CREATED: design.txt

PROGRAMMING ASSIGNMENT 2

CMPS 111 Spring 2013

Authors: Andrew Bao, Konstantin Litovskiy, Tyler Esser & Nick Wood

User space thread scheduler:

Our thread scheduler implements a lottery scheduling scheme. The threads are stored in a custom linked list data structure, with nodes containing each threads ticket counts, and pointers to the threads.

The lottery is implemented by adding randomizing on the total sum of tickets between all threads and then iterating though the linked list subtracting the number of tickets of each thread from the randomized value. When the value falls below 1, the current node’s thread is chosen.

This implementation is basically an array implementation of assigning an index in the array to a thread, and randomizing on the sum of indexes. However, this would require the use of a large array, and extra data structures. Our implementation has the easy of such a structure without the overhead.

Take for example this list of threads:

| List |

| Head | -> | ID: 1 | -> | ID: 2 | -> | ID: 3 |

| Tickets: 5 | | Tickets: 3 | | Tickets: 5 |

The array implementation of the lottery schedule looks like this:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

| ID:1 | ID:1 | ID:1 | ID:1 | ID:1 | ID:2 | ID:2 | ID:2 | ID:3 | ID:3 | ID:3 | ID:3 | ID:3 |

Randomizing on the # of indexes gives us the index of the chosen thread, such as 6.

If we first lump the indexes of the same thread together such as the following:

| 1 - 5 | 6 - 8 | 9 - 13 | or | 5 | 8 | 13 |

| ID:1 | ID:2 | ID:3 | | ID:1 | ID:2 | ID:3 |

We can see that this looks very similar to the data stored in the list, and instead of jumping to an index directly; we can get to the index by subtracting the values of each thread as we pass it.

So to get to 6, we visit ID:1 and remove 5 tickets from our index, this leaves us with 1 ticket. Visiting ID:2 we remove 3 tickets, leaving us with less than 1 ticket, and thus choosing thread ID:2 as we would with an array.

Our program “theApp” generates a random number of threads between 1 and “NUM\_THREADS”, and for each thread generates a priority value from 0 to 39. Each of these threads is then created using the “thread\_create()” function. After main has finished creating threads, it sits in a while loop waiting for all of its child threads to finish before quitting itself. Our application treats main as the first thread.

Each thread runs a function which consists of a loop that will run for a random # of iterations, this number is determined by “THREAD\_RUNS”. After the function has gone through all of its iteration it exits. During the course of each iteration the function has a 10% chance of yielding itself. We chose to allow our functions to yield and exit to better replicate a real scheduler’s duties.

The thread lottery is initiated when a thread either exits, yields itself, or is yielded by the scheduler using a timer interrupt to enforce the quanta. If the thread exited, it is removed from the scheduler before a new thread is selected, if a thread was yielded it remains in a scheduler and a new thread is chosen.