CS 3013 Operating Systems

WPI, C Term 2007

Project 4 – It's Always Wise To Synchronize

This project is to be done by each individual student. It is NOT a group project

Introduction

This assignment is intended to help you put into practice the concepts of synchronization and resource sharing in a realistic setting where a number of "consumer" threads are used to handle requests initiated by "consumer" threads. This situation mimics a server that uses several threads to receive requests and then dispatches these requests to waiting consumer threads. For this assignment you will be using threads along with thread synchronization primitives to coordinate the actions of the threads.

Problem

The basic idea of this assignment is to use the initial (main) thread of your program to generate producer and consumer threads to handle requests. Rather than receive requests from other processes, your producer threads will generate requests at periodic intervals and place them in a shared queue to be retrieved by one or more consumer threads to be serviced. This configuration is shown in Figure 1.

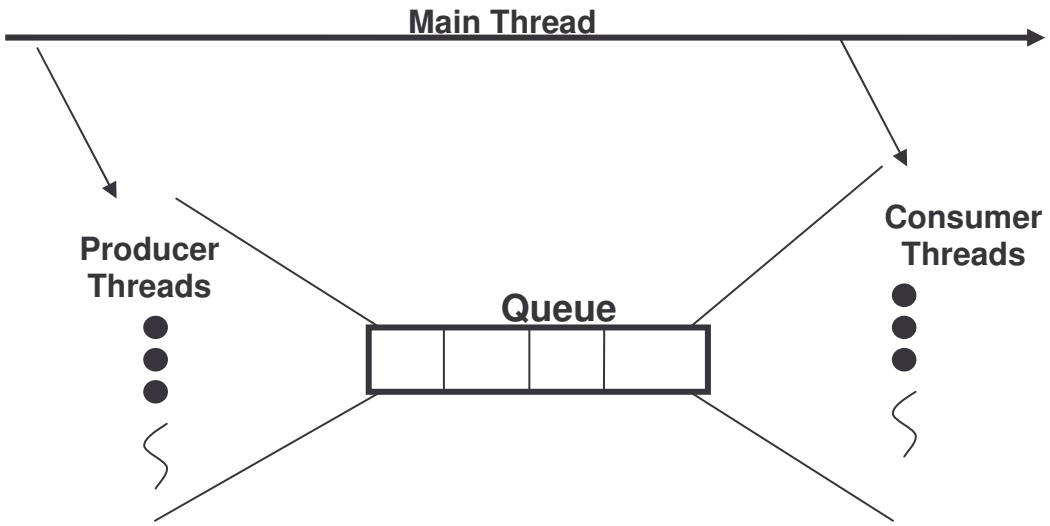

Figure 1: Main Thread, Multiple producers and consumers, and a Queue of size qsize

The rate at which requests are produced is denoted as λ. The rate at which they are consumed is denoted as μ. The default value of both λ and μ for this project is 10 requests/sec, indicating that new requests arrive or are consumed every 1/10 sec. or 100,000μs. Each of these rates is configurable as described later in this document. Just for clarification, these rates are as viewed from the queue; each of the *numproducers*, for instance, will generate requests at a rate of λ/*numproducers*.

Main Thread

When your main thread begins, it first needs to set up and initialize global data structures as well as synchronization primitives that will be shared by it and the other threads. The primary data structure to be shared is the queue of requests as shown in Figure 1. Each request will be denoted by the time when it is created. This value will be obtained via the gettimeofday() system call and stored in a struct timeval structure (defined in <time.h>). The requests will be stored in a linked list, with this list being singly or doubly linked.

Since the number of total requests is allowed to be very large, your producers should generate/allocate entities called queue_entries and your consumers should release/free these entities. These queue_entries should each contain a struct timeval Time, as well as the appropriate forward (and possibly backward) pointers.

Semaphores need to be created to control access to the queue. These semaphores should be created using the semaphore routines available with the pthread library package (see "Note" later on.)

After creating the shared queue and semaphores, your main thread needs to create the *numproducer* producer threads and *numconsumer* consumer threads. These numbers should be set to the value of 1 by default. Again these values can be configured.

Before creating all of the other threads, your main thread should determine the average interarrival time for requests by calculating 1/λ. For example with λ = 10 req/sec, the average interarrival time should be 100ms or 100000μs. It should then calculate the interarrival time as seen by EACH of the producer threads. The same calculation should be done for the service times.

After initialization and thread creation, your main thread is to wait for all other threads to complete (using pthread join) and print out results from this run of the program (see sample output later in this document). The final task of the main thread is to cleanup data structures and synchronization primitives that were created.

Producer Threads

Rather than use a fixed interarrival time, you should use the average interarrival time to generate a random value centered on the average. To do so, you should convert the interarrival time to an integer number of microseconds and use the uniform() routine described later in this document to obtain a uniformly distributed random number. Your threads should use the usleep() call to sleep for the given number of microseconds before they each wake up, record the current time of the request and try to place the newly generated "'request" in the queue. If the queue is full then your threads should block until space is available in the queue.

Remember, it is essential that any thread touching the queue and global variables get semaphores before doing so.

Once the request has been placed in the queue then your threads should randomly determine the amount of time until the next request and again use usleep() to wait the necessary amount of time. Your threads should continue in this fashion until they have generated *numrequest* requests where the default value for *numrequest* is 100.

Consumer Threads

Consumer threads should continually loop waiting for new requests to become available in the queue. You will need to use semaphores to control access to the queue by these threads.

You will be using the Consumer Thread to determine several performance metrics. Remember that the total time that a request lives is made up of its Queue Time and its Service Time.

