Michael Sanchez
CS 502 – Cisco
Project 4- Readme

[Project write-up omitted. HCL]

I initially tested my program using the 7 provided test cases. I then wrote my own test program to try to really hammer the code. My test program uses a number of #defines to control how it should behave. In essence, the initial process creates a random # of processes controlled by AVG_PROCS_PER_LEVEL. Each of these processes then creates processes similarly to a maximum depth defined by NUM_PROC_LEVELS. Each process that is spawned this way then creates a random # of threads controlled by AVG_THREADS_PER_PROC. Each of these created threads continues on while the "master" thread exits after sleeping a bit of time. This sleep keeps the pid valid so it can be sent to. Each thread maintains a list of known process ids. This list is initially seeded with its parent process. Some checks are in place to prevent certain processes from being added to this list such as pid 1, 0, and the shell process that started the program. Each thread then sends a random number of commands at a random interval controlled by AVG_CMDS and AVG_CMD_DELAY respectively. When a thread executes a command, it randomly chooses to send, receive, receive all, stop the mailbox, or exit completely. The overall probability of each command is controlled by the WEIGHT_XXXX defines. When a thread sends, it chooses a random process it knows about to send to and sends it a random process id that it knows about. When a thread receives, it "learns" the process id of the sending thread, as well as the process id it got sent. If it receives all, it does this, but for all messages in the mailbox. Stopping just stops the mailbox, and exiting just causes the thread to prematurely quit. Using this program, you can create a large # of randomly acting threads that send messages back and forth. I found it VERY easy to completely cripple the machine as you end up with approximately (AVG_PROC_PER_LEVEL ^ NUM_PROC_LEVELS) * AVG_THREADS_PER_PROC threads which quickly becomes a very large number, particularly as the number of proc levels expands. Be warned ☺. Using this program with some moderate numbers I was able to root out several very subtle race conditions. Additionally, when sending and receiving syscalls are made, I chose to not have them block. This was to prevent processes from getting stuck sending or receiving off each other which was causing the processes to never exit. The can be changed by setting the SEND_BLOCK and RECEIVE_BLOCK defines. I also utilized the unimplemented syscall 224 (gettid) to get a thread pid to pid mapping – mostly for debugging purposes - since I found threads ended up looking like their thread pid when doing things via current->pid in the kernel.