Lab 2: The dining philosophers

Magnus Johansson

April 23, 2007

1 The assignment

You are to write an implementation of the dining philosophers problem. See page 209 (6th edition) or 207 (7th edition) in the textbook for a description of the problem. Your program should take the number of philosophers as an argument and it should be deadlock free. See section 8.4 (6th edition) or 7.4 (7th edition) in the textbook for more information on how to prevent deadlocks. As before you are free to write it in any programming language you wish, as long as you can do POSIX system calls from it. The assignment has been prepared for Java so you will get some code for free if you choose to do it in Java.

There are two main points of this assignment: The first is to make you comfortable with semaphores. You should read up on semaphores in the textbook before doing the lab. There are two different interfaces to semaphores in the POSIX standard: POSIX 1003.1b semaphores and SystemV semaphores. SystemV is the older of the two, and very difficult to learn. 1003.1b has a cleaner interface. The 1003.1b interface is the natural choice for learning semaphores. However, since the jtux interface is broken for 1003.1b semaphores, this lab will use SystemV semaphores. To compensate for the very difficult interface of SystemV, there is a simplified interface included in the lab package that is similar to, but not identical to the 1003.1b interface. See Semaphore.java for details. It contains the normal create, destroy, wait, and signal operations. The second point of this assignment is to use shared memory.

If you choose to do the lab in C you should use the 1003.1b interface to make your life a bit easier. See the man page for `sem_wait` and `sem_init`, among others.

The lab package contains four java files:

- `Semaphore.java` contains the simplified interface to SystemV semaphores. You will probably not have to modify this file.
- `Rice.java` contains the rice bowl that the philosophers will eat from. You will have to modify this to use shared memory.
- `Philosopher.java` contains the code that a single philosopher will run. It is here the philosopher will pick up the chopsticks, eat, and put them down.
- `Philosophers.java` contains the main program that will create the semaphores and the philosophers and set things up. The philosophers will be children to this process.

For this lab you will need to modify Philosophers.java, Philosopher.java, and Rice.java. Here's a suggested implementation order:

1. Familiarize yourself with the given skeleton. Read through all the files and make sure you understand the general idea. Then start by modifying Philosophers.java. You should create the rice bowl and the necessary semaphores, and fork children to run the philosopher process (use the run() method of the Philosopher class). Don't forget to wait for the children and to destroy the semaphores and the rice bowl afterwards. The source code contains more information. Take a look at the first lab for more information on forking and waiting.

   Run the program by typing `./philosophers.sh 5` if you want five philosophers. If you run your program at this point the philosophers will eat, but they will do so without using the chopsticks. Also, the skeleton rice bowl given will behave as if each philosopher had its own rice bowl. When the program finishes you should see that they all have eaten the same amount of rice.

When the program is running, take some time to look at the semaphores from the command prompt. The command to use is `ipcs`, which stands for Inter Process Communication Status. Semaphores are a part of the IPC system. You should see your semaphores there, including one for the rice bowl. If your semaphores remain after your program ends you are doing something wrong. Make sure you destroy your semaphores when you are done with them. You can manually clean up any remaining semaphores with the command `ipcrm`. See the man pages for ipcs and ipcrm for details (type `man ipcs` and `man ipcrm`, respectively). There is also a script in the lab package called `ipc_clean.sh`. This script will clean up all your ipc-resources and it can be very useful if your program is buggy and doesn't properly clean up itself.

2. Modify Philosopher.java. In this file you need to grab and put down the chopsticks. See the file for details. When you are done with this step you should have a deadlock free implementation of the dining philosophers.

3. Finally you should modify the rice bowl to use shared memory instead so that the philosophers will share the same rice bowl. Take a look at Rice.java for more details. The system calls you will have to use are

   - USysVIPC.shmget()
   - USysVIPC.shmat()
   - USysVIPC.shmdt()
   - USysVIPC.shmctl()

   Take a look at the jtux documentation and the man pages to get the full information about these. You will probably also want to use some predefined constants:

   - UConstant.IPC_PRIVATE
   - UConstant.IPC_CREAT
   - UConstant.IPC_RMID

   These are described in the man pages of the system calls above.

   You will also have to use two helper functions if you do this assignment in Java:

   - UUtil.jaddr_from_seg(a, b, s);
   - UUtil.jaddr_to_seg(a, b, s);

   These two functions will read from and write to the shared memory. Normally if you write C-code this step is unnecessary since from C you can access to shared memory directly once it is attached. However, in Java you can not since machine dependent stuff is abstracted away. These two functions are a part of the jtux package, but are not in the Posix specification. The first will read s bytes from the shared memory attached at address a, and store them in the byte array b. The second will store s bytes from the byte array b to the shared memory attached at address a. See the jtux documentation for more information.

   Don't forget to protect the shared memory with semaphores! Otherwise two philosophers may eat the same piece of rice, and the total amount eaten will be greater than the original amount in the rice bowl. You will also need to remove the shared memory when you're finished with it. Otherwise it will remain even after all of your processes have finished. You can use ipcs from the command line to see if you have any shared memory or semaphores remaining after your program has finished.

Some hints:

- Make your program very verbal. It it much easier to see what is going on if your program does a lot of output.
- Slow down your program to encourage deadlocks and to make it easier to see what happens. There is a line in the skeleton that shows how to pause for a random amount of time.

2 How to hand in

Send an email to `magnusj@it.uu.se` with the source code attached. Also list all the participants.