Parallel Programming Exercises

Andreas Wilhelm

Technische Universität München

April 20, 2015

Organization

- Lecture could start at:
  - 4:00 PM
  - 4:15 PM
  - 4:30 PM
- Poll is available at http://doodle.com/ai2kp6872uutyait
- Duration: as long as we need, up to 90 min

Organization

- Assignments on parallel programming techniques
- Topics
  - Pthreads (Posix Threads)
  - OpenMP (Open Multi-Processing)
  - Dependency analysis
  - MPI (Message Passing Interface)
- Code examples are in C99 (no C++)
- My email address is: wilhelma@in.tum.de

Assignments

- Submission of 80% of the assignments gives 0.3 bonus
- Submissions will be checked for:
  - plagiarism
  - correctness (output, threads, synchronization)
  - speedup
  - memory leaks
- Submission is done on website: https://131.159.74.59/Submission
  - requires your LRZ ID and your password
  - password is not stored and only used for authentication
  - download SSL-certificate here
- Assignment instructions are on the last slides
- Final exam will contain small programming tasks
- Solutions will be made public

Assistance on Assignments

This week

- Given by: Andreas Wilhelm
- Room: 01.04.060
- Date and Time: Thursday 1:00 PM - 2:30 PM

After this week

- Given by: Khalid Alkhalili
- Email: kfkhalili@hotmail.com
- Room: 01.04.011
- Possible Date and Time:
  - Tuesday 1 PM, 2 PM, 3 PM (90 mins.)
  - Wednesday 1 PM, 2 PM, 3 PM (90 mins.)
  - Thursday 1 PM, 2 PM, 3 PM (90 mins.)
- Poll is available at http://doodle.com/8mf869znf5zd8zqc

