cs452 – Spring 2016
Kernel 2

Bill Cowan
University of Waterloo

I. Introduction

In the second part of your development of the kernel, you make it possible for tasks to communicate by message passing. You use this capability to create a name server. By the end of this part of the kernel you must have:

1. working implementations of `Send`, `Receive` and `Reply`,
2. a running name server, created by the first user task, and
3. implementations of `WhoIs` and `RegisterAs` as wrappers for `Send`s to the name server.

As discussed in class you must choose a method by which every task knows the task id of the name server, which is its bootstrap into the world of services provided by servers closely associated with the kernel.

In addition to the kernel primitives you must program a server and some clients. The server is the custodian of a Rock/Paper/Scissors game; the clients play the game against one another by interacting with the server.

Finally, we would like you to measure the time (in microseconds) for Send/ Receive/Reply transactions in sixteen conditions generated by Send/Receive order (Send before Receive and Receive before Send), message size (four (4) and sixty-four (64) bytes, caches (off and on), O2 optimization (off and on).

You may find that your kernel doesn't run with O2 optimization on. If so, don't worry: just send results with optimization off. Usually this means that the optimizer is re-allocating registers you allocated by hand. Judicious use of volatile often solves this problem.

II. Description

II.1. Kernel

To accomplish this part of the kernel you must have the following kernel primitives operating:

- `int Send( int tid, void *msg, int msglen, void *reply, int replylen)`,
- `int Receive( int *tid, void *msg, int msglen )`, and
- `int Reply( int tid, void *reply, int replylen )`.

See the kernel description and the lecture notes for the details of how these primitives should operate.

II.2. FirstUserTask

In addition you must program the FirstUserTask, which initiates the application. It, or its children,

- creates the name server,
- creates the Rock/Paper/Scissors server, and
- creates the Rock/Paper/Scissors clients.

The interaction between these tasks tests your kernel and name server.

II.3. Rock/Paper/Scissors (RPS) Server

The RPS server should accept and provide service for the following three types of request.

1. Signup. Signup requests are sent by clients that wish to play. They are queued when received, and when two are on the queue the server replies to each, asking for the first choice.
2. Play. Play requests tell the server which of Rock, Paper or Scissors the choose on this round. When play requests have been received from a pair of clients, the server replies giving the result.
3. Quit. Quit tells the server that a client no longer wishes to play. The server replies to let the client go, and responds to next play request from the other client by replying that the other player quit.

II.4. Rock/Paper/Scissors Clients

Clients that play the game should

1. find the RPS server by querying the name server,
2. perform a set of requests that adequately tests the RPS server,
3. send a quit request when they have finished playing, and
4. exit gracefully.

The game should pause at the end of every round of the game so that the TA can see what happened. `bwgetc( )` is handy for this.

Unless the opposing player is very stupid, a client cannot do better than playing randomly.

II.5. Performance Measurement

It is already possible to measure the performance of the most performance critical parts of your kernel. The following describes a few tests that you must do, handing in the results so that I can analyse them and let you know where you stand. (It also demonstrates what may be some counterintuitive facts about what influences performance the most.) In doing the measurements you examine the effect of

- instruction order,
- data and instruction caches on or off,
- message size, 4 bytes or 64 bytes, and
- compiler optimization, none or O2.

To perform the test instantiate exactly two tasks: one task sends a message to the other, which receives it and then replies. They should repeat the exchange often enough that you get an estimate, accurate to at least 5% of the time taken by Send/Receive/Reply. You may measure time any way you like, from the clock on the wall (Ontario Hydro time) to the 40-bit instruction counter. However, in the data file you send to me (wmcowan@cgl.uwaterloo.ca) the unit of time must be microseconds. The measurement procedure should be repeated sixteen times, once for each combinations of factor shown below. Please submit your results by e-mail to cs452@cgl.uwaterloo.ca. They should be 16 plain text records. Each record should have six fields, tab separated. The first four fields should contain the exact entries given below; the fifth field should identify your group; and the sixth field should contain the time, in microseconds, that you estimate for your Send/Receive/Reply in the condition specified.

The table you send me should exactly resemble the one below with the group and time columns filled in.

