Pthreads

Module 4 self study material

Callbacks

Pointers to functions

`pthread_create()`

`pthread_exit()`

`pthread_join()`

Semantics of `fork()` when using threads

Concurrent data modification and data races

Operating systems 2019

1DT044, 1DT096, 1DT003

karl.marklund@it.uu.se

Uppsala university

Callback

In computer programming, a callback is a piece of executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at some convenient time.

The invocation may be immediate as in a **synchronous callback** or it might happen at later time, as in an **asynchronous callback**.

The ways that callbacks are supported in programming languages differ, but they are often implemented with subroutines, lambda expressions, blocks, or **function pointers**.

Pointer

A programming language object, whose value refers to (or points to) another value stored elsewhere in the computer memory using its address is called a pointer.

- A pointer references a location in memory.
- Obtaining the value stored at that location is known as **dereferencing** the pointer.

Pointee

A pointer references a location in memory.

★ We use the term **pointee** for the thing that the pointer points to.

Pointer & Pointee

We use the term pointee for the thing that the pointer points to.

Named variable	Terminology	Storage address	Content
x	pointee	0x10010000	127
xptr	pointer	0x10010004	0x10010000

The **pointer** **xptr** points to the **pointee** **x**.

Dereferencing

A pointer references a location in memory.

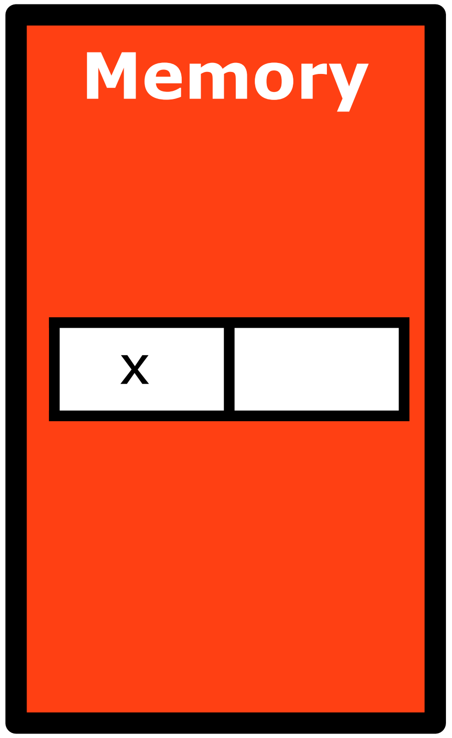
- ★ Obtaining the value stored at that location is known as **dereferencing** the pointer.

Pointers in C (1)

Syntax	Token	Semantics
`int x;`		Variable declaration (x of type int)
`x = 43;`	`=`	Variable assignment
`int *xptr;`	`*`	Pointer declaration (pointer to integer)
`&x`	`&`	Address-of operator
`*xptr`	`*`	Pointer dereference operator

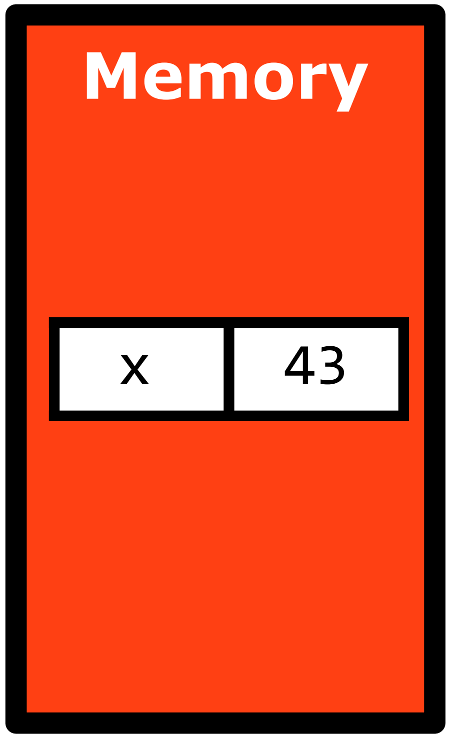
Pointers in C (2)

Syntax	Token	Semantics
`int x;`		Variable declaration (x of type int)



