

Week 8: POSIX Threads

COMP 2129

COMMONWEALTH OF AUSTRALIA
Copyright Regulations 1969
WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Outline

- POSIX Threads
 - Introduction
 - Thread creation, termination
 - Passing arguments to threads
 - Thread synchronization upon termination (`pthread_join`)
 - Thread scheduling
- Examples

Introduction

- We need a way to tell the compiler that a sequence of instructions (=Task) shall be executed in parallel.
 - Examples: Tasks **T1** (min), **T2** (max).
- We need to be able to specify communication and synchronization between tasks.
 - Example communication: task **T1** needs to communicate the computed minimum value (*min*) back to task **T3** for output.
 - Example synchronization: task **T3** can only output the *min* and *max* results after they have been received from task **T1** and **T2**.
- POSIX threads allow us to do that
 - The sequence of instructions of a task becomes a POSIX thread.
 - POSIX threads can communicate and synchronize with each other.
 - The operating system schedules threads (i.e., lets them execute). Works both on uniprocessors and multicores.

Example:
parallel *min*, *max* computations
on integer array:

```
#define maxN 1000000000
int m[maxN];

int i; int min = m[0];
for(i=1; i < maxN; i++) {
    if(m[i] < min)
        min = m[i];
}

int k; int max = m[0];
for(k=1; k < maxN; k++) {
    if(m[k] > max)
        max = m[k];
}

printf("min %d max %d\n",
        min, max);
```

T1

T2

T3

The POSIX threads API

- **API = Application Programming Interface:** a set of functions, procedures or classes provided by an operating system or library for use by application programs.
- Many companies provide vendor-specific APIs to program threads.
- IEEE provides the 1003.1c-1995 POSIX API standard
 - also referred to as Pthreads (for “POSIX threads”).
 - POSIX: Portable Operating System Initiative (+X for “UniX”)
- Most operating systems support Pthreads
- Pthreads widely used today, e.g., with



Apache



MySQL

- Pthreads very similar to NT threads, Solaris threads, Java threads.
- Studying Pthreads will make it easy for you to work with other types of threads.

Hello World, revisited

```
#include <stdio.h>
#include <unistd.h> /* for the sleep() system call */

int main(void){

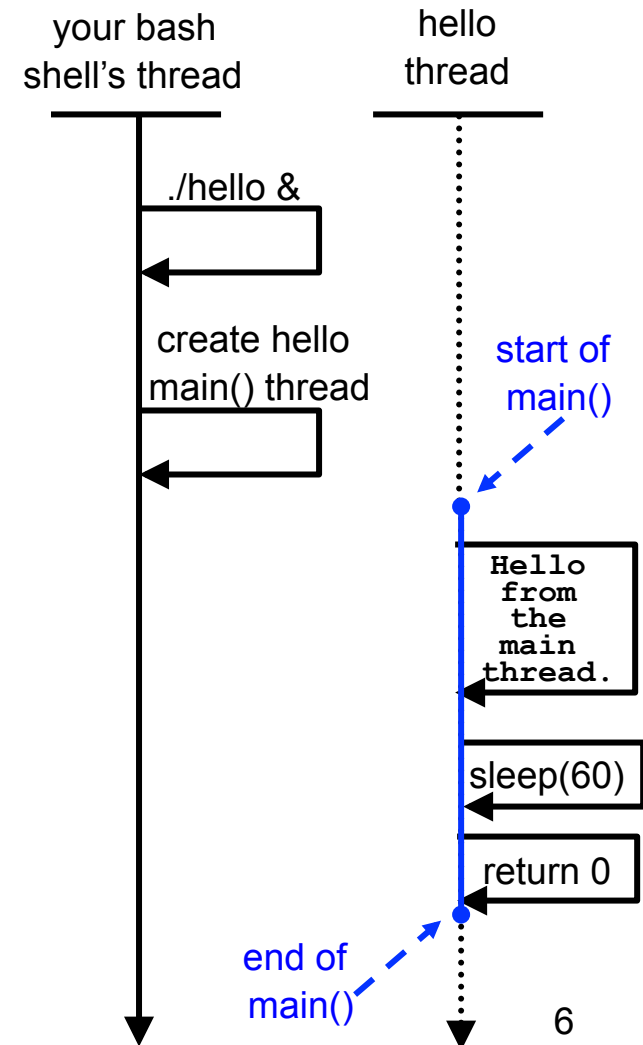
    printf("Hello from the main thread.\n");

    sleep (60); /* take a rest for 60 seconds... */

    return 0;
}
```

```
[bburg@elc1 ~]$ gcc -o hello hello.c
[bburg@elc1 ~]$ ./hello &
[bburg@elc1 ~]$ Hello from the main thread.
[bburg@elc1 ~]$ ps -eLf
```

The diagram to the right is a UML sequence diagram.
Time flows from the top to the bottom.