Total Request Time = Queue Time + Service Time

When a consumer thread removes a request from the queue it should obtain the current time using *gettimeofday*(). Using that current time and time when the request was created, the worker thread can determine the "Queue Time" for this request.

The worker thread next "services" the request by using a uniform random distribution based on the service rate μ. For example if μ is 20 req/sec. then the average service time is 1/20 sec (or 50000μs) and the uniform() function should be called to generate a uniform random value. The worker thread should use usleep() to wait for this amount of time. Once it awakes it should again call gettimeofday() to get the completion time for the request. It should use the completion time and the request time to compute the "Total Request Time" for this request in the system (includes both queue time and processing time). The queue time and total time should be accumulated in shared global variables. Note access to these variables is shared amongst all threads, so access to them needs to be protected. Consumer threads should continue in this fashion until they have consumed *numrequest* items. At this point the threads should terminate normally without adding any values to the cumulative variables.

Basic Objective

The basic objective for the project, worth 80% of the points, is to write the code for the described program with proper use of thread and synchronization primitives. As described here, there are a number of parameters associated with the program. These parameters and their default values are shown in Table 1.

Table 1: Configuration Parameters

Parameter	Description	Valid Range	Default
numproducer	Number of Producer Threads	1 – 10	1
numconsumer	Number of Consumer Threads	1 – 10	1
qsize	Queue Size	1 – infinity	10
numrequest	Number of Requests	10 – infinity	100
arate	Arrival rate, λ req/sec	1 – 100	10
srate	Service rate, μ req/sec	1 – 100	10

The default value should be used for each parameter unless overridden with an alternate value specified on the command line when the program is started. Alternate values are specified with parameter=value (note no space around the "=" sign) pairs. As part of startup your program must process any command line arguments of this form. You may find functions such as sscanf() or atoi useful for this processing. Your code must ensure that all specified parameter values are in the range shown in Table 1. Should any specified value be outside of its valid range then your program should report the error and immediately terminate.

Given this description, the following shows three sample runs of your program, which is expected to compile to the executable "proj4". As shown your program should initially print out the parameter values used. Values for the average request queue and total time should be shown as an integral number of micro-seconds. This output is intended to show the form of the output and not necessarily the correct values.

```
%proj4
Parameter values:
numproducer=1 numcomsumer=1 qsize=10 numrequest=100 arate=10 srate=10
Results:
Average queue time         = 155 useconds
Average total request time = 100158 useconds

%proj4 numconsumer=5 srate=20
Parameter values:
numproducer=1 numcomsumer=5 qsize=10 numrequest=100 arate=10 srate=20
Results:
Average queue time         = 10 useconds
Average total request time = 50000 useconds

%proj4 numconsumer=5 srate=150
Parameter srate exceeds maximum value of 100.
```

Random Distribution

Rather than use a fixed interarrival and service time, you should use a distribution that is uniformly distributed around a given average (mean). In this program you need to use the following *uniform*() function, which returns a random number between zero and twice the given average.

```
int uniform(int avg)/* return a random integer i, 0 <= i < 2*avg */
{
     int range = 2 * avg;
     return( random() % range);
}
```

Random number generators use "seed values" and in order to set the seed value for each process' random number generator you should use the routine srandom() (a man page exists for random(3) and srandom(3)). Use srandom(getpid()); at the beginning of your code to set the random number generator seed to a program-specific value.

What Does It Mean?

It is very possible to do this project and never really engage your brain. Sure you've gotten the code to work – and it works correctly. But this project is REALLY about studying queuing and its behavior. To show that you've engaged your brain on the level above simple coding, include a document describing the meaning of your queuing system. What matters when setting parameters; how does your system ***behave***.

Note

The program makes use of the routines gettimeofday() and usleep(), which appear to provide accuracy to the nearest microsecond. However, due to the granularity of the system clock and other activity on the machine, it is unlikely that such precision will be obtained. For purposes of this project, such imprecision is fine, and you should report the values you obtain. Other useful functions are: sscanf, atoi, sem_init, sem_wait, sem_post, sem_destroy, pthread_create, pthread_join, gettimeofday, and usleep.

Additional Work

For the remaining 20% of the project extend the output to include maximum queue length.

```
Results:
Average queue time         = 10 useconds
Average total request time = 50000 useconds
Maximum queue length = 7
```

Design experiments that will answer the following questions:

a) What is the relationship between λ and μ such that the maximum queue length remains finite? Conversely, what causes the maximum queue length (and the average queue time) to become larger and larger?
b) How should you specify *numproducers* and *numconsumers* so as to minimize the total time for the requests.

Show how you designed your experiments – good design means changing a minimum number of inputs on each test.

Submission

Use the turnin command to submit your project with the project name of proj4.

Evaluation Guidelines
Project 4

Submission Mechanics	**15 total**
You used turnin and followed the rules given above, thus saving the TA's considerable time and effort.	5 points
When we cd into proj4, we can type "make" and the executable is produced from the source(s).	5 points
The code has comments and is structured – it's not spaghetti code.	5 points
Basic Commands	**65 total**
Output format is as specified in the writeup.	10 points
Producer threads create requests using the time distribution *uniform()*	5 points
Consumer threads service requests using the time distribution *uniform()*	5 points
Program responds to all valid parameter values as specified in Table 1.	
Queue size can get very large and not run out of memory.	10 points
Multiple producer and consumer threads	5 points
Many requests without running out of resources.	5 points
Other input parameters	5 points
Test script is included that allows exercise of parameters	10 points
Document submitted that describes the meaning of the created system	10 points
Additional Work	**20 total**
Output format is as specified in the write-up including Maximum queue length.	5 points
Submission includes an additional file describing the details of the experiments conducted and answering the questions raised in the Additional Work section.	15 points