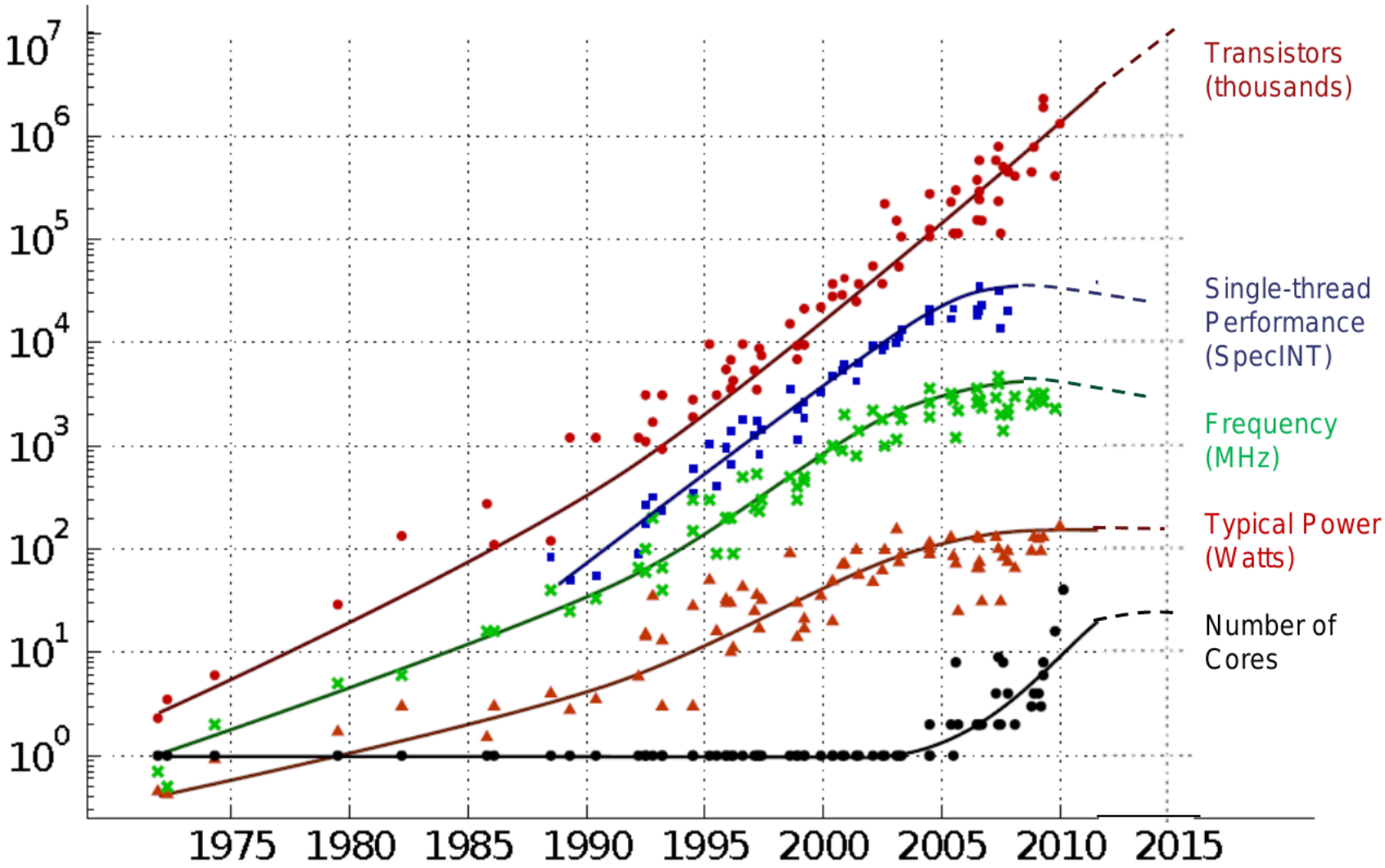
Resources

- POSIX Threads Programming
- An Introduction to Parallel Programming, by Peter Pacheco
- Programming with Posix Threads, by David Butenhof
- The Linux Programming Interface, by Michael Kerrisk
- Patterns for Parallel Programming, by Timothy G. Mattson; Beverly A. Sanders; Berna L. Massingill

Course Prerequisites

- Knowledge of C
  - memory management
  - pointers
  - global vs. static variables
- C books
  - (C89) The C Programming Language, Second Edition, by Brian W. Kernighan; Dennis M. Ritchie
  - (C99) C Primer Plus, Fifth Edition, by Stephen Prata
- Experience with Linux Command Line
- Resources
  - Book: The Linux Command Line
  - Basic video introduction: The Shell
- Knowing GCC
  - An Introduction to GCC, by Brian Gough

50th Anniversary of Moore's Law



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Year 2005: The Free Lunch Is Over

- A Fundamental Turn Toward Concurrency in Software
- Software doesn't get (much) faster with the next microprocessor generation
- Software developers have to rewrite their applications to use multiple processors in order top speed them up
- Parallel Programming is hard
  - to write - complex APIs and needs more code than serial version
  - to do it correctly - it's easy to introduce new bugs
  - to debug, order of thread execution is undefined
  - to make it scalable - will your applications scale with more cores?
  - better qualified developers are necessary

