Lecture 10 — Symmetric Multiprocessing

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 3, 2018

Multiprocessing

Not that long ago, a typical computer had one processor with one core.

It could accordingly do exactly one thing at a time.

1 processor: 1 general purpose processor that executes user processes.

There may be special-purpose processors in the system (RAID controller).

Only one general purpose processor so we call it a uniprocessor system.

Multiprocessing

Now, desktops, laptops, and even cell phones are using multi-core processors.

A quad-core processor may be executing four different instructions from four different threads at the same time.

In theory, multiple processors may mean that we can get more work done in the same amount of (wall clock) time, but this is not a guarantee.

Plus, if one of the CPUs in a two-CPU system fails, the system can likely carry on at a reduced performance level.

(A)symmetric Multiprocessing

Multiprocessor systems can be asymmetric or symmetric.

Asymmetric: there is a boss processor and workers.

Symmetric: the workers are in charge and all processors are equal, comrade!

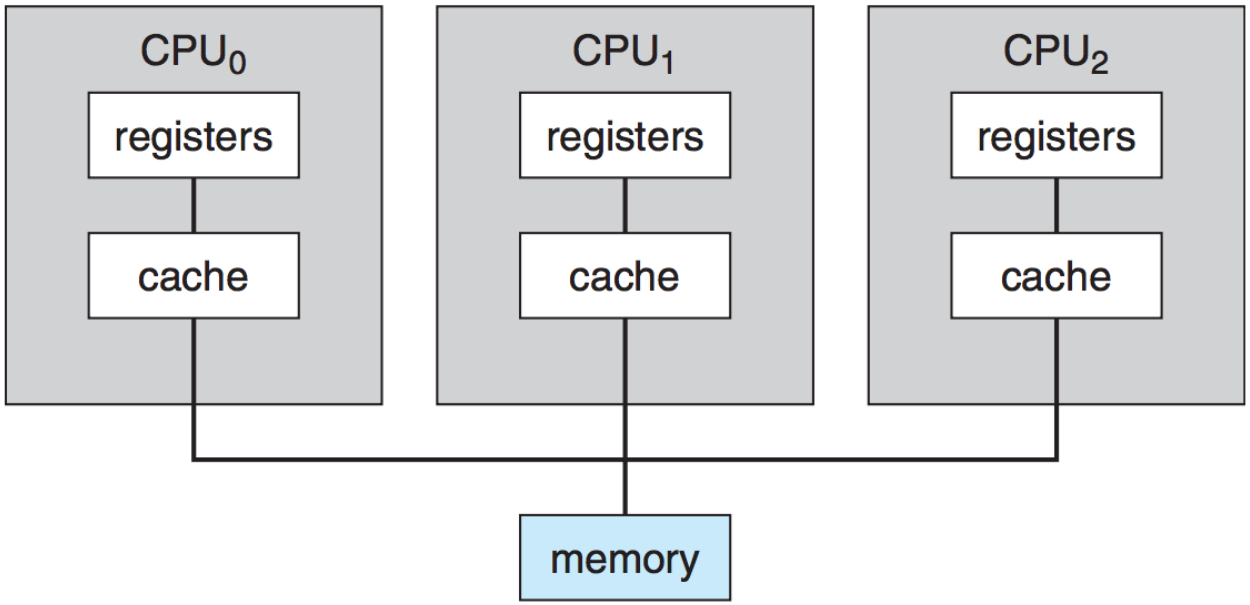
We are mostly familiar with symmetric systems.

Symmetric Multiprocessing

A formal definition of SMP:

1. There are two or more general purpose processors.
2. The processors share the same main memory and I/O devices and are interconnected by a bus.
3. All processors are capable of performing the same functions.
4. The system is controlled by an OS that provides interaction between processors and their programs.

SMP Organization



Terminology Note

Terminology note: we often refer to a logical processing unit as a core.

CPU may refer to a physical chip that contains 1+ logical processing units.

As far as the operating system is concerned, it does not much matter if a system has four cores in four physical chips or four cores in one chip.

Either way, there are four units that can execute instructions.

Execution

1 process, 1 thread: it does not matter how many cores are available.
At most one core will be used to execute this task.

If there are multiple processes, each process can execute on a different core.

But what if there are more processes and threads than available cores?

We can hope that the processes get blocked frequently enough and long enough?