Hello World, revisited (cont.)

```
#include <stdio.h>
#include <unistd.h> /* for the sleep() system call */

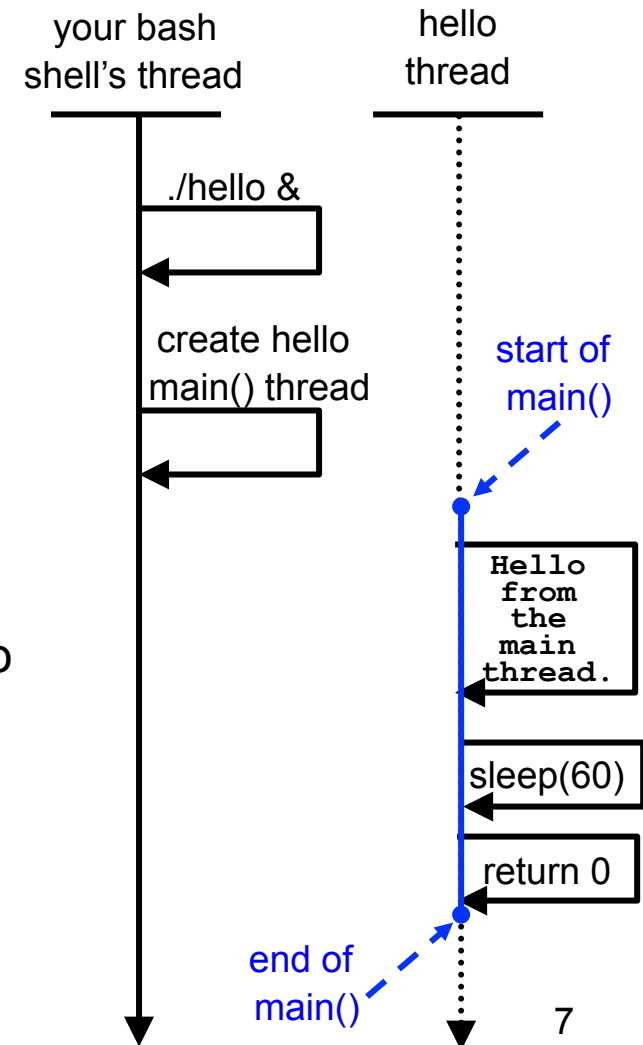
int main(void){

    printf("Hello from the main thread.\n");

    sleep (60); /* take a rest for 60 seconds... */

    return 0;
}
```

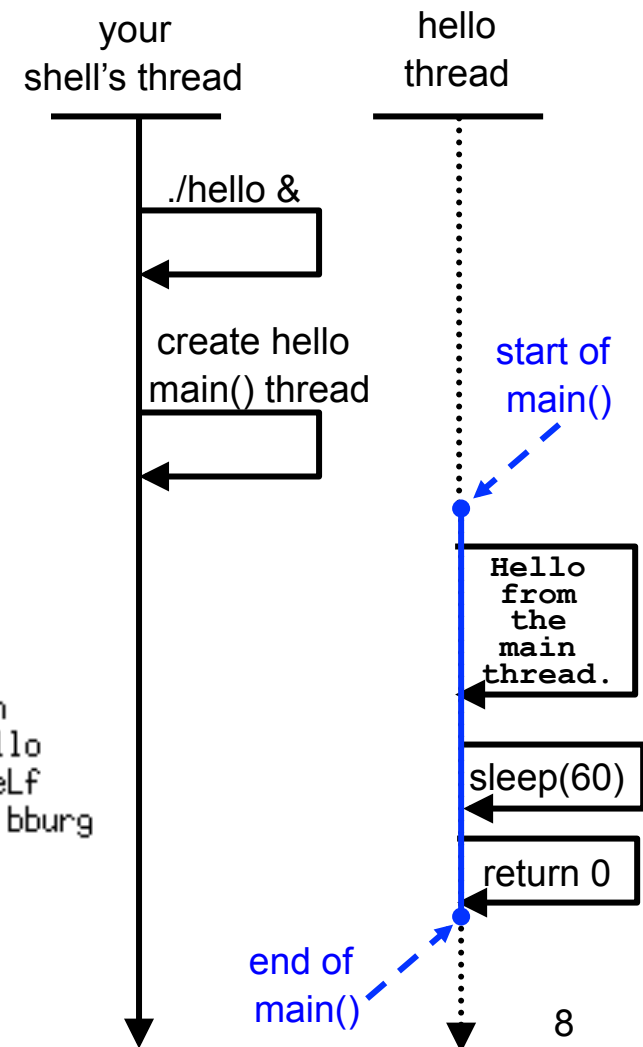
- The user starts the hello program from the shell.
- The shell creates a new process (thread) for the hello program.
- The main() function of the hello program starts executing (**this is the program's main() thread**).
 - outputs "Hello from the main thread."
 - sleeps for 60 seconds
 - terminates, returning 0 to the shell.



Hello World, revisited (cont.)

- Use `ps -eLf` to find out about all programs running on the system.
- Use `ps -eLf | grep <yourID>` to find out about all of *your* programs running on the system.
 - In the above command, “|” is a pipeline. A pipeline (“|”) takes the output of one command and makes it the input to another command.
 - Use `man grep` to find out about grep.
 - sample output:

```
bburg      5542  5541  5542  0    1 Sep13 pts/9    00:00:00 -bash
bburg      18654 11834 18654  0    1 08:57 pts/4    00:00:00 ./hello
bburg      18655 11901 18655  0    1 08:57 pts/6    00:00:00 ps -eLf
bburg      18656 11901 18656  0    1 08:57 pts/6    00:00:00 grep bburg
[bburg@elc ~]$
```



Our first POSIX Thread

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

pthread_t my_thread;

void* threadF (void *arg) {
    printf("Hello from our first POSIX thread.\n");
    sleep(60);
    return 0;
}

int main(void) {

    pthread_create(&my_thread, NULL, threadF, NULL);

    printf("Hello from the main thread.\n");

    pthread_join(my_thread, NULL);