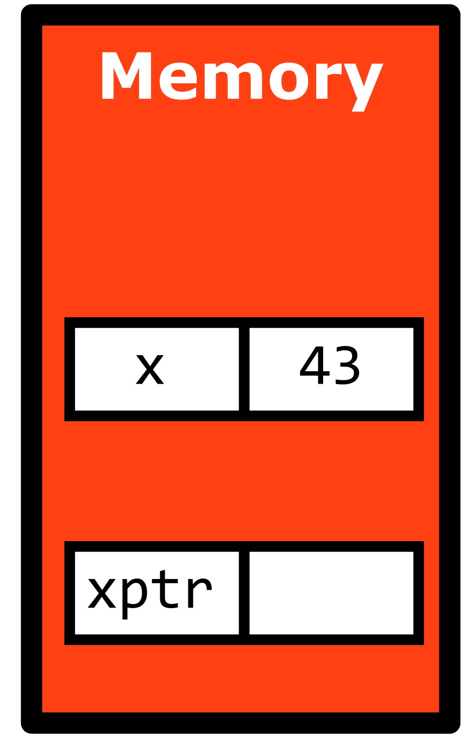
Pointers in C (2)

Syntax	Token	Semantics
`int x;`		Variable declaration (x of type int)
`x = 43;`	`=`	Variable assignment



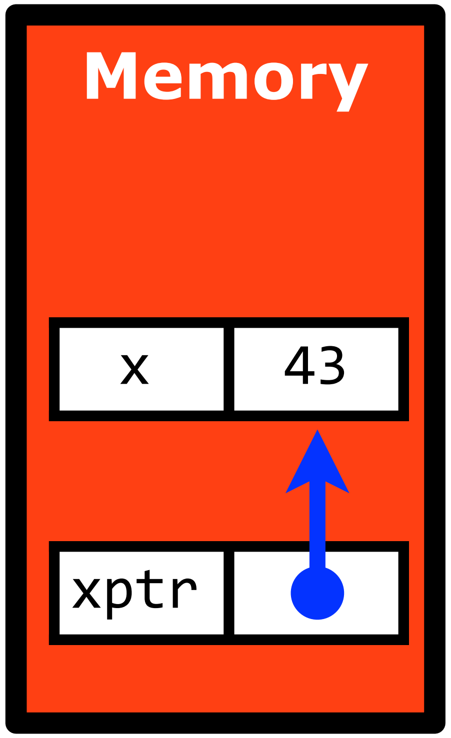
Pointers in C (2)

Syntax	Token	Semantics
`int x;`		Variable declaration (x of type int)
`x = 43;`	`=`	Variable assignment
`int *xptr;`	`*`	Pointer declaration (pointer to integer)



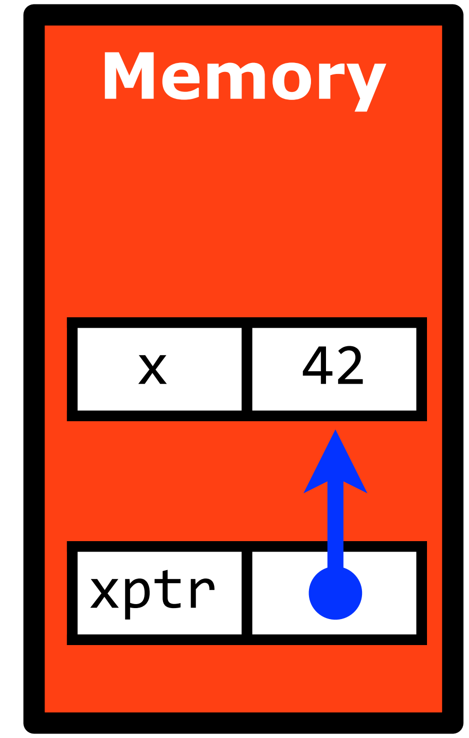
Pointers in C (2)

Syntax	Token	Semantics
`int x;`		Variable declaration (x of type int)
`x = 43;`	`=`	Variable assignment
`int *xptr;`	`*`	Pointer declaration (pointer to integer)
`xptr = &x;`	`&`	Pointer assignment using the address-of operator
`x == 42;`	`==`	Boolean expression (False)