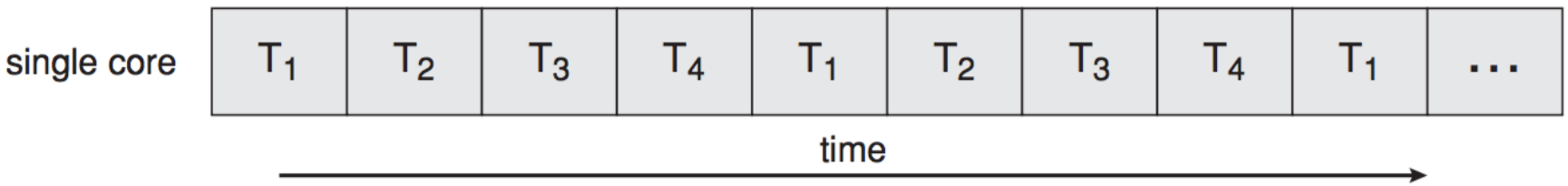
Execution

Switch between the different tasks via a procedure we call time slicing.

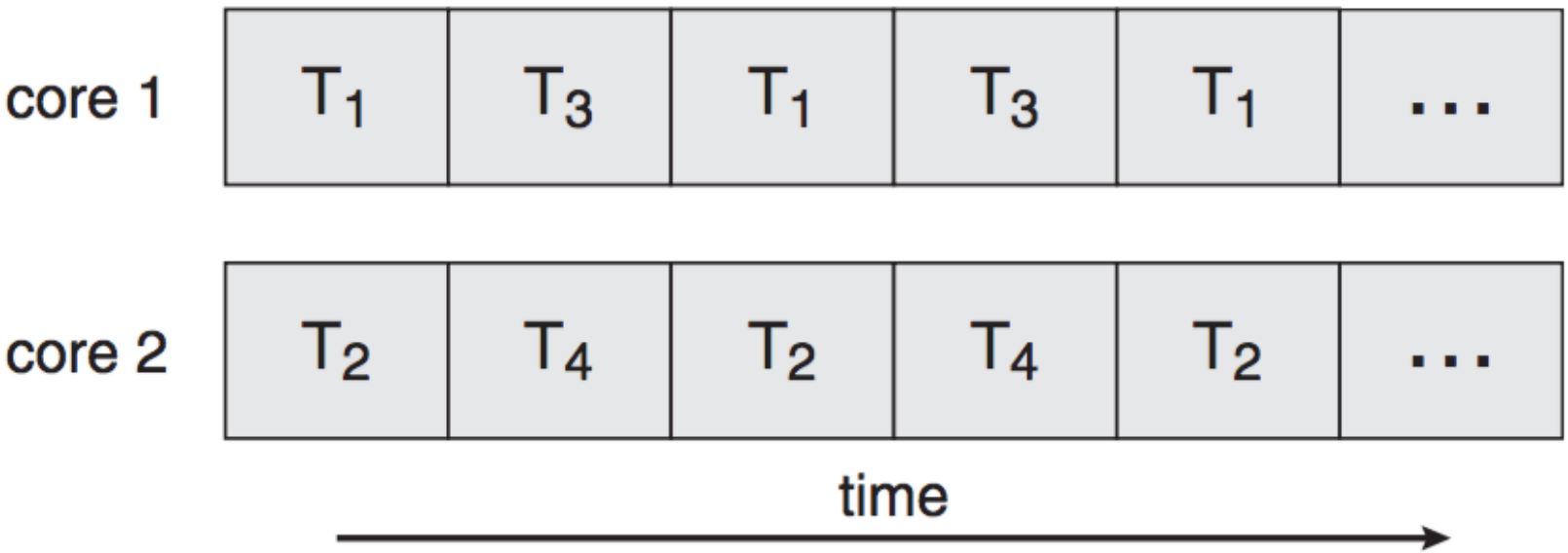
So thread 1 would execute for a designated period, such as 20 ms, then thread 2 for 20 ms, then thread 3 for 20 ms, then back to thread 1 for 20 ms.

To the user, it seems like threads 1, 2, and 3 are being executed in parallel.
20 ms is fast enough that the user does not notice the difference.

Single Core Execution



Time slicing will still occur, if necessary:



Parallelism

Multiple threads at the same time = tasks completed faster?

Parallelism and Speedup

Depends on the nature of the task!

Fully parallelized: $2 \times$ Threads $= 2 \times$ Speed

Partially parallelized: $2 \times$ Threads $= (1 < n < 2) \times$ Speed

Cannot be parallelized: $2 \times$ Threads $= 1 \times$ Speed

Speedup Example

Suppose: a task that can be executed in 5 s, containing a parallelizable loop.

Initialization and recombination code in this routine requires 400 ms.

