CS452 – Spring 2015
Kernel 2

Bill Cowan
University of Waterloo

A. 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. To do so 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 Sends 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 and Receive before), 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.

Comment. It is not uncommon to discover weaknesses in the first part of the kernel as you work on the later parts. So it is good strategy to start early.

B. Description

B.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.

In addition you must program a first user task, which

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

B.2. User tasks.

The following user tasks test your kernel and name server. They should be created by the first user task.

B.2.1. *Rock/Paper/Scissors (RPS) Server*

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

- 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.
- 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.
- 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.

B.2.II. *Rock/Paper/Scissors Clients*

Clients that play the game should

- find the RPS server by querying the name server,
- perform a set of requests that adequately tests the RPS server,
- send a quit request when they have finished playing, and
- 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 very handy for this.

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

B.3. Performance Measurement

You must also perform a simple performance measurement of your kernel so far. The measurement is also intended to show you how several important factors influence the performance of your kernel. The factors we wish to examine are

- Receive before or after Send,
- data and instruction caches,
- message size, and
- compiler optimization.

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

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 the unit of time must be microseconds. This 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.

Message length	Caches	Send before Reply	Optimization
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.

- In the message length conditions, x byte message means x bytes sent and x bytes replied.
- Use priorities to ensure the order of Send and Receive.

- Optimization off means no optimization flag; optimization on means using the O2 flag.
- 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.

C. 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.

* You should be accumulating your hand-in documents as you go along because we expect a description of your complete kernel when its development is complete. For now hand-in only a description of what is new or changed.