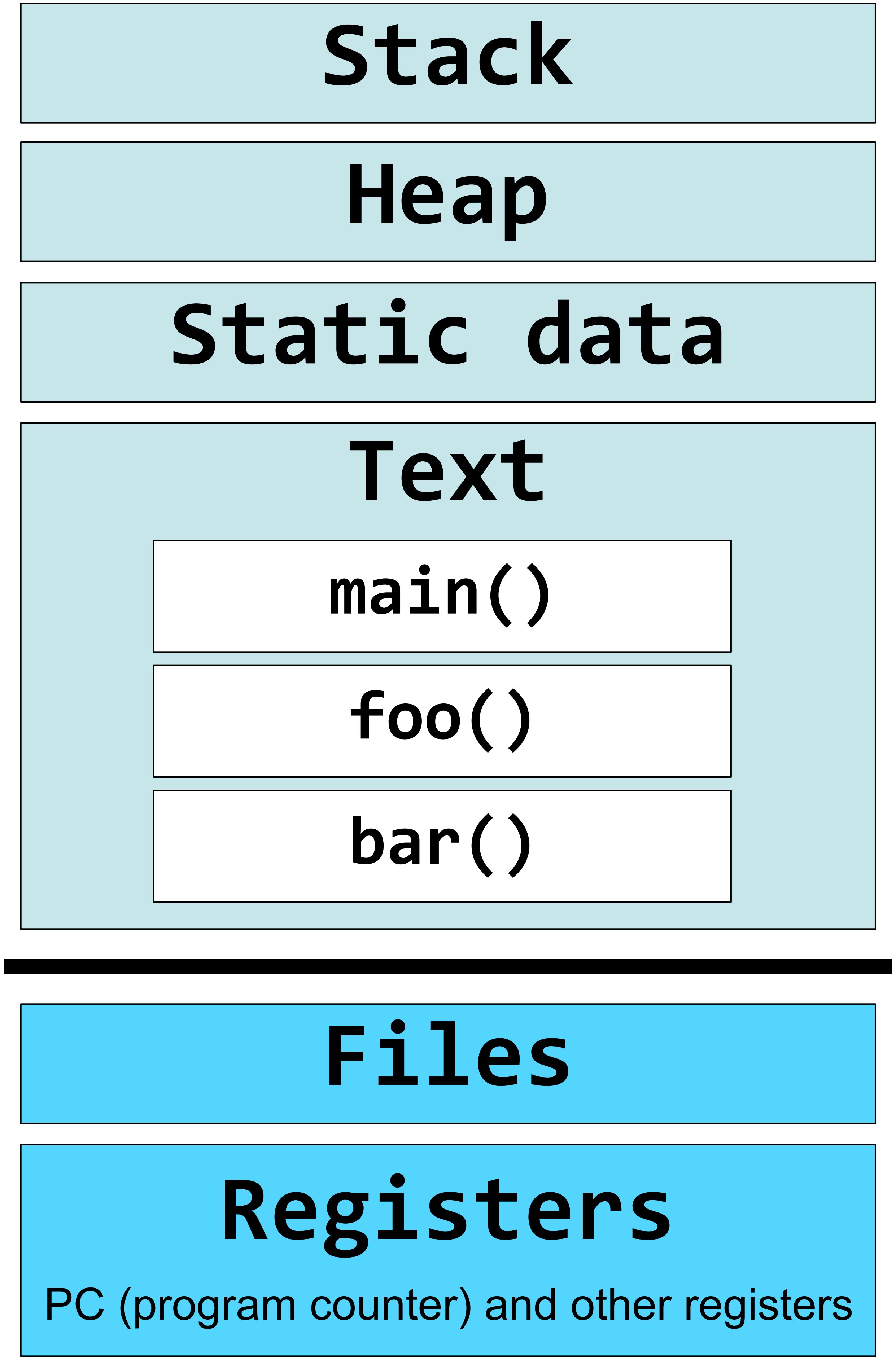


Pointers in C (2)

Syntax	Token	Semantics
`int x;`		Variable declaration (x of type int)
`x = 43;`	`=`	Variable assignment
`int *xptr;`	`*`	Pointer declaration (pointer to integer)
`xptr = &x;`	`&`	Pointer assignment using the address-of operator
`x == 42;`	`==`	Boolean expression (False)
`*xptr = 42;`	`*`	Pointee assignment using pointer dereference
`*xptr == 42;`	`==`	Boolean expression (True) using pointer dereference



Function pointers



A function is simply an address in memory.