Posix Thread Programming

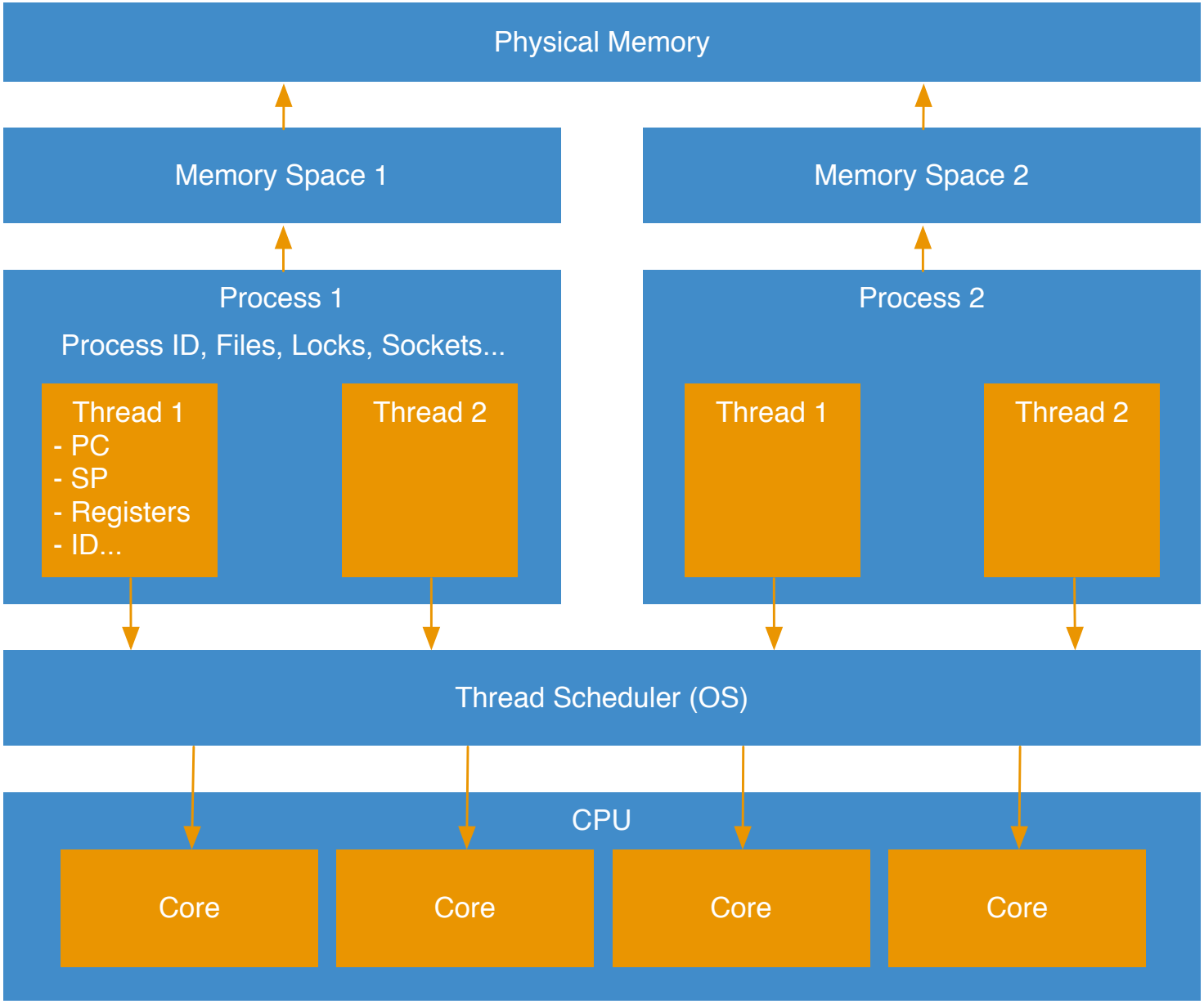
Definition: Thread

A thread is an independent stream of instructions that can be scheduled to run as such by the operating system.

POSIX Threads (Pthreads)

- Were defined in 1995 (IEEE Std 1003.1c-1995)
- Is an API that defines a set of types, functions and constants
- Is implemented with a `pthread.h` header and a thread library
- Functions can be categorized in four groups:
  - Thread management
  - Mutexes
  - Condition variables
  - Read/write locks and barriers