So with one processor executing, it would take about 4.6 s to execute the loop.

Split it up and execute on two processors: about 2.3 s to execute the loop.

Add to that the setup and cleanup time of 0.4 s and we get a total time of 2.7 s.

Completing the task in 2.7 s rather than 5 s represents a speedup of about 46%.

Amdahl's Law

Gene Amdahl came up with a formula for the general case of how much faster a task can be completed based on how many processors we have available.

Let us define S as the portion of the application that must be performed serially and N as the number of processing cores available.

Amdahl's Law:

$$\text{speedup} \leq \frac{1}{S + \frac{1-S}{N}}$$

Amdahl's Law

Take the limit as $N \rightarrow$ infinity and you will find the speedup converges to $\frac{1}{S}$.

The limiting factor on how much additional processors help is the size of S.

Matches our intuition of how it should work.

Amdahl's Law on the 5 s Task

Applying this formula to the example from earlier:

Processors	Run Time (s)
1	5
2	2.7
4	1.55
8	0.975
16	0.6875
32	0.54375
64	0.471875
128	0.4359375

Observations on the 5 s Task

1. Diminishing returns as we add more processors.

2. Converges on 0.4 s.

The most we could speed up this code is by a factor of $\frac{5}{0.4} \approx 12.5$.

But that would require infinite processors (and therefore infinite money).

