William Kelley

CS415-Operating Systems

Thread Homework

Question 1:

List the reasons why a mode switch between threads may be less expensive computationally than a mode switch between processes.

Switching process requires OS to process more information.

Memory is shared by threads, so there's no need to exchange memory or data during thread creation or switching.

Thread switching does not require kernel to get involved, which in turn saves time on switching user to kernel mode.

Question 2:

One of the design decisions required for the implementation of threads is what do you do when a program makes a system call. Once must decide if only the thread upon which the system call occurs blocks until system call returns or does the entire process block.

In an environment where there is a 1-to-1 mapping between user-level and kernel-level threads, explain why blocking only the thread that made the system call (and allowing the other threads in the process to continue) will make mutli-threaded program run faster than the matching single-threaded program on a machine that only has 1 core.

The multi-threaded program would run faster because it doesn't remove all the rest of threads within the process. When ULT's execute a system call, not only is that thread blocked, but all of the threads within the process are blocked as well. If you skip over the system call(and return to it later), then the rest of the threads can run and come back to the system call after the remaining threads are processed and hope that the system call can go ahead and be processed at that time. There may be times that it wouldn't speed up the program, but I think more times than not, it would.

Question 4(Question 3 on next page):

Consider the function from the previous assignment. What aspect of the problem definition means that we can safely do the array manipulations without worrying about thread safety and synchronization?

Since addition is commutative we can execute threads in any order.

Question 3:

Your Glorious Instructor (think the educational equivalent of a third-world dictator, with less hair) has an interesting problem for you to solve. You have two very large arrays in C+

+ with up to 10,000,000 data samples in each array. Your Glorious Instructor needs the average of the data stored in those arrays.

Write a function in C++ that returns void and accepts four parameters: a dataset, a dataset, a result set, and size:

void arrayAverage(int *dataset1, int* dataset2, int *resultset, int size)

After executing, the resultset will contain the result of averaging each of the elements of the two datasets. Use threads to speed up this process...

...assume you have an 8-core machine. Partition the arrays based on the number of cores and assign different parts of two arrays to each thread. Use the thread join function to wait until all work is done.

```
#include <iostream>
#include <iomanip>
#include <string>
#include <thread>
#include <algorithm>
#include <chrono>

using namespace std;
using namespace std::chrono;

void fillWithRandoms(int *dataset1, int *dataset2, int size);
void averageArrays(int *dataset1, int *dataset2, int *resultset, int
size);
void arrayAverage(int *dataset1, int *dataset2, int *resultset, int
size);
void singleThreadAverage(int *dataset1, int *dataset2, int *resultset,
int size);
void buildRange(int *ranges);

const int SIZE = 10000000;

int main(){
  int *ary = new int[SIZE];
  int *bry = new int[SIZE];
  int *results = new int[2];

  fillWithRandoms(ary, bry, SIZE);
  arrayAverage(ary, bry, results, SIZE);

  cout << results[0] / SIZE << endl << results[1]/SIZE << endl;

  singleThreadAverage(ary, bry, results, SIZE);



  return 0;
}

void fillWithRandoms(int *dataset1, int *dataset2, int size){
  int i = 0;
  while (i < size){
    *dataset1 = rand() % 31;
    *dataset2 = rand() % 65;
    ++dataset1;
    ++dataset2;
    ++i;
  }
}

void averageArrays(int *dataset1, int *dataset2, int *resultset, int
size){
  int base = 0;
  if (size == (SIZE * (1.0 / 8.0)))
  {
    base = size - (SIZE * (1.0 / 8.0));
  }
  else {
    base = size - (SIZE * (1.0 / 8.0)) + 1;
  }
  while (base < size || base != SIZE){
     resultset[0] += dataset1[base];
     resultset[1] += dataset2[base];
     ++base;
  }
}

void arrayAverage(int *dataset1, int *dataset2, int *resultset, int
size){
  auto start = high_resolution_clock::now();  //for determining length
of time taken to run
  int i = (size * (1.0 / 8.0));
  int holder = i;
  const int num_threads = 8;
  thread t[num_threads];
  for (int i = 0; i < num_threads; ++i)
  {
    t[i] = thread(averageArrays, dataset1, dataset2, resultset,
holder);
    holder += i;
  }

  for (int i = 0; i < num_threads; ++i){
    t[i].join();
  }

  auto stop = high_resolution_clock::now(); //for determining length of
time taken to run
  auto duration = duration_cast<microseconds>(stop - start);
  cout << "Time to run threads: " << duration.count() << "
microseconds" << endl;
}

void singleThreadAverage(int *dataset1, int *dataset2, int *resultset,
int size){
  auto start = high_resolution_clock::now();

  thread th1 = thread(averageArrays, dataset1, dataset2, resultset,
size);
  th1.join();

  auto stop = high_resolution_clock::now(); //for determining length of
time taken to run
  auto duration = duration_cast<microseconds>(stop - start);
  cout << "Time to run single threaded: " << duration.count() << "
microseconds" << endl;
}

/*
Time to run threads: 524123 microseconds
15
60
Time to run single threaded: 3714 microseconds
*/
```