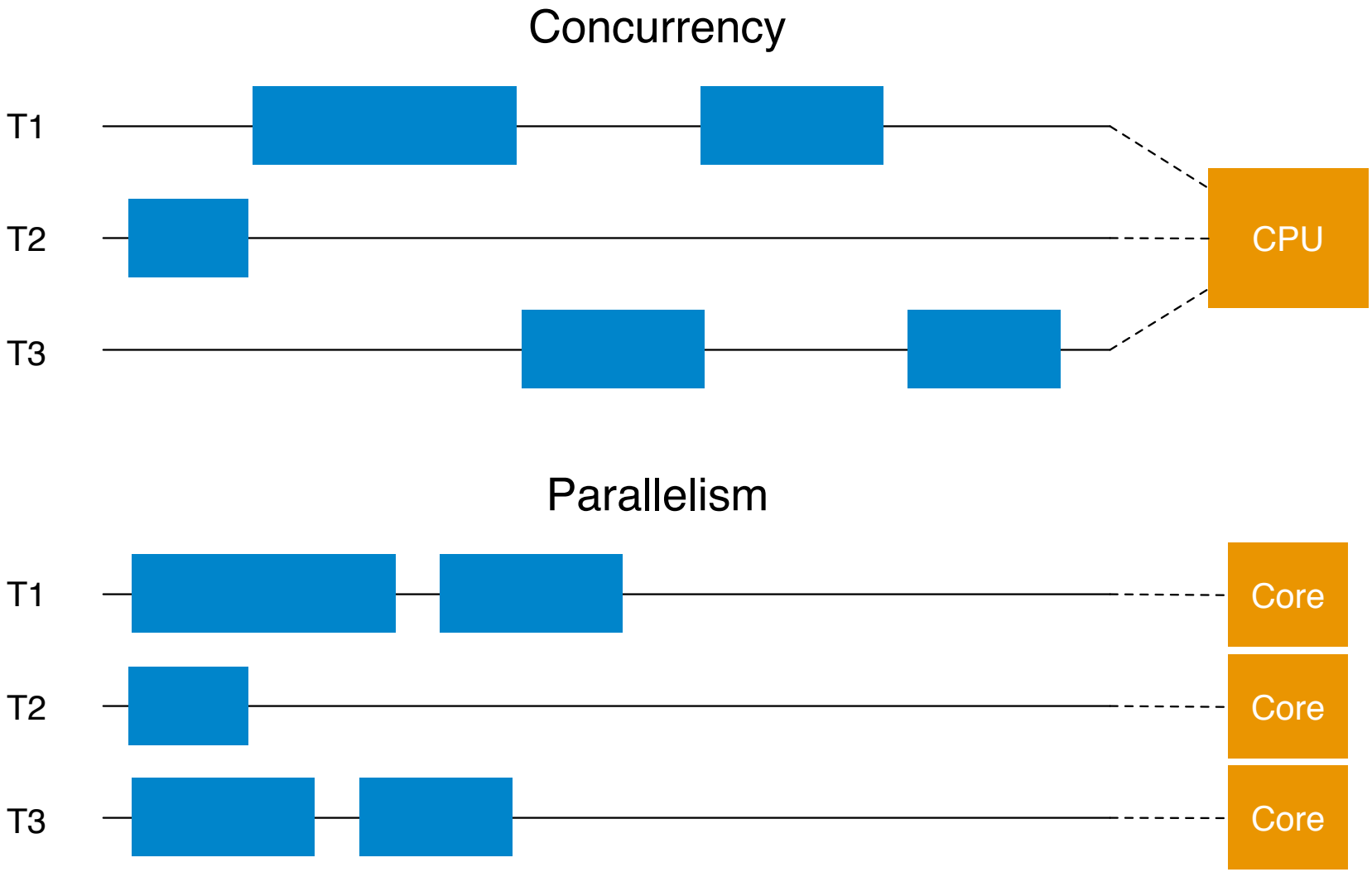
Processes vs. Threads



Why use Multithreading?

- **Performance gains**
  Parallel processing by multiple processor cores
- **Increased application throughput**
  Asynchronous system calls possible
- **Increased application responsiveness**
  Application does not need to block operations
- **Replacing process-to-process communications**
  Threads may communicate by shared-memory
- **Efficient use of system resources**
  Lightweight context switches possible
- **The ability to create well-structured programs**
  Some programs are inherently concurrent

Concurrency vs. Parallelism



Why are threads ‘faster‘ than processes?

- Creating a new process with fork() has a big overhead: whole memory must be copied
  - Waste of memory space!
- Synchronization with processes usually involves system calls.

Create Pthreads

```
1  int pthread_create(pthread_t *thread,
2                     const pthread_attr_t *attr,
3                     void *(*start_routine) (void *),
4                     void *arg);
```

- pthread_t *thread,
  - Pointer to thread identifier.
- const pthread_attr_t *attr
  - Optional pointer to pthread_attr_t to define behavior, if NULL defaults are used.
- void *(*start_routine) (void *),
  - Pointer to function prototype that is started. Function takes void pointer as argument and returns a void pointer.
- void *arg
  - Pointer to the argument that is used for the executed function.