It make sense to view functions as pointers.

But pointers to what?

What is the data type for function pointers?

Pointers to functions in C

```
// Function double takes an int as parameter and
// returns an int.
int double(int x) {
   return 2*x;
}

// Pointer to a function taking an int as
// parameter and returning an int.
int (*)(int)

// Declare a pointer with name double_ptr
// pointing to a function taking an int as
// parameter and returning an int.
int (*double_ptr)(int)
```

Ordinary function calls in C

```
#include <stdio.h>

float plus     (float a, float b) { return a+b; }
float minus    (float a, float b) { return a-b; }
float multiply (float a, float b) { return a*b; }
float divide   (float a, float b) { return a/b; }

typedef enum foo {add, sub, mul, div} op_t;

float operation(float a, float b, op_t op) {
  float result;

  switch(op) {
  case add : result = plus (a, b); break;
  case sub : result = minus (a, b); break;
  case mul : result = multiply (a, b); break;
  case div : result = divide (a, b);
  }
  return result;
}


int main(void) {
  printf("Switch:   2*5 = %f\n", operation(2, 5, mul));


}
```

An enumeration.

Integers and enum values can be mixed freely, and all arithmetic operations on enum values are permitted.

We can for example use op_t values in a switch statement.

Ordinary function call.

Callbacks in C - an example

Using callbacks we can replace the switch statement.

The return value of the callback function is float.

The arguments of the callback functions are two floats.

```
float calculate(float a, float b, float (*callback)(float, float)) {
  return callback(a, b);
}
```

Now we can use the callback function.

The name of the parameter is callback.

As with ordinary pointer declarations, we use a star.

Callbacks in C == function pointers

```c
#include <stdio.h>

float plus     (float a, float b) { return a+b; }
float minus    (float a, float b) { return a-b; }
float multiply (float a, float b) { return a*b; }
float divide   (float a, float b) { return a/b; }

typedef enum foo {add, sub, mul, div} op_t;

float operation(float a, float b, op_t op) {
  float result;

  switch(op) {
  case add : result = plus (a, b); break;
  case sub : result = minus (a, b); break;
  case mul : result = multiply (a, b); break;
  case div : result = divide (a, b);
  }
  return result;
}

float calculate(float a, float b, float (*callback)(float, float)) {
  return callback(a, b);
}

int main(void) {
  printf("Switch:   2*5 = %f\n", operation(2, 5, mul));

  printf("Callback: 2+5 = %f\n", calculate(2, 5, plus));
}
```