    return 0;
}
```

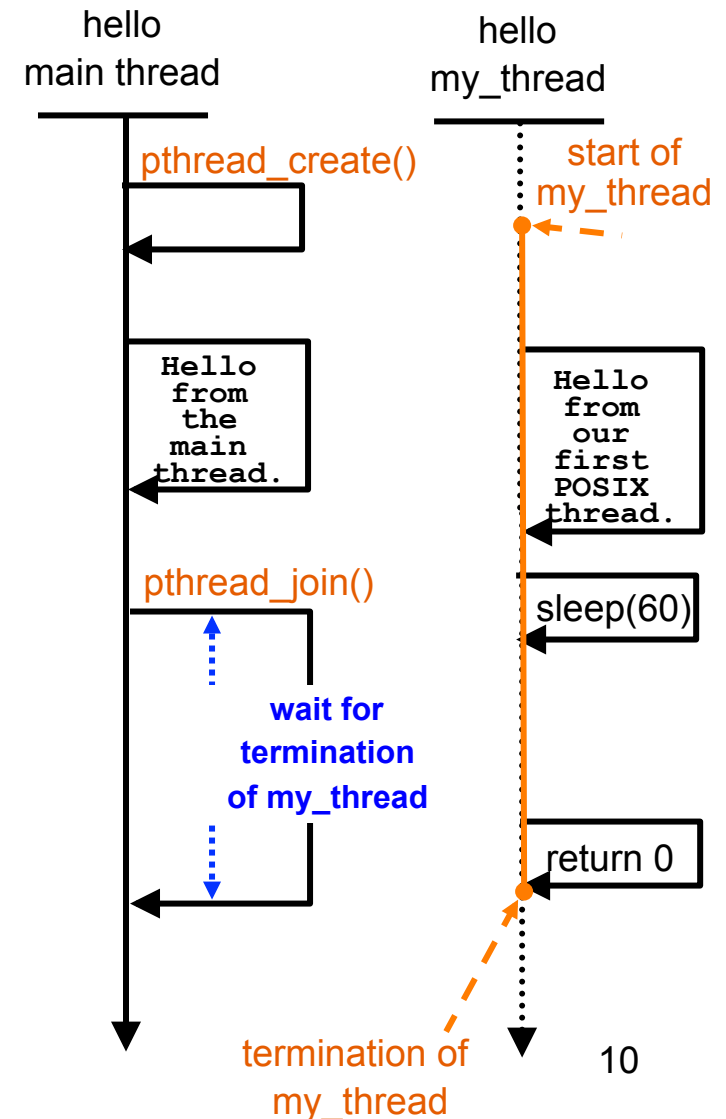
```
[bburg@elc1 ~]$ gcc -o hello hello.c -lpthread
```

- To compile this program, you need to specify **-lpthread**
 - Tells GCC to link in the POSIX thread library.
- `ps -eLf | grep <yourID>` shows 2 threads for hello:
 - the main thread
 - function main()
 - my_thread
 - function threadF()

bburg	11901	11900	11901	0	1	Sep15	pts/6	00:00:00	-bash
bburg	18702	11834	18702	0	2	09:08	pts/4	00:00:00	./hello
bburg	18702	11834	18703	0	2	09:08	pts/4	00:00:00	./hello
bburg	18704	11901	18704	0	1	09:08	pts/6	00:00:00	ps -eLf
bburg	18705	11901	18705	0	1	09:08	pts/6	00:00:00	grep bburg

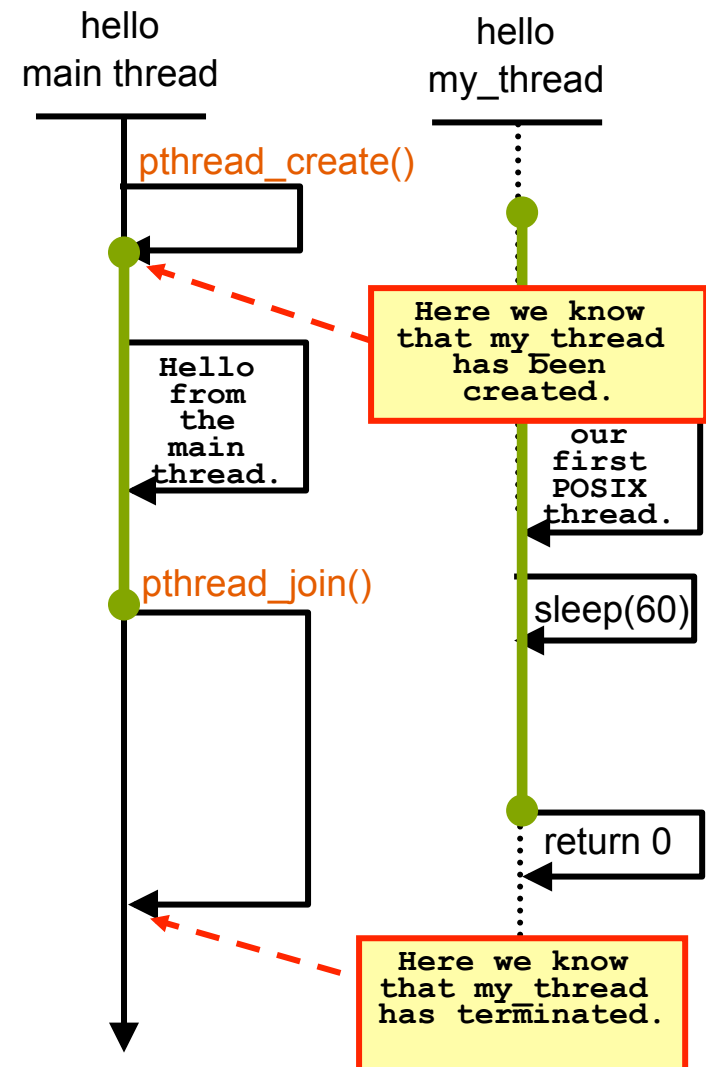
Our first POSIX Thread (cont.)

- The main thread of program hello calls `pthread_create()` to create the POSIX thread 'my_thread'.
- `threadF()` is the pointer to the function that this thread will execute.
- After its creation, the new thread starts executing its thread function (`threadF`).
- `my_thread` and the main thread execute **in parallel**.
- The main thread calls `pthread_join()` to wait for `my_thread` to terminate.
 - The call to `pthread_join()` returns after `my_thread` has terminated.
 - We say that the main thread is **blocked** (has to wait).



Execution Indeterminism

- For threads that execute in parallel, no assumption about the statement execution order *between threads* is possible!
 - Example: the statements between `pthread_create()` and `pthread_join()` of the main thread execute in parallel to the statements of `my_thread` (overlap depicted in **green**).
 - The output of the main thread might happen before the output of `my_thread`, or vice versa:
Hello from the main thread.
Hello from our first POSIX thread.
or
Hello from our first POSIX thread.
Hello from the main thread.
- We will discuss later how we can enforce a particular execution order of statements between threads.
 - This is called synchronization.



Thread-Safe Routines

- A function, library routine or system call is **thread-safe**, if it can be called from several threads simultaneously and produce correct results.
- `printf()` is **thread-safe**
 - **Even if called by 2 threads simultaneously, it is guaranteed that outputs won't be mixed.**
 - Each `printf()` output happens **atomically** (i.e., as one unit, not split into parts).
 - Example from the previous slide will not produce mixed-up output like this:

*Hello from thHello from our first POSe main thread.
IX thread.*

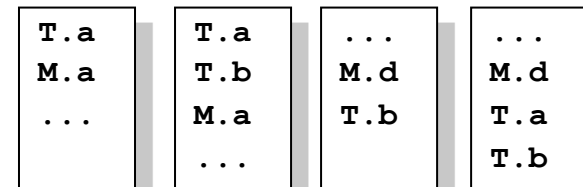
- `printf` will **serialize** the output:
 - main thread before `my_thread`, or
 - `my_thread` before main thread.

Statement Execution Order between Threads

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  pthread_t mythread;
5
6  void* threadF (void *arg) {
7      printf("T.a\n");
8      printf("T.b\n");
9  }
10
11 int main(void) {
12
13     printf("M.a\n");
14
15     pthread_create(&mythread, NULL, threadF, NULL);
16     printf("M.b\n");
17     printf("M.c\n");
18     pthread_join(mythread, NULL);
19
20     printf("M.d\n");
21
22     return 0;
23 }
```

