CS162 Spring 2011 - Project 1

Instructor: Ion Stoica
Stephen Dawson-Haggerty, Jorge Ortiz, David Zhu

1 Project 1: Concurrency and Multiprogramming

Concurrency is a fundamental part of operating systems. Although underlying hardware often supports only one point of execution, all modern operating systems allow this resource to be multiplexed between multiple threads of program control. The goal of this project is for you to use the synchronization primitives available to you to build correct, parallel programs. This assignment is separated into two sections: an individual portion and a group portion. You must complete the individual portion on your own, and hand it in along with your design document. The group portion will be submitted later, and only once per group.

1.1 Individual Portion

Synchronization questions:

1.1.1 Definitions

- Multiprocessing
- Multiprogramming
- Multithreading
- Preemption

1.1.2 Threads

Suppose you have a variable x initialized to 0, and then start two threads running the following pieces of code:

Thread A	Thread B

for (int i = 0; i < 3; i++) {
  x += 1;
}
	for (int j = 0; j < 3; j++) {
	  x += 2;
	}

Assume that load and store instructions are atomic, and that x must be loaded into a register before being increased (and stored back to memory afterwards).

a) What are the possible values of x after both threads finish? Explain at least one way in which each value can be reached.

b) Suppose that in thread B, you replace the line x += 2 with a special, atomic hardware instruction that increases x by 2 (this instruction cannot be preempted). What are the possible values of x after the threads finish now?

1.1.3 Readers and Writers

a) Suppose that we have implemented reader-writer locks using a lock and condition variables as in the lecture slides

Reader	Writer

//First check self into system
lock.acquire();
while ((AW + WW) > 0) {
  WR++;
  okToRead.wait(&lock);
  WR--;
}
AR++;
lock.release();
 	
// Perform actual read-only access
AccessDatabase(ReadOnly);
 	
// Now, check out of system
lock.acquire();
AR--;
if (AR == 0 && WW > 0)
  okToWrite.signal();
lock.release();
	// First check self into system
	lock.acquire();
	while ((AW + AR) > 0) {
	  WW++;
	  okToWrite.wait(&lock);
	  WW--;
	}
	AW++;
	lock.release();
	
	// Perform actual read/write access
	AccessDatabase(ReadWrite);
	
	// Now, check out of system
	lock.acquire();
	AW--;
	if (WW > 0){
	  okToWrite.signal();
	} else if (WR > 0) {
	  okToRead.broadcast();
	}
	lock.release();

(In this code, the variables AW, WW, AR, and WR count active writers, waiting writers, active readers, and waiting readers respectively.)

Suppose that all of the following read and write requests arrive in very short order (while R1 and R2 are still executing): Incoming stream: R1, R2, W1, W2, R3, R4, R5, W3, R6, W4, W5, R7, R8, W6, R9

In what order would the database process these requests? If you have a group of requests that are equivalent (unordered), indicate this by surrounding them with brackets. You can assume that the wait queues of the condition variables are FIFO (i.e. signal() wakes up the oldest thread on the queue).

b) How can you redesign the code in a) to run requests in an order that guarantees that a read always returns the results of writes that have arrived before it but not after it? (Another way to say this is that the reads and writes occur in the order in which they arrive, while still allowing groups of reads that arrive together to occur simultaneously.)

1.1.4 Deadlocks

a) Consider the following three threads executing different paths of execution. Determine if and where deadlock can occur for each thread. For example, Thread 1 can deadlock on Line 2 would mean that Thread 1 can deadlock and wait forever to acquire mutex_e.

Thread 1	Thread 2	Thread 3
lock(mutex_d) lock(mutex_e) e=d*2+e unlock(mutex_e) unlock(mutex_d)	lock(mutex_e) lock(mutex_f) f=f+e unlock(mutex_f) unlock(mutex_e)	lock(mutex_f) lock(mutex_d) f=f*d unlock(mutex_d) unlock(mutex_f)

Complete the following, or write deadlock cannot occur
Thread 1 can deadlock on line <>
Thread 2 can deadlock on line <>
Thread 3 can deadlock on line <>

b) Repeat part a) for the following two threads.

