Rami Bitar Lab 3 answers

3: In my tests where I created several processes of equal priority to run infinitely, one thing I saw that confused me is that the null process was being given time to run despite the running processes both having higher priority which is very odd. The processes I created had roughly equal running time. When I tested processes with different priority the context switch from main it ended up hogging the cpu and the print statement telling me the total cpu time was never printed.

4.3.1: Based on testing I can conclude the processor has reached a fairly even state with the processes not having equal but similar cpu times the range grows as total number of loops increase. Interesting note the 2nd process (cpubnd2) finishes first every time its run and is called by the scheduler most often.

4.3.2: With I/O bound testing it was just a succession of passing the cpu to the other processes, switching between a few of the iobnd processes while had yet to complete and then between each other and the null process when all 4 of them I would assume were asleep. And because none of them ever consumed cpu cycles the inner loop hardly ever printed anything and the amount of prcpuused was almost nothing.

4.3.3: From what I could gather from it processing quickly and testing with various values The I/O bound processes were very close in terms of cpuusage (well within 1 second of each other). The same goes for CPU bound processes (well within 1 second of the other). The I/O bound processes did consume significantly less cpu time than the CPU bound processes.

NOTE For Problem 4: I had to lower the iterations several times because the cpu usage printed and the end of the process would disappear from the terminal due to the size of the output. Feel free to increase the total iterations for testing.

5. In the attacker process I created a function takeover() which prints takeover succeeded 6 times and then returns. Inside the main function of stacksmashA from Lab 2 I created an extremely array and filled each entry in the array with a pointer to takeover(). I experimented with what numbers worked to takeover stacksmashV but if the number was too small (lower than 1000) it would just crash XINU and restart. If the number was too high the same thing kept occurring so I decided on 1024 because it is the size of the attacker's stack. Unfortunately the Takeover is run several times, 18 times I think. I tried reducing the size of the array to prevent it but it did not work. Afterwards a Xinu trap activates. The simple way to protect a process from this kind of attack is by adding basic memory protection to prevent a process from using memory not allocated to it. In the case a process tries it is immediately terminated and the OS can run as normal freeing the memory used by the offending process.

Bonus:The simplest solution is to create a 2nd readylist which holds only RT processes Which will always be checked by resched first and if it is empty then it will enter the normal (TS) process readylist and select one from there. The priority of RT processes within its own table will be the same as the TS table, based on cpu time used. The RT list unlike the TS list will not need a null process because the TS list already contains one. Another Option is to make TS processes have a capped priority and RT a minimum priority is 1 higher than the TS cap. With this RT processes will always run first as long as they don't give up the CPU.