pthread_join allows **synchronization** between threads: a thread can **wait** (block) until another thread has terminated. More general forms of synchronization will follow...

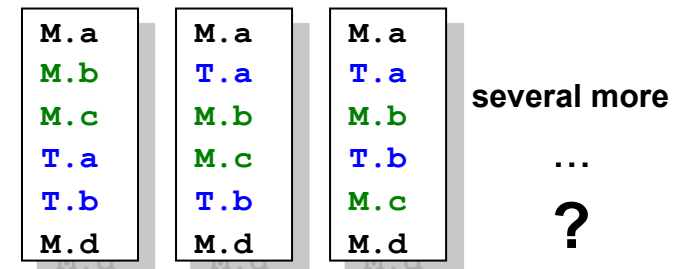
- No statement of thread mythread can execute before mythread's creation (call to pthread_create).
- No statement of thread mythread can execute after mythread's termination (after pthread_join() has returned).
- Impossible statement orderings (printf output):



Statement Execution Order between Threads

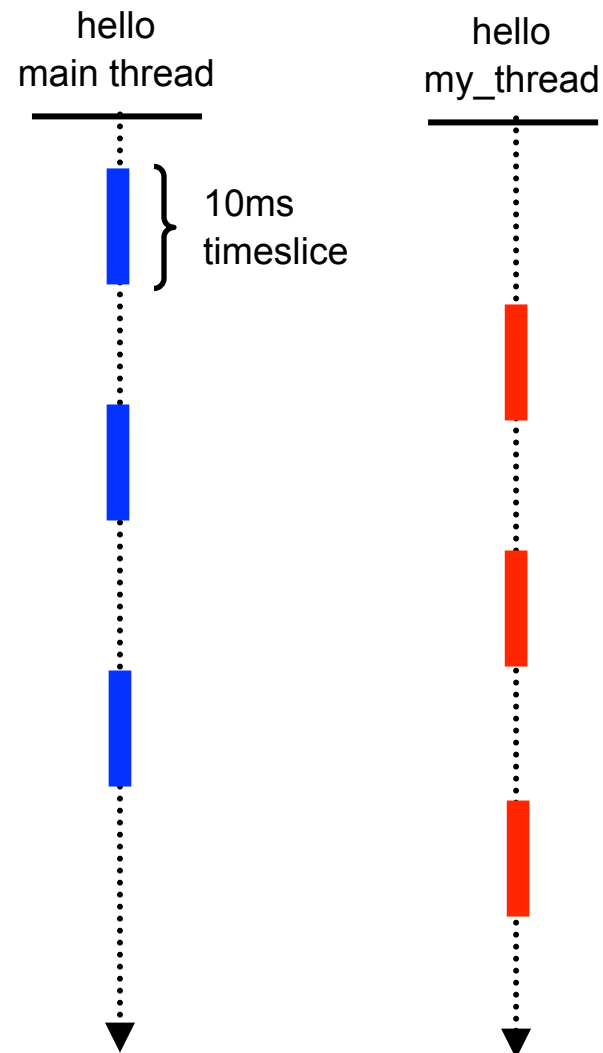
```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  pthread_t mythread;
5
6  void* threadF (void *arg) {
7      printf("T.a\n");
8      printf("T.b\n");
9  }
10
11 int main(void) {
12
13     printf("M.a\n");
14
15     pthread_create(&mythread, NULL, threadF, NULL);
16     printf("M.b\n");
17     printf("M.c\n");
18     pthread_join(mythread, NULL);
19
20     printf("M.d\n");
21
22     return 0;
23 }
```

- The statements of thread mythread execute in parallel to the statements between pthread_create() and pthread_join() of the main thread.
- Possible statement orderings (printf output):