Group	Message length	Caches	Send before Reply	Optimization	Time
	4 bytes	off	yes	off	
	64 bytes	off	yes	off	
	4 bytes	on	yes	off	
	64 bytes	on	yes	off	
	4 bytes	off	no	off	
	64 bytes	off	no	off	
	4 bytes	on	no	off	
	64 bytes	on	no	off	
	4 bytes	off	yes	on	
	64 bytes	off	yes	on	
	4 bytes	on	yes	on	
	64 bytes	on	yes	on	
	4 bytes	off	no	on	
	64 bytes	off	no	on	
	4 bytes	on	no	on	
	64 bytes	on	no	on	

Hints.

1. In the message length conditions, x byte message means x bytes sent and x bytes replied.
2. Use priorities to ensure the order of Send and Receive.
3. Optimization off means no optimization flag; optimization on means using the O2 flag.
4. A few kernels break when O2 optimization is turned on. If your kernel breaks

put ‘broken’ in the time field of the optimization on records.

III. Hand in

Hand in the following, nicely formatted and printed.

1. A description of how to operate your program, including the full pathname of your executable file which we will download for testing.
2. A description of the structure of your kernel so far. We will judge your kernel primarily on the basis of this description. Describe which algorithms and data structures you used and why you chose them.
3. The location of all source code you created for the assignment and either a set of MD5 hashes of each file or an SHA1 hash of your repository. The code must remain unmodified after submission until the assignments are returned. Therefore, you should start kernel 2 with a copy of your kernel 1 results.
4. A listing of all files submitted.
5. A short description of what priorities you chose for the game tasks, and why you chose them.
6. The measurements you made, and a brief explanation of where in your code you think the time is being spent.
7. Output produced by your game task and an explanation of why it occurs in the order it does.

In addition, please remember to send the requested table to cs452@cgl.uwaterloo.ca.

IV. Comments on Timing

In the last few days I have had a few conversations with students about timing, which have gotten me thinking.

IV.1. Units

Recall. I asked you to send me your measurements in microseconds, not in milliseconds or in clock ticks. That’s because I want to combine and compare the numbers you send, and it makes no sense at all to add together numbers measured in microseconds and numbers measured in ticks of unknown duration. So please do as I asked.

Other things, however, could be done. If a asked you for ratios of times such as, ‘By what factor is one time greater than another?’ I get numbers that are unit independent. One execution time is twice as long as another whether they are measured in microseconds, centuries, or ticks; we just have to keep the units consistent. It’s the same if I calculate percentages, or any quantity that divides out the unit.

If, however, I were to measure one quantity in microseconds and the other in ticks, the ratio would have units like microseconds per tick, which is a calibration factor of some sort. For example if you were to measure the *same thing* twice, once in microseconds, once in ticks, the ratio would be the number of ticks in a microsecond, just the thing you need to turn all your measurements in terms of ticks into measurements in terms of microseconds, avoiding having to measure a second time.

I hope the comments so far mean that you know how to get the same measurement in ways as different as running a million SRRs (SRR = Send/Receive/Reply) while looking at the wall clock or turning on a timer to a known frequency while you run a few SRRs.

IV.2. Precision

Whatever method you choose you should give at least a few minutes thought to measurement precision. If you were, for example, to run ten SRRs while watching the second hand of the wall clock the result would be uselessly imprecise. On the other hand, building a network stack so that you can run an ntp client that adjusts the multiplier used with your crystal oscillator so that you aree with an atomic clock run by a standards institute is sending a man to do a boy's job.

So, 'How precise should your measurements be?' Here are a few things I have said in class that might come into your thinking.

1. You will see a speed-up of about 2.5 times with compiler optimization and ten to fifteen time when you turn on the caches.
2. The fastest SRR times are usually less than ten microseconds; the slowest ones are sometimes as long as a millisecond.
3. By the time you are doing your train project you will definitely want message copying that is better than old faithful:
   ```
   while ( *++t = *++s ) ;
   ```
4. Your train project will spend a lot of its computation time inside the kernel with interrupts disabled so your kernel better be fast.
5. At top speed the trains run about 50 cm per second.