Terminal — bash — 36×5

bash

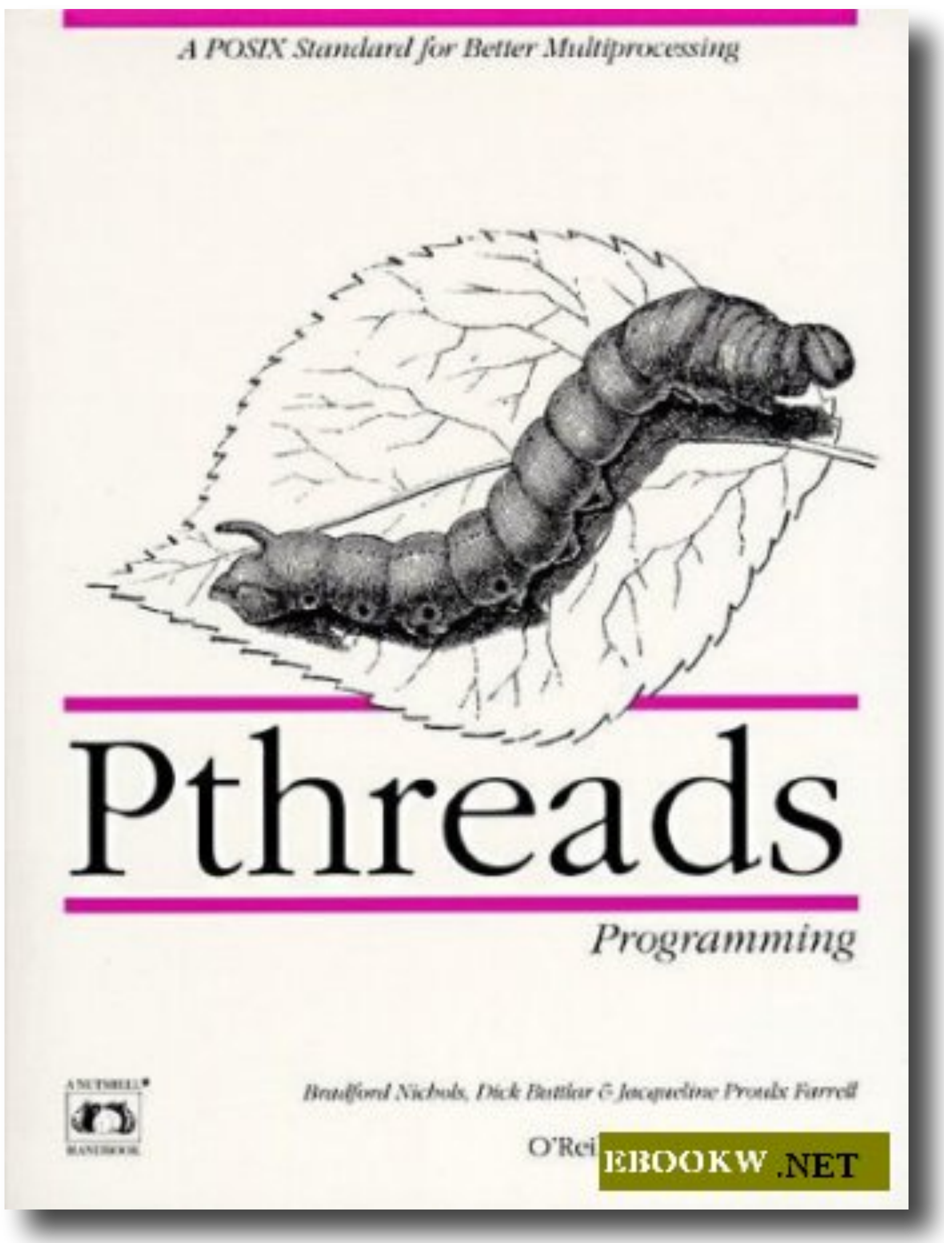
```
> gcc callback.c
> ./a.out
Switch:   2*5 = 10.000000
Callback: 2+5 = 7.000000
>
```

Ordinary function call.