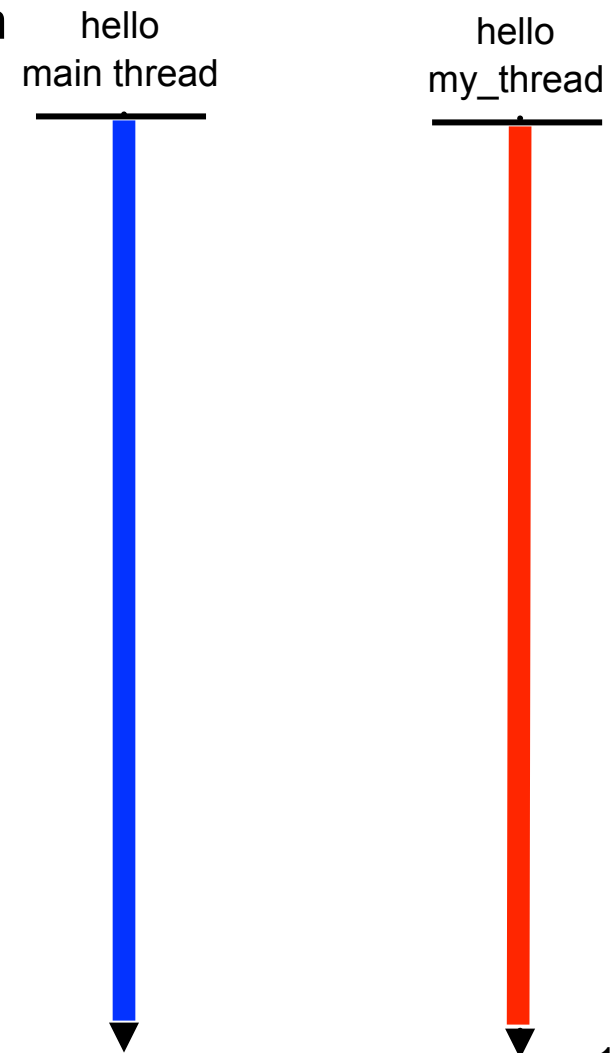
The Operating System Schedules Threads...

- On uniprocessor systems, all threads share a single processor.
- We can still use threads, because the operating system interleaves execution of threads.
Example: every thread executes for 10ms. The user has the illusion that threads execute in parallel.
- This is also called
 - multi-programming
 - quasi-parallelism
 - logically parallel



The Operating System Schedules Threads... (cont.)

- On multicores or multiprocessors, we can achieve “true” parallelism (several threads executing at the same time).
- This is also called
 - multi-processing
- If there are more threads than processors/cores, the operating system must still multiplex between threads. But at any time, more than one thread is physically executing.
- The programmer should not rely on particular scheduling algorithms.
 - Might break when the program is run on another hardware architecture or OS.
 - Assume true parallelism, don't rely on particular statement execution orders!
 - Trying out 10 times does not prove anything!



Shared address space between threads

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

int i = 0;

pthread_t my_thread;

void* threadF (void *arg) {
    printf("Hello, i is %d.\n", i);
}

int main(void) {
    i = 1;
    pthread_create(&my_thread, NULL, threadF, NULL);


    pthread_join(my_thread, NULL);

    return 0;
}
```

- Threads share the address space.
- Variable i is visible by both threads
 - main thread
 - my_thread
- Both a blessing and a curse
 - (will become apparent soon)

pthread_create()

```
int pthread_create(  
    pthread_t *tid,           // Purpose: create a new thread.  
    const pthread_attr_t *attr, // pointer to thread ID  
    void * (*start_routine) (void *), // pointer to thread attributes  
    void *arg,                // pointer to function that the thread will execute  
                                // pointer to argument  
);
```

 run-time argument passing

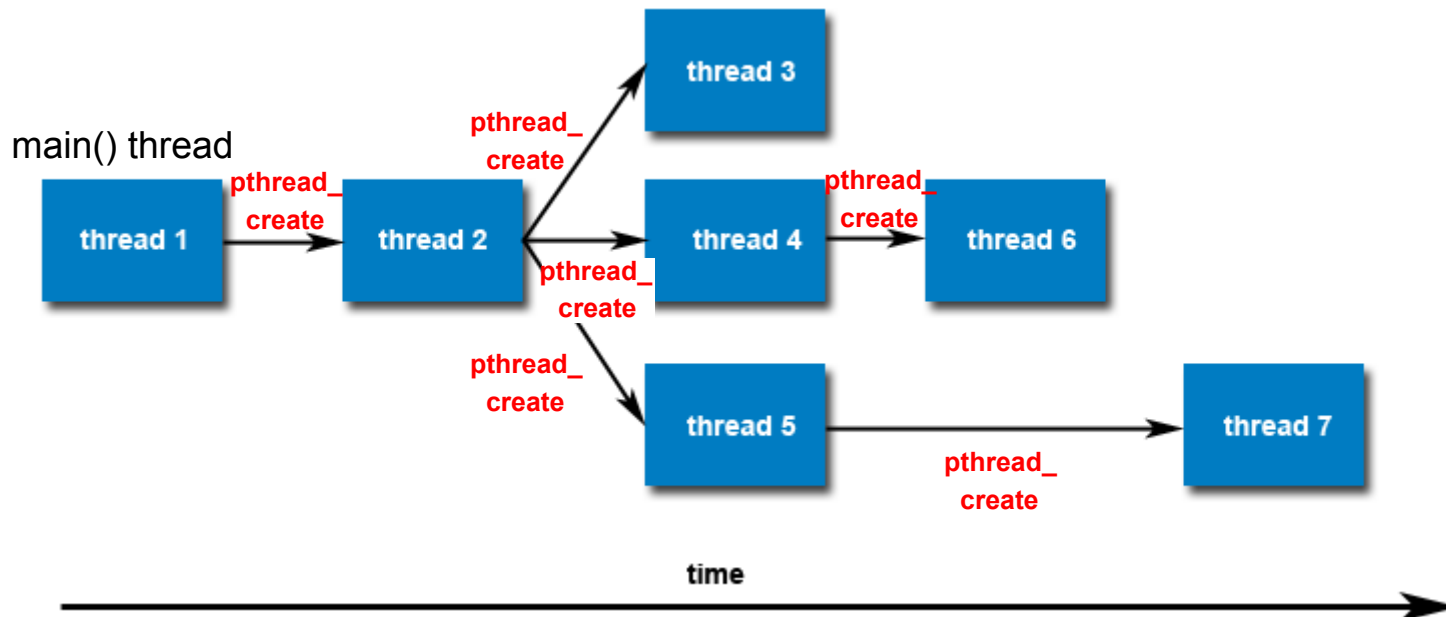
Arguments:

1. The thread ID variable that will be used for the created thread.
 - (We need a way to refer to the thread, e.g., when we want to join it).
2. The thread's attributes (explained on subsequent slides).
 - NULL specifies default attributes.
3. The function that the thread will execute once it is created (start_routine).
 - `void * (*start_routine) (void *)` is a function pointer to a function that
 - has `void *` as its return value
 - accepts one `void *` argument
4. The argument that will be passed to `start_routine()` at **run-time**.

pthread_create()

Once created, threads are peers

- may create other threads
- no implied hierarchy or dependency between threads
- No assumption on execution order of siblings possible.
 - Example: assume thread2 creates thread3 and then thread4.
 - You must not assume that thread3 will start execution before thread 4.
 - Except when applying Pthread scheduling mechanisms



Passing Arguments to a Thread Function

- `pthread_create()` allows us to pass *one* argument to a thread.