Thread 1	Thread 2
lock(mutex_h) lock(mutex_i) lock(mutex_j) j = i + h unlock(mutex_j) unlock(mutex_h) lock(mutex_g) g=g+i unlock(mutex_g) unlock(mutex_i)	lock(mutex_g) lock(mutex_i) i = g + g unlock(mutex_i) unlock(mutex_g)

c) Given the following processes and their resource allocations, use the bankers algorithm to determine if a deadlock is inevitable; show your calculations. The system has 10 units of X and 15 units of Y.

Process	has_X	may_need_X	has_Y	may_need_Y
1	3	10	2	4
2	0	6	6	7
3	5	5	2	6
4	1	2	5	5

1.1.5 Fork, Exec, and Wait

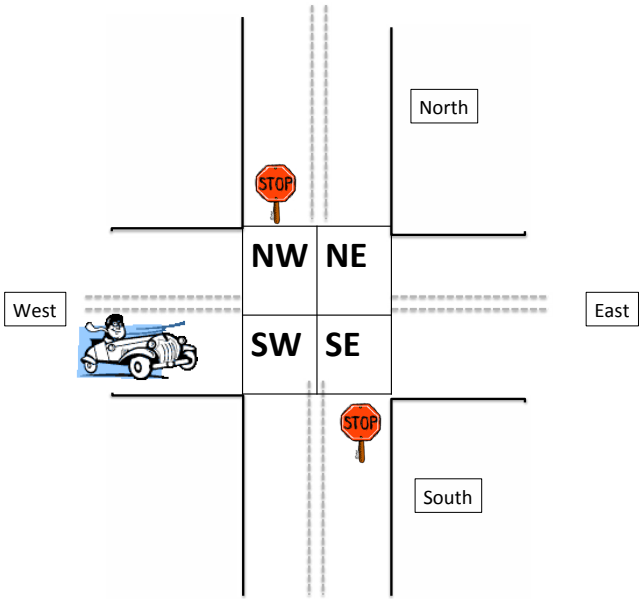
```
1  int main(int argc, char **argv) {
2  FILE *fp;
3  int tid;
4  int waitint;
5
6  fp = fopen("test.txt", "a");
7  if (!fp) {
8    perror("fopen");
9    return 1;
10 }
11 printf("[%i] parent starting up\n", getpid());
12 tid = fork();
13 if(tid > 0) {
14   printf("[%i] parent about to wait\n", getpid());
15  fprintf(fp, "[%i] parent wrote to file\n", getpid());
16   fclose(fp);
17
18   wait(&waitint);
19   printf("[%i] wait returned: wait_int: %d\n", getpid(), waitint);
20   fclose(fp);
21 } else {
22   char *exec_argv[3] = {"external", "hello from exec!", NULL};
23
24   printf("[%i] child running\n", getpid());
25   fprintf(fp, "[%i] child wrote to file\n", getpid());
26   fclose(fp);
27
28   execv("external", exec_argv);
29 }
30 return 0;
31 }
```

Read the man pages for fork, exec, and wait on a linux machine (i.e. type 'man fork'). Now examine the code above or run the code (forkexercise.c). The code provided is the same as the one shown in the document (above). Once you familiarize yourself with the code, please answer the following questions.

1. Write out the contents of what is printed to the screen.
2. Write the contents of the text file.
3. If we were to use `pthread_create()` instead of `fork()` would the file output change? Why or why not?