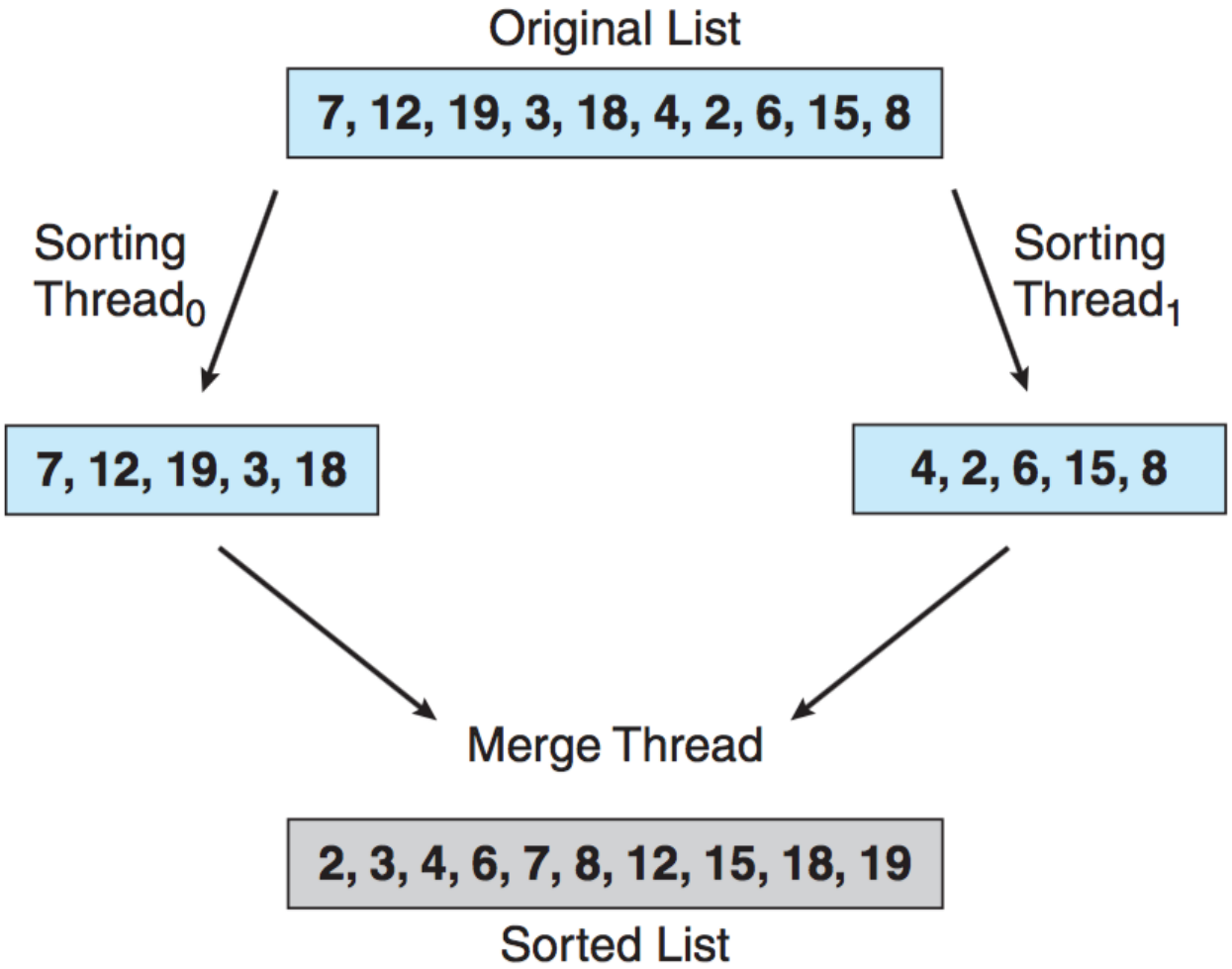
Merge Sort Example

Recall from data structures and algorithms the concept of merge sort.

This is a divide-and-conquer algorithm like binary search.

Split the array of values up into smaller pieces, sort those, and then merge the smaller pieces together to have sorted data.

Merge Sort Example



Merge Sort

For a small array it makes sense to use one of the simple algorithms like insertion sort.

(despite $\Theta(n^2)$ worst-case runtime).

You may have seen something similar in optimized binary search algorithms where, at some point, a linear search is used.

So let us begin with the code for the insertion sort routine:

Insertion Sort Code

```
void insertion_sort( double *array, size_t a, size_t b ) {
  size_t j, k;
  double tmp;
  for ( k = a + 1; k < b; ++k ) {
    tmp = array[k];

    for ( j = k; j > a; --j ) {
      if ( array[j - 1] > tmp ) {
        array[j] = array[j - 1];
      } else {
        array[j] = tmp;
        break;
      }
    }
    if ( tmp < array[a] ) {
      array[a] = tmp;
    }
  }
}
```

The Merge Sort

If the list is below a certain size, perform the insertion sort.

Otherwise, divide the list in half and call merge sort recursively on each half.

Merge the results of those two lists.

Merge Code

```
#include <assert.h>

// Merge the entries from a to b - 1 and from b to c - 1
void merge( double *array, size_t a, size_t b, size_t c ) {
  assert( a <= b && b <= c );

  size_t i = 0, j = a, k = b;
  double *sorted_array = (double *) malloc( (c - a)*sizeof( double ) );

  while ( j < b && k < c ) {
    if ( array[j] <= array[k] ) {
      sorted_array[i] = array[j];
      ++j;
    } else {
      sorted_array[i] = array[k];
      ++k;
    }

    ++i;
    }

  for ( ; j < b; ++i, ++j ) {
    sorted_array[i] = array[j];
  }

  for ( ; k < c; ++i, ++k ) {
    sorted_array[i] = array[k];
  }

  for ( i = 0, k = a; k < c; ++i, ++k ) {
    array[k] = sorted_array[i];
  }

  free( sorted_array );
}
```

Insertion Sort Code

```
#define USE_INSERTION_SORT 30

void merge_sort( double *array, size_t a, size_t c ) {
  assert( a <= c );

  if ( c - a < USE_INSERTION_SORT ) {
    insertion_sort( array, a, c );
    return;
  }
  size_t b = a + (c - a)/2;

  merge_sort( array, a, b );
  merge_sort( array, b, c );
  merge( array, a, b, c );
}
```

Argument Structure

To run this as a new thread, we can pass in only one argument for the parameters to the sort function.

So we need a `struct` object to contain the arguments:

```
typedef struct interval {
  double *array;
  size_t a;
  size_t c;
} interval_t;
```

User API

The user does not know that we are going to use a parallel algorithm so the function call should seem somewhat more “natural” and delegate to our internal function:

```
void merge_sort( double *array, size_t n ) {
  interval_t arg;
  arg.array = array;
  arg.a = 0;
  arg.c = n;
  merge_sort_internal( &arg );
}
```