```
1  #include<stdio.h>
2  #include<pthread.h>

3  #define NUMT ...
4  pthread_t threadIDs[NUMT]; /* thread IDs */

5  void * tfunc (void * p) {
6      // thread work function
7  }

8  int main(void) {
9      int i;
10     for (i=0; i < NUMT; i++) {
11         pthread_create(&threadIDs[i], NULL,
12                       tfunc, ??? );
13     }
14     for (i=0; i < NUMT; i++) {
15         pthread_join(threadIDs[i], NULL);
16     }
17     return 0;
18 }
```

- Inside the loop several threads are created.
- The same thread function (tfunc) is used for each thread.
 - Common scenario with data parallelism.
- Thread argument passing is used to differentiate between threads.
- Example:
 - tell each thread on which part of a data array it shall work.
 - Tell each cook which tomato to slice.

Passing Arguments to a Thread Function (cont.)

```
1  #include<stdio.h>
2  #include<pthread.h>

3  #define NUMT ...
4  int args[NUMT];          /* argument array */
5  pthread_t threadIDs[NUMT]; /* thread IDs */

6  void * tfunc (void * p) {
7      printf("thread arg is %d\n", * (int *) p );
8  }

9  int main(void) {
10     int i;
11     for (i=0; i < NUMT; i++) {
12         args[i] = i; /* init arg for thread #i */
13         pthread_create(&threadIDs[i], NULL,
14             tfunc, (void *) &args[i] );
15     }
16     for (i=0; i < NUMT; i++) {
17         pthread_join(threadIDs[i], NULL);
18     }
19     return 0;
20 }
```

- In Line 4 we declare an array of integer variables. Each thread has its own “private” array element as argument.



- Each thread will receive a pointer to its “private” array element as argument.
 - In Line 14, `&args[i]` determines the address of the *i*th integer variable.
 - Taking the address of an integer variable gives us an integer pointer
→ type-cast to a `void *`
 - In Line 7, the thread function casts back the `void *` to an `int *`.
 - Dereference to get the actual argument value.

What will not work...

```
1  #include<stdio.h>
2  #include<pthread.h>

3  #define NUMT ...
4  pthread_t threadIDs[NUMT];      /* thread IDs */

5  void * tfunc (void * p) {
6      sleep(10);
7      printf("thread arg is %d\n", * (int *) p );
8  }

9  int main(void) {
10     int i;
11     for (i=0; i < NUMT; i++) {
12         pthread_create(&threadIDs[i], NULL,
13                        tfunc, (void *) &i );
14     }
15     for (i=0; i < NUMT; i++) {
16         pthread_join(threadIDs[i], NULL);
17     }
18     return 0;
19 }
```

- In Line 12, a pointer to the loop index variable is passed as argument to each thread.
- Instead of setting up a unique parameter for each thread, the loop index variable is the shared argument between all threads.
- Leads to disaster quickly...
 - The threads may not access the parameter fast enough to read the value “assigned” to them.
 - See, e.g., sleep (10)...

What will not work... (cont.)

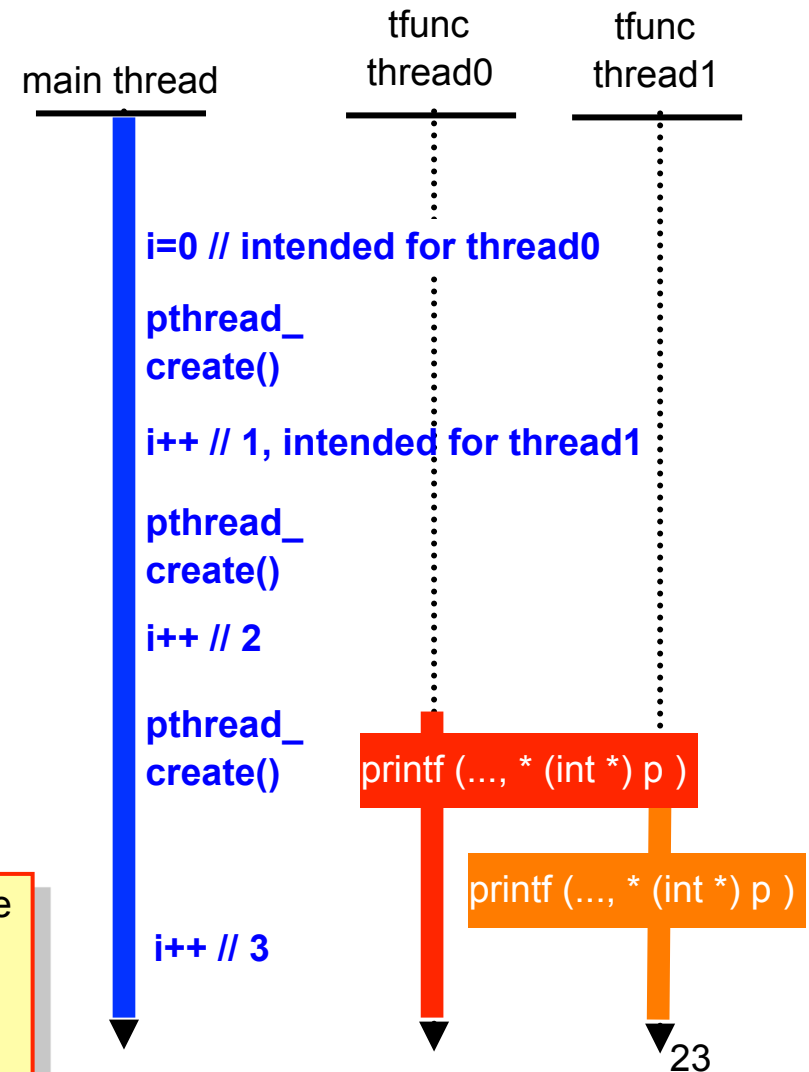
```
1 #include<stdio.h>
2 #include<pthread.h>

