Project 1: Leap Frog Pseudo-Random Number Generator

Siddhartha Datta Roy and Vishnu Rajeevan

October 4, 2012

Implementation

We have implemented the leap frog method in kernel code. Executing the leap frog method in kernel space versus userspace technically should lead to timing and performances boosts.

The basis of our implementation depends on each thread of a process writing to the proc file before any reads are done. Each time a write is done, the code checks if the thread's parent process is already stored as a process that is already requesting random numbers. If it is not, the thread's parent process is saved in a linked list along with the seed that is requested for that thread. The linked list then points to a linked list of all the threads, that the process owns, that are requesting random numbers. When a random number is requested, the thread is cross referenced against the aforementioned linked list. If the process/thread exists, then, based on the seeding and previous random numbers given to that process/thread, a new random number is outputted. If the thread has not previously requested a random number, then a default value is given. The random number generator is just seeded with the default value of A.

There are a few limitations with this module. First, if the thread is reading before writing, it is always seeded with the same number. Thus, the random numbers generated are always the same. Also, another limitation is that all threads have to write before reading in a process. A thread cannot mix reading and writing since all threads in the same process require the same seed and to generate random numbers, the number of threads in the process needs to be known.

One design decision we made is to check if the seed input is a long in the user code rather than the kernel code. This decision was made so that if a non numerical value was given the module would either have to error or choose a default value for the seed. We did not want to have to choose a default value and wanted to error. This is more efficient to do in the user space as there is no context switch that is required when the error is given in the user space. Another design decision that was made was to not permit threads from being deleted from the PRNG. The reason for this was if a thread is deleted, it would bring about the issue of having to restart the random number generation. This is due to the fact that the random number generated for each thread is dependent on the number of threads in each process. This decision only causes performance issues if a very large number of threads are run at a time. Otherwise there is no issue. Another decision we made was to only allow the first thread of a particular process set the seed. This is to simplify the process of initializing processes. When we ran the code we wrote, to create 100 random numbers using 100 threads, it took 0.02 seconds. When we ran the reference code in user space with a 100 threads to produce 100 random numbers, it took 0.01 seconds. Thus our implementation of the kernel side PRNG was not as efficient as the user space PRNG. This could be a result of the fact that writing and reading to proc files is not the most efficient way that the user can interact with the kernel.