Create Pthreads - Example

```
1  void * hello()
2  {
3    printf("Hello World from pthread!\n");
4    return NULL;
5  }
6
7  void main()
8  {
9    pthread_t thread;
10
11   pthread_create(&thread, NULL, &hello, NULL);
12
13   printf("Hello World from main!\n");
14
15   pthread_join(thread, NULL);
16 }
```

Waiting for Pthread to finish

```
1  int pthread_join(pthread_t thread,
2                   void **retval);
```

- pthread_t thread,
  - Pointer to thread identifier, for which this function is waiting.
- void **retval
  - Optional pointer pointing to a void pointer. This can be used to return data of undefined size.

Compile & Output

```
gcc --std=gnu99 -pthread -Wall
    -o hello_world hello_world.c

Hello World from main!
Hello World from pthread!
```

More than One: Hello World with Pthreads Ver. 1

```
1  void main()
2  {
3    long num_threads = 3; pthread_t *thread;
4
5    thread = malloc((num_threads * sizeof(*thread));
6
7    for (int i = 0; i < num_threads; i++)
8      pthread_create(thread + i, NULL, &hello, NULL );
9
10   for (int i = 0; i < num_threads; i++)
11     pthread_join(thread[i], NULL );
12 }
```

Output

```
[user]$ ./hello_world_pthread_1
Hello World from pthread!
Hello World from pthread!
Hello World from pthread!
```

Single Argument: Hello World with Pthreads Ver. 2

```
1  void * hello(void *ptr)
2  {
3    printf("Hello World from pthread %d!\n", *((int*)ptr));
4    return NULL ;
5  }
```

Single Argument: Hello World with Pthreads Ver. 2

```
1  void main()
2  {
3    int num_threads = 3;  pthread_t *thread;  int *thread_arg;
4
5    thread = malloc(num_threads * sizeof(*thread));
6    thread_arg = malloc(num_threads * sizeof(*thread_arg));
7
8    for (int i = 0; i < num_threads; i++)
9    {
10     thread_arg[i] = i;
11     pthread_create(thread + i, NULL, &hello, thread_arg + i);
12   }
13
14   for (int i = 0; i < num_threads; i++)
15       pthread_join(thread[i], NULL );
16 }
```

Output

```
[user]$ ./hello_world_pthread_2
Hello World from pthread 0!
Hello World from pthread 1!
Hello World from pthread 2!
```

Many Arguments: Hello World with Pthreads Ver. 3

```
1  struct pthread_args
2  {
3    long thread_id;
4    long num_threads;
5  };
6
7  void * hello(void *ptr) {
8    struct pthread_args *arg = ptr;
9    printf(
10     "Hello World from pthread %ld of %ld PID = %d TID = %d!\n"
11     arg->thread_id,
12     arg->num_threads,
13     getpid(),
14     (unsigned int)pthread_self());
15
16   return NULL ;
17 }
```

Many Arguments: Hello World with Pthreads Ver. 3

```
1  void main()
2  {
3    long num_threads = 3;
4    pthread_t *thread;
5    struct pthread_args *thread_arg;
6
7    thread = malloc(num_threads * sizeof(*thread));
8    thread_arg = malloc(num_threads * sizeof(*thread_arg));
9
10   for (int i = 0; i < num_threads; i++)
11   {
12     thread_arg[i].thread_id = i;
13     thread_arg[i].num_threads = num_threads;
14     pthread_create(thread + i, NULL, &hello_pthread, thread_arg
15   }
16
17   for (int i = 0; i < num_threads; i++)
18     pthread_join(thread[i], NULL );
19 }
```

Output

```
[user]$ ./hello_world_pthread_3
Hello World from pthread 1 of 3 PID = 23750 TID = 23752!
Hello World from pthread 0 of 3 PID = 23750 TID = 23751!
Hello World from pthread 2 of 3 PID = 23750 TID = 23753!
```

Return Result from Pthread in struct Argument

```
1  struct pthread_args
2  {
3    int in, out;
4  };
5
6  void * triple(void *ptr)
7  {
8    struct pthread_args *arg = ptr;
9
10   arg->out = 3 * arg->in;
11
12   return NULL ;
13 }
```

