CS 3013 Operating Systems | WPI, A Term 2006
Craig E. Wills | Project 4 (30 pts)
Assigned: Tuesday, September 19, 2006 | Due: Monday, October 2, 2006

Important: This project is to be done by each indivdual student. This 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 "worker" threads are used to handle requests initiated by a primary thread. This situation mimics a server that uses one thread to receive requests and then dispatches these requests to waiting worker 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 handle requests. Rather than receive requests from other processes, your main thread will generate requests at periodic intervals and place them in a shared queue to be retrieved by one or more worker threads to be serviced. This configuration is shown in Figure 1.

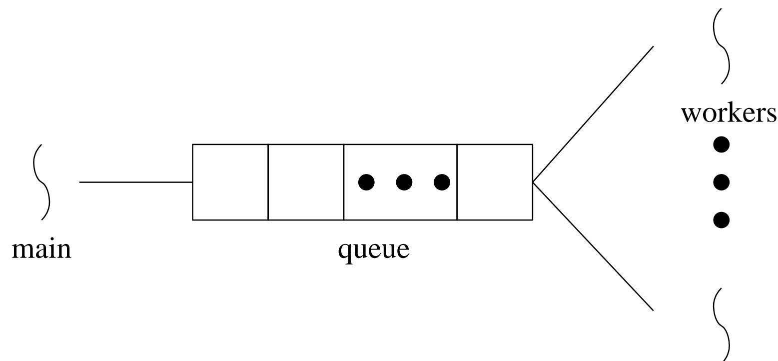

Figure 1: Single Main Thread, Multiple Worker Threads, Buffer of Size *qsize*

The rate at which requests are generated is denoted as λ. The rate at which they are serviced 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 serviced every 1/10 sec. or 100000μs. Each of these rates is configurable as described later in this document.

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 worker 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>`). An array of requests can be stored as an array or a linked list. However, if you use an array then the size of the queue (*qsize*) is configurable and therefore your program must be able to allocate a variable size array. The *queue* array of size *qsize* can be allocated as follows:

```
struct timeval *queue;        /* array of timeval structs */
queue = (struct timeval *)malloc(qsize*sizeof(struct timeval));
```

The C/C++ `sizeof` operator returns the size (in bytes) of the *timeval* structure, which is multiplied by the queue size. Alternately, the C++ *new* operator can be used to allocate the array as in

```
queue = new struct timeval[qsize];
```

The queue buffer can then be treated as an array with indices from 0 to *qsize-1.* Alternately if you use a linked list for the queue then you need to dynamically allocate and deallocate entries as needed. Regardless of your queue implementation, 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.

After creating the shared queue and semaphores, your main thread needs to create the number of worker threads according to the value of *numworker*, which should be set to the value of 1 by default. Again this value can be configured.

After creating all of the worker threads your main thread should determine the average interarrival time for requests by calculating $1/\lambda$. For example with $\lambda = 10$ req/sec, the average interarrival time should be 100ms or 100000μs. Rather than use a fixed interarrival time, you should use the average interarrival time to generate a random value centered around 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 main thread should use the *usleep()* call to sleep for the given number of microseconds before it wakes up, records the current time of the request and tries to place the newly generated "'request" in the queue. If the queue is full then your main thread should block until space is available in the queue.

Once the request has been placed in the queue then your main thread should randomly determine the amount of time until the next request and again use *usleep()* to wait the necessary amount of time. Your main thread should continue in this fashion until it has generated *numrequest* requests where the default value for *numrequest* is 100. At this point, the main thread should generate *numworker* "null" requests (both fields of the *timeval* struct set to zero) indicating that all requests have been generated. These null requests should be placed in the queue without delays. These special requests will be used by the worker threads to know when they are done.

The final portion of your main thread is to wait for all worker 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.

Worker Threads

Worker 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 the worker threads. When a worker thread first removes a request from the queue it should obtain the current time using *gettimeofday()*. Using the 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 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 because access to these variables is shared amongst all threads access to them needs to be protected.

Worker threads should continue in the fashion until they receive a null request. At this point the thread should terminate normally without adding any values to the cumulative variables.

Basic Objective

The basic objective for the project, worth 26 of 30 points, is to write the code for the described program with proper use of thread and synchronization primitives. As described in the handout 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
numworker	Number Worker Threads	1 – 10	1
qsize	Queue Size	1 – 100	10
numrequest	Number of Requests	10 – ∞	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* 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:
numworker=1 qsize=10 numrequest=100 arate=10 srate=10
Results:
Average request queue time = 155 useconds
Average request total time = 100158 useconds

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

% proj4 numworker=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 to 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.

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, you such imprecision is fine and you should report the values you obtain.

Additional Work

For the remaining four points on the project, your program should support the setting of `qsize=-1` on the command line, which indicates that rather than a fixed queue size, your queue size should be unlimited. In this scenario, your program should support an unlimited queue size (subject to dynamic memory demands) and you must ensure that your main generator thread *never* blocks. You may need to rewrite your main thread to support this new functionality.

Submission

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