3 #define NUMT ...
4 pthread_t threadIDs[NUMT];      /* thread IDs */

5 void * tfunc (void * p) {
6     sleep(10);
7     printf("thread arg is %d\n", * (int *) p );
8 }

9 int main(void) {
10     int i;
11     for (i=0; i < NUMT; i++) {
12         pthread_create(&threadIDs[i], NULL,
13                       tfunc, (void *) &i );
14     }
15     for (i=0; i < NUMT; i++) {
16         pthread_join(threadIDs[i], NULL);
17     }
18     return 0;
19 }
```

By the time thread0 dereferences the pointer to variable i, the value has already been changed by the main thread. Thread0 reads the wrong value (2 instead of 0). Thread1 also reads 2 → two chefs start chopping tomato nr. 2. Ouch!



Our first Race-condition

- The correctness of the previous program depends on how fast the created threads manage to read variable *i* (reading must happen before the main thread updates *i*, otherwise a wrong value is read).
 - (Assume there was no `sleep(10)` statement).
 - Execution speed among threads is determined by the operating system that schedules the threads.
 - Depends on system load also.

Race Condition: *“a situation in which multiple threads read and write a shared data item and the final result depends on the relative timing of their execution”.*

From the previous example:

- shared data item: variable *i*
- final result: what value thread 0 reads as its parameter (0 or >0).
- relative timing: when and how fast thread 0 and the main thread execute relative to each other.

Passing Arguments to a Thread the Dirty Way

```
1 #include<stdio.h>
2 #include<pthread.h>

3 #define NUMT ...
4 pthread_t threadIDs[NUMT];      /* thread IDs */

5 void * tfunc (void * p) {
6     int k = (int) p;      /* cast void * to int! */
7     printf("thread arg is %d\n", k );
8 }

9 int main(void) {
10     int i;
11     for (i=0; i < NUMT; i++) {
12         pthread_create(&threadIDs[i], NULL,
13             tfunc, (void *) i );
14     }
15 }
16 for (i=0; i < NUMT; i++) {
17     pthread_join(threadIDs[i], NULL);
18 }
19 return 0;
20 }
```

Note: another disadvantage of this approach is that we cannot pass a return value back to the main thread via the thread function's argument.

- In Line 14 the value of *i* is type-cast to a void pointer.
- In Line 6, the void pointer is type-cast back to an integer.
- This works, but it mixes by-value and by-reference parameter passing mechanisms.
- Bad programming practice
- Use the `-Wall` argument to GCC, to find out about this.
 - `-Wall` enables all warnings.
 - Recommended in general. Your program should compile without warnings. Eliminate potential bugs *before* they get to you!

Passing more than one argument to a thread

```
1  #include<stdio.h>
2  #include<pthread.h>

3  pthread_t threadID;

4  struct multi_arg{
5      int range;
6      int tomato;
7  };

8  void * tfunc (void * p) {
9      struct multi_arg * ptr = (struct multi_arg *) p;
9      printf("range: %d, tomato %d\n",
10             ptr->range, ptr->tomato);
11 }

12 int main(void) {

13     struct multi_arg ma;
14     ma.range = 1;
15     ma.tomato = 2;

16     pthread_create(&threadID, NULL, &tfunc,
17                   (void *) &ma );
18     return 0;
19 }
```

To pass more than one argument to a thread...

- Create a struct that can hold multiple arguments (Lines 4-7).
- Create a variable of that struct type (lines 13-15).
- Pass a pointer to this struct variable to pthread_create (Line 17).
- In the thread function, cast the void * back to a pointer to the struct (Line 9).

pthread_join()