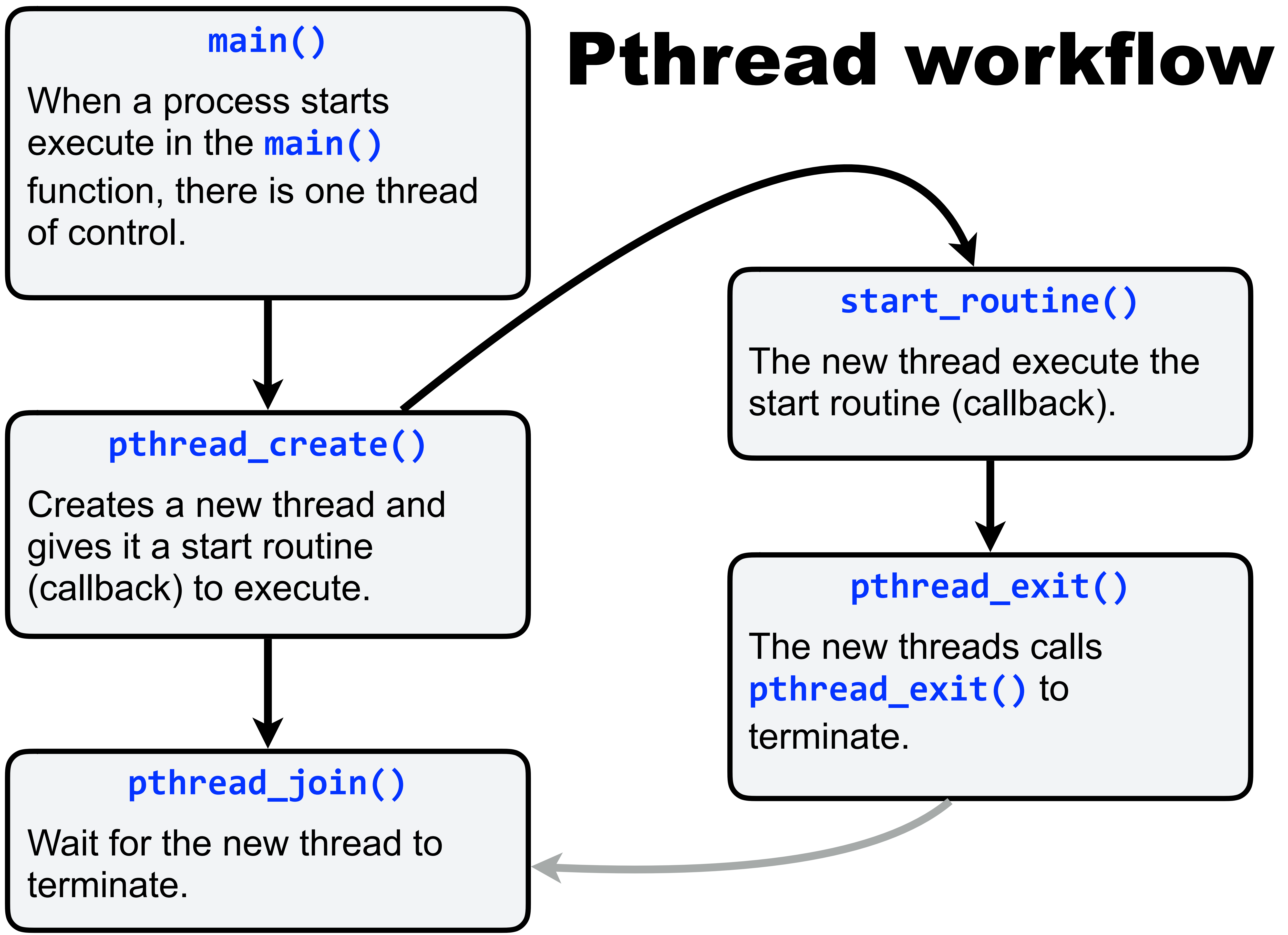
Using a callback (plus).

Pthreads

A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.



- ★ May be provided either as user-level or kernel-level.
- ★ API specifies behaviour of the thread library, implementation depends on the specific library.
- ★ Common in UNIX operating systems (Solaris, Linux, Mac OS X).


Pthread workflow
main()
When a process starts execute in the main() function, there is one thread of control.
pthread_create()
Creates a new thread and gives it a start routine (callback) to execute.
pthread_join()
Wait for the new thread to terminate.
start_routine()
The new thread execute the start routine (callback).
pthread_exit()
The new threads calls pthread_exit() to terminate.

pthread_create()

Creates a new thread executing a start routine (callback). function.

```
#include <pthread.h>

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

On success, the ID of the created thread will be stored here.

What does this mean?

Return type of the function

Name of function pointer

Type of parameter to the function

```
void * ( * start_routine )( void * )
```

pthread_create()

Creates a new thread executing a start routine (callback). function.

```
#include <pthread.h>

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

Upon its creation, the thread executes **start_routine**, with **arg** as its sole argument.

- **start_routine** must return **void***
- **start_routine** must take a single argument of type **void***

pthread_exit()

Terminates the calling thread.

```
#include <pthread.h>

void pthread_exit(void *value_ptr);
```

- The value **`value_ptr`** is made available to any successful join with the terminating thread.
- An implicit call to **`pthread_exit()`** is made when a thread other than the thread in which **`main()`** was first invoked returns from the start routine. The function's return value serves as the thread's exit status.

pthread_join()

Suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated.

