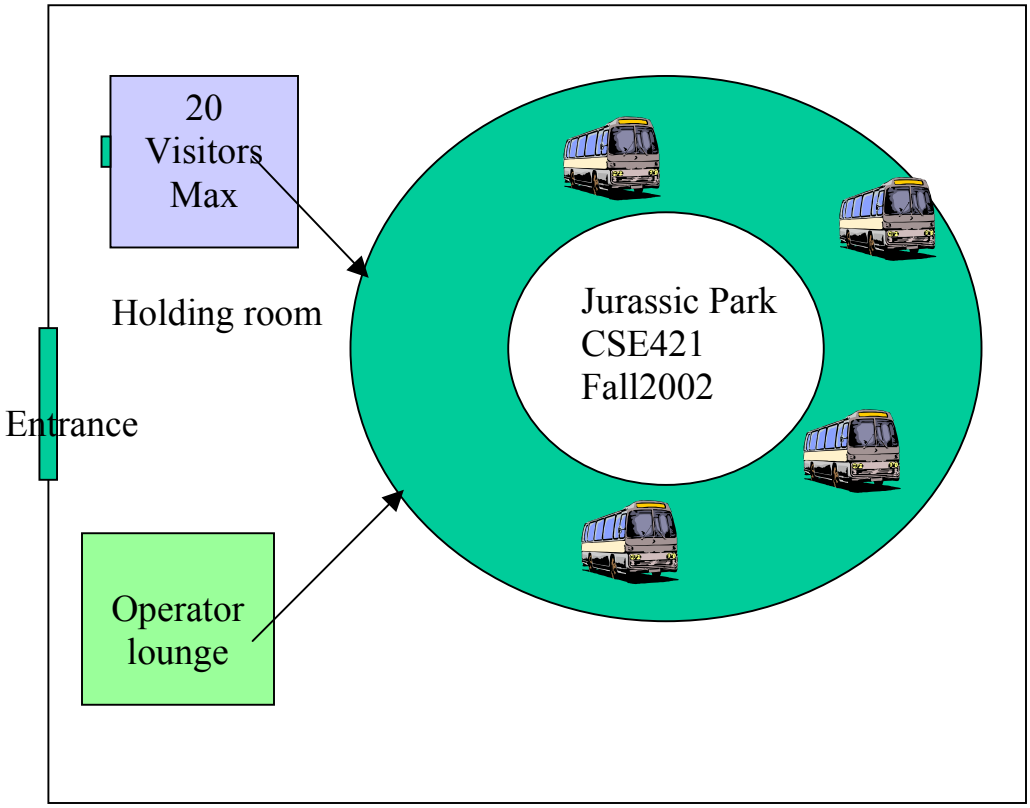

Figure 1: An Approximate Model of the Jurassic park CSE421 Fall 2002