Return Result from Pthread in struct Argument

```
1  void main() {
2    long num_threads = 3; pthread_t *thread;
3    struct pthread_args *thread_arg;
4
5    thread = malloc(num_threads * sizeof(*thread));
6    thread_arg = malloc(num_threads * sizeof(*thread_arg));
7
8    for (int i = 0; i < num_threads; i++){
9      thread_arg[i].in = i;
10     pthread_create(thread + i, NULL, &triple, thread_arg + i);
11   }
12
13   for (int i = 0; i < num_threads; i++){
14     pthread_join(thread[i], NULL );
15     printf("Triple of %d is %d\n",
16             pthread_args[i].in,
17             pthread_args[i].out);
18   }
19 }
```

Return Result from Pthread as Pointer to Memory

```
1
2  void * triple(void *ptr) {
3
4    int *out = malloc(sizeof(*out));
5    *out = 3 * (*(int*)ptr);
6
7    return out;
8  }
```

Return Result from Pthread as Pointer to Memory

```
1  void main() {
2    long num_threads = 3; pthread_t *thread;
3    int *in;
4
5    thread = malloc(num_threads * sizeof(*thread));
6    in = malloc(num_threads * sizeof(*in));
7
8    for (int i = 0; i < num_threads; i++){
9      in[i] = i;
10     pthread_create(thread + i, NULL, &triple, in + i);
11   }
12
13   for (int i = 0; i < num_threads; i++){
14     int *out;
15     pthread_join(thread[i], &out );
16     printf("Triple of %d is %d\n", in[i], *out);
17     free(out);
18   }
19 }
```

What have we covered so far?

- Creating new threads with pthread_create
- Waiting for threads to finish with pthread_join
- Passing arguments to a pthread function
- Returning results from pthread function

Assignment: Histogram

Task

- A histogram represents the frequencies of different alphabetic characters in an input file (here: case insensitive).
- Use POSIX threads to parallelize a given program that computes such a histogram.

Usage of the program

- Sequential:
  `./histogram_seq <file> 1 (<#repetitions>)`
- Parallel:
  `./histogram_par <file> (<#threads>) (<#repet.>)`

Assignment: Example - ”War and Peace“ (Tolstoy)

```
Process war_and_peace.txt by 1 thread(s)

315232:          |
283708:          |
252185:          |
220662:          |                                 |
189139: |        |                       |         |
157616: |        |      | |            | |        | |
126092: |        |      | |            | |      | | |
 94569: |      | |      | |      |     | |      | | |
 63046: |      | |      | |      |     | |      | | | |
 31523: | | | | | | | | |      |     | | | | |    | | | |    |    |
     0: | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
        a b c d e f g h i j k l m n o p q r s t u v w x y z

Time: 0.01325 seconds
```

Assignment: Build Histogram (Sequential)

```
1  #include "histogram.h"
2
3  void get_histogram(unsigned int nBlocks,
4                     block_t *blocks,
5                     unsigned int* histogram,
6                     unsigned int num_threads) {
7
8     unsigned int i, j;
9     for (i=0; i<nBlocks; i++) {
10       for (j=0; j<BLOCKSIZE; j++) {
11          if (blocks[i][j] >= 'a' && blocks[i][j] <= 'z')
12             histogram[blocks[i][j]-'a']++;
13          else if(blocks[i][j] >= 'A' && blocks[i][j] <= 'Z')
14             histogram[blocks[i][j]-'A']++;
15       }
16    }
17 }
```

Assignment: Provided Files

- Makefile
  - contains rules to build executables
  - available targets: parallel, sequential, all (default), clean
  - 'mode=debug make [target]' to build debug version, use 'make clean' before
- main.c
- histogram.h
  - Header file for histogram_*.c
- histogram.c
  - Defines `print_histogram`
- histogram_seq.c
  - Sequential version. Replace it with pi series given in the slides.
- student/histogram_par.c
  - Implement the parallel version in this file
- war_and_peace.txt
  - Input data: War and peace from Tolstoy

Assignment: Extract, Build, and Run

1. Extract all files to the current directory

```
tar -xvf histogram.tar
```

2. Build the program

```
make [sequential] [parallel]
```

- sequential: build the sequential program
- parallel: build the parallel program

3. Run the sequential program (100 repetitions)

```
student/histogram_seq war_and_peace.txt 1 100
```

4. Run the parallel program (with N threads)

```
student/histogram_par war_and_peace.txt N 100
```

Submission

1. Log into the webiste
2. Go to Assigments
3. Upload your `histogram_par.c`
4. Press Submit