```
int pthread_join(  
    pthread_t tid,  
    void ** status,  
);  
// Purpose: wait for a thread to terminate.  
// thread ID we are waiting for  
// exit status
```

Note: with pthread_create() we are passing a pointer to a thread ID as argument. With pthread_join() it is the thread ID itself!

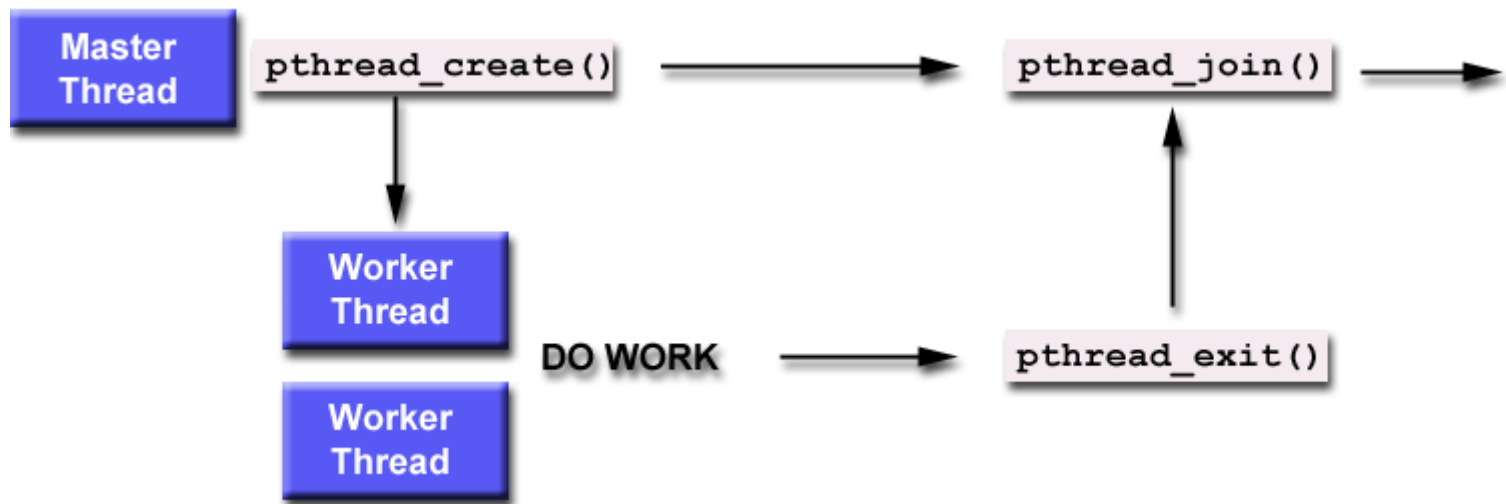
Arguments:

1. The thread ID of the thread we are waiting for.
2. The completion status of the exiting thread will be copied into *status, unless status is NULL, in which case the completion status is ignored.

Note:

- Once a thread is joined, the thread no longer exists. Its thread ID is no longer valid, and it cannot be joined again, e.g. with another thread!

pthread_join() (cont.)



- Joining threads is a mechanism to accomplish synchronization between threads.
- `pthread_join` blocks the calling thread until the specified thread terminates.
- A thread can only join one `pthread_join` call. It is a logical error to attempt to join a thread several times, e.g., from different threads.
- Further synchronization methods will be discussed later
 - mutexes, condition variables

Thread termination

There are three ways to terminate a thread:

- 1) a thread can return from his start routine
- 2) a thread can call `pthread_exit()`
- 3) a thread can be cancelled by another thread.

In each case, the thread is destroyed and his resources become unavailable.

Gotcha: terminating a thread does not release

- dynamic memory allocated by the thread (`malloc`),
- close files that have been opened by the thread.

Cleaning up those resources is the responsibility of the programmer!

Thread termination (cont.)

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS      5

void *PrintHello(void *threadid)
{
    int tid;
    tid = (int)threadid;
    printf("Hello World! It's me, thread #%d!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL,
                           PrintHello, (void *)t);
        if (rc){
            printf("ERROR pthread_create returned %d\n",
                  rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Distinction between pthread_exit() and exit():

- **pthread_exit()** is used to explicitly exit a thread.
 - Example: called after a thread has completed its work and is no longer needed.
- **exit()** terminates the *whole* program, including all threads.
 - Example: used when a program encounters a severe error condition which makes it impossible to continue.
- Both can be called from the main thread or any Pthread.

Thread termination (cont.)

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  void * Hello(void *)
5  {
6      sleep(20);
7      printf("Hello World!\n");
8      pthread_exit(NULL);
9  }
10
11 int main (int argc, char *argv[])
12 {
13     pthread_t thread;
14
15     pthread_create(&thread, NULL,
16                  Hello, NULL);
17
18     pthread_exit(NULL);
19 }
```

- When **main()** terminates, all threads are terminated.
 - This is equivalent to calling `exit()` at the end of `main()`.
 - Example: the Hello thread created in Line 15 won't be able to print its message, because right after thread creation, main terminates...
- Calling `pthread_exit()` in main (Line 18) will allow the other threads to continue.
 - Example: now the Hello thread can sleep for 20 seconds, print its message, and terminate. At this point, the whole program will terminate.

Thread termination (cont.)

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  void * Hello(void *)
5  {
6      sleep(20);
7      printf("Hello World!\n");
8      pthread_exit( (void *) 22 );
9  }
10
11 int main (int argc, char *argv[])
12 {
13     void * rPtr;
14     pthread_t thread;
15
16     pthread_create(&thread, NULL,
17                   Hello, NULL);
18
19     pthread_join ( thread, &rPtr );
20
21     printf("Thread returned %d\n",
22           (long) rPtr);
23
24 }
```

- `pthread_exit()` can be used to pass an exit status to any thread that will join the exiting thread.
- The value of the exit status must be **void ***.
 - We can cast any value to void *.
 - See example, Line 8.
- `pthread_join()` requires a void ** as argument for returning the void * value of the exit status.
 - Void ** is a void * passed by-reference!
- Example:
 - In Line 13 we declare a void * rPtr.
 - Passing rPtr by reference means passing a pointer to rPtr (&rPtr), Line 19.
 - The void * value in rPtr must then be cast back to its original type (long in this example).