The code can be downloaded from the courses web site
`http://inst.eecs.berkeley.edu/~cs162/sp11/projects/p1/indiv/forkexercise.c`
`http://inst.eecs.berkeley.edu/~cs162/sp11/projects/p1/indiv/external.c`

1.2 Directing Traffic



Synchronization problem: We will use locks to direct traffic. Cars come to the intersection and either continue in the direction they are driving, make a left turn, or make a right turn. Each quadrant of the intersection has a lock associated with it. No two cars should ever be heading in directions that cross. Keep in mind that there cars that need to make a left turn will be moving across the intersection. This path should also never intersect with another trajectory (i.e. EAST-to-SOUTH intersecting with SOUTH to NORTH or NORTH to SOUTH).

We have provided you with code (`intersection.c, intersection.h`) that uses a lock for each quadrant and prints out the actions taken by a car. **Examine the code carefully.** Before the action is printed, you must first acquire the necessary locks. Please fill in the missing code. The code uses the POSIX library. Read the necessary documentation on the use of lock in POSIX and finish the code so that there is no deadlock. If the code exits successfully each time, the locking strategy was successful.

Code:
`http://inst.eecs.berkeley.edu/~cs162/sp11/projects/p1/indiv/intersection.c`
`http://inst.eecs.berkeley.edu/~cs162/sp11/projects/p1/indiv/intersection.h`

2 Group Portion: Thread-safe Chat Server

Now that you have gotten your feet wet with using some multi-programming primitives we will start applying those concepts in the implementation of a concurrent thread-safe chat server. Thread safety means that the system should provide each concurrent user a consistent view of the system. The code written in this assignment will serve as the core logic for the machinery behind distributed chat server.

2.1 Specification overview

The core functionality is subject to various constraints, highlighted in this specification. The key concepts to be modeled are users, groups, messages and the interaction between them. Users are created by simply logging on to the system. They can concurrently create, list, and join groups, as well as send messages to an individual or a group. A group contains one or more users. Each user and group has a unique name associated with it.

One of the key challenges is to ensure concurrent safe access to the data. You are expected to use the synchronization primitives learned in class to ensure this. We will test how your system will respond under heavy number of users and all of them performing concurrent operations such as sending messages.

At a high level we require the following:

1. Users enter and leave the chat server at any time.
2. A user can choose to join multiple chat groups.
3. User and chat group names must be unique and concurrent reading of these names are allowed.
4. Message delivery must be in the order that is received. No two users in the same group shall see two messages in a different order.
5. Senders and receivers can log off or leave a group while a message is in transit and graceful error handling is expected.

To facilitate auto testing, we have provided some skeleton code for you to start with. It is available for download from the project page as a tar file. You are expected to fill in ChatServer.java with necessary functionalities, create your own ChatUser class which extends from a BaseUser class that we provide. You are also free to create any new classes that might be necessary for your design.

2.1.1 Part A - Initial Design

In this part of the assignment, we want you to write your design document. The specification and requirements are broad by design. However, we do require a few mechanisms to be in place so that we can standardize the testing process across the code from different teams. The code we have included with the assignment consists of the following classes:

ChatServer The `ChatServer` object is the front-facing class which will be organizing access to the chat groups. It handles admission control and chat group operations. It should also direct message passing between users. The ChatServer process runs as its own thread. It is also the class that we will be interfacing with our test harness. It needs to implement ChatServerInterface that we provide.

User This object represents an active user in the system. Each user process runs as a separate thread.

Read and familiarize yourself with the classes that we have provided. Each user is expected to handle the sending and receiving of messages. In addition, the chat server should handle incoming messages and forward them to valid, registered users that are logged on as well as chat groups, groups of users that share a broadcast domain.

Our tests will consists of the following steps:

1. A user is created and assigned a random name.
2. The user will identify itself with the chat server and attempt to join as a new user.
3. The server checks if the max number of concurrent users have been reached, if not, accept the user. If it is at capacity, place the user on a queue to wait.
4. Upon acceptance, the user will query for the list of users and groups that are currently logged in.
5. The user may choose to become a member of an existing chat group or create a new one.
6. The user will randomly choose a user or group as the destination of a message and attempt to send a message.
7. The above steps will repeat until at some random point in time the user will log out.
8. Once a user logs out, if there is a user waiting on the queue, the waiting user will be accepted on a FCFS basis.