Internal Implementation

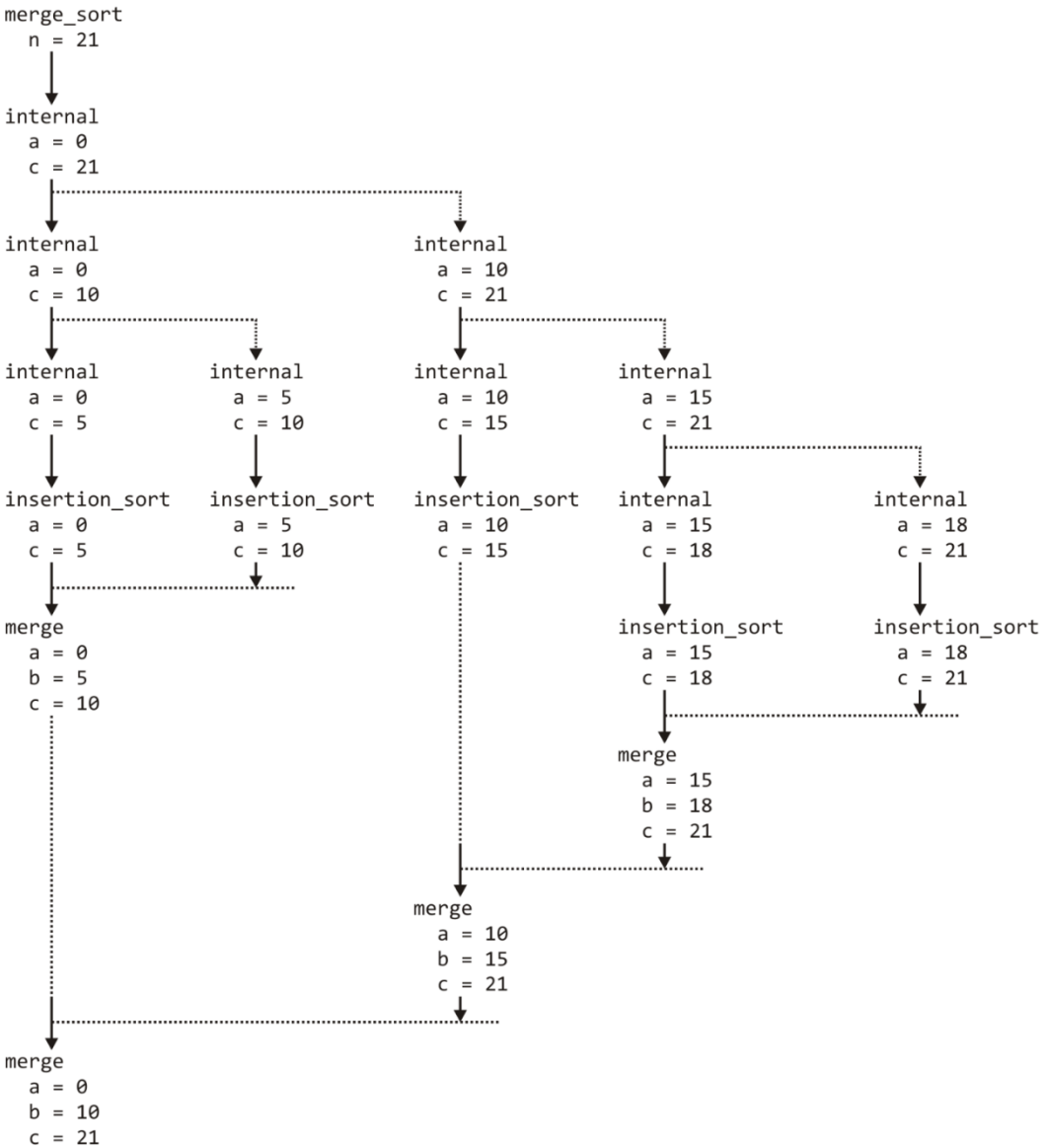
```
void *merge_sort_interal( void *void_arg ) {
  // the argument is an arbitrary pointer
  //   cast it to a pointer to an instance of 'interval_t'
  interval_t *arg = (interval_t *)void_arg;
  if ( ( arg->c - arg->a ) < USE_INSERTION_SORT ) {
    insertion_sort( arg->array, arg->a, arg->c );
    return NULL;
  }

  size_t b = arg->a + (arg->c - arg->a)/2;

  interval_t arg1, arg2;

  arg1.array = arg->array;
  arg1.a = arg->a;
  arg1.c = b;

  arg2.array = arg->array;
  arg2.a = b;
  arg2.c = arg->c;
```

Internal Implementation

```
  pthread_t other_thread;

  // Create a thread to sort the second half
  pthread_create( &other_thread, NULL, merge_sort_interal, &arg2 );
  merge_sort_interal( &arg1 );

  // Wait for them to finish
  pthread_join( other_thread, NULL );

  merge( arg->array, arg->a, b, arg->c );

  return NULL;
}
```

Execution, Visually

Execution on an array of capacity 21 and insertion sort threshold of 5:



Runtime Analysis

The runtime of merge sort, according to the data structures and algorithms course is $\Theta(n \ln(n))$;

If each of the threads execute on their own core, then the run time will be reduced to $\Theta(n)$.