Your design document should use the testing steps to guide its design. We require at least two types of threads – users and chat server – but you may decide to introduce more. Please describe all the threads in your design, the placement of message buffers, and the synchronization mechanisms you use to allow the threads in your system to communicate safely.

Please keep in mind that all buffers sizes are finite and your design must handle the case where a message queue is full, a group has reached the maximum number of members, the chat server has to many messages enqueued, etc. Your design document should include some discussion of how this will be handled.

2.2 Part B - Interaction with the Chat Server

At this point your design should have been approved by your TA and you should start implementing. We're going to test the specification requirements in separate phases. This phase will focus on testing user log-in and log-out, chat group creation and deletion, and querying the chat server for information about the state of the chat server – how many users are logged in, how many groups their are, the list of current users and the list of current chat groups – as the chat server is active.

Make sure that your code handles the following.

- A user calls the authentication method on the chat server. If the user's name is unique, the user is admitted to the system.

- A user calls the method to create a group. Group names must be unique.
- A user request the number of users/chat group in the system. If the user is not valid, the request is rejected.
- A user requests a list of users/groups in the system. If the user is not valid, the request is rejected.
- A user requests to join a group. A user may join multiple groups at once.
- A users logs out.

When a user create a group she automatically becomes a member of the group. The group is deleted only after every user has left the group.

There can be no more than 100 users logged in at any time and the maximum number of users in a chat group is 10. If any part of the system is at capacity, you must handle the situation accordingly – either by including a wait queue or rejecting the request explicitly. If a wait queue is implemented it can be no larger than 10% of the maximum number of users allowed the group or chat server. Be clear about which choice you have made and the mechanisms you have employed for dealing with requests when at maximum capacity.

2.3 Part C – Message delivery

Now that we have the core functionality in place for adding and removing users and chat groups, write the code that interprets messages that come in from the user. Sending a message to a user/group that does not exist (is not currently in the system) should fail explicitly.

Every user, whether in a one-on-one conversation, or in a chat group, should see the same messages in the same order as every other user, with one exception. **Each user should maintain a log of the conversation(s) they have been involved in.** Make sure that each message sent and received in timestamped, and includes the source, destination, and the message itself. If a user sends a message and that message is not successfully received or posted to the group, it should appear only on the sender's message log with the error flag set in the message.

We will test the following:

- A user can send a message to another user.
- A user can send a message to the group she belongs to.
- A user receives message successfully from the group.
- A user receives messages successfully from individual users.

If a user is not part of a group, they can neither receive nor send messages to/from that group. If a user logs out while a message is in transmit, the receiving user should not receive that message.

2.4 Part D - In-order delivery

In this part of the assignment we ask you to write the code that ensures the each user receives each message that enters a group in the same order as the other members of that group. We also carefully design a strategy for ensuring the order of various activities and to prevent the loss of a message or a message being received by an unintended receiver.

We want to ensure the following message delivery semantics.

1. Messages must only be consumed in the order they are received. Message order must be preserved across all users in a group and between pairwise user conversations.

2. Only the current members of the group can consume messages posted to the group.

3. The order of events and message posts should be consistent with the expected message-delivery semantics (i.e., a user cannot see a message that was posted before she joined the group).

Preserving these semantics are important for assuring the expected behavior in a chat system. All users should see messages in the same order, and this order-preserving property is especially important in chat systems where the order constitutes the meaning behind the conversation. You certainly would not want a user to be confused about the conversation because the message order was different from that of other members in the chat group.

The next two constraints are about correctness.

1. Why are the last two constraints important for correctness?

2. Can a message be lost if these constraints are not maintained?

You may used a combination of synchronization primitives to ensure that these are constraints are being successfully maintained.