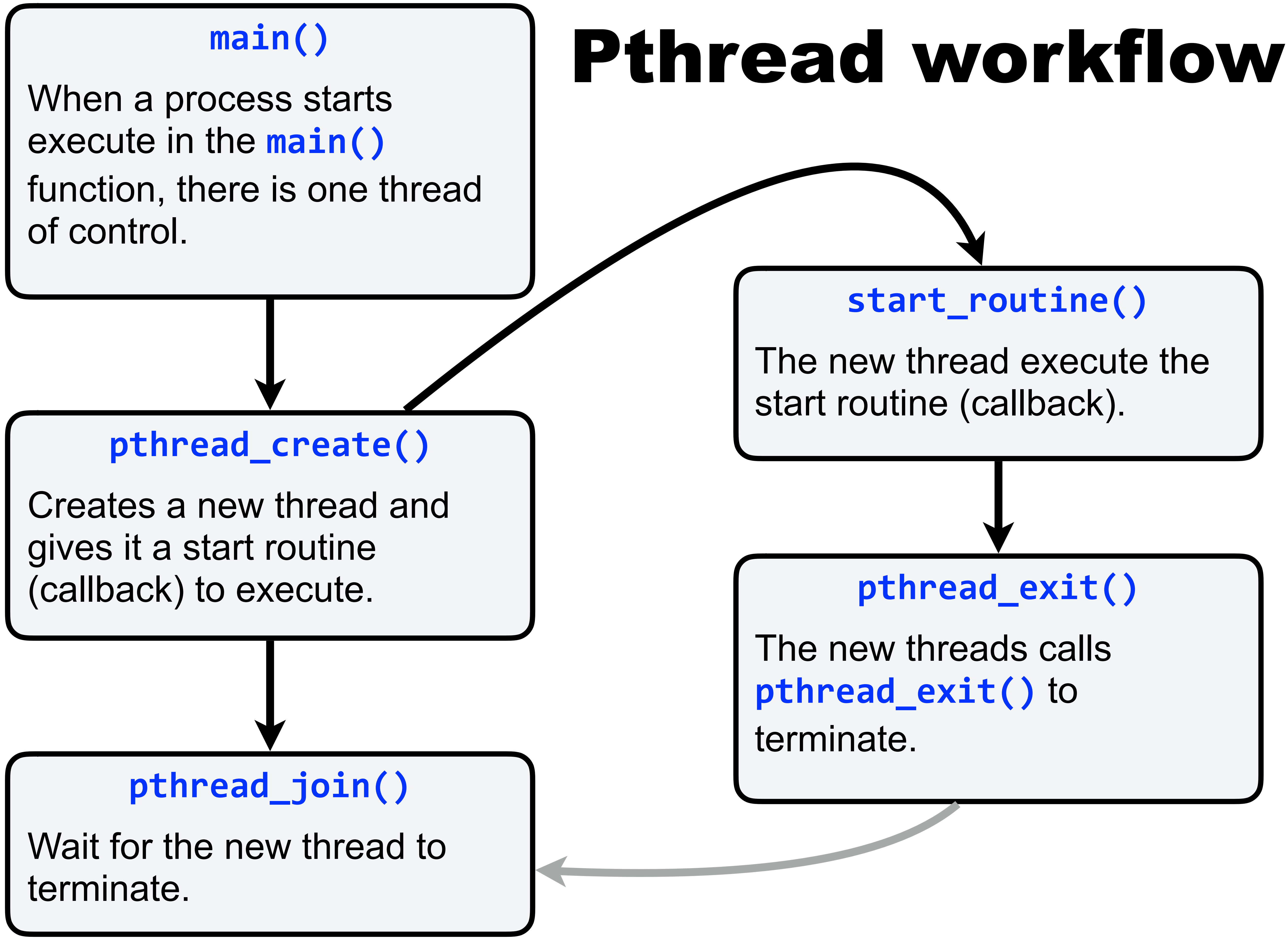
```
#include <pthread.h>

int pthread_join(
      pthread_t thread,
      void **value_ptr);
```

Thread ID of the target thread.

pointer to pointer

- On return from a successful **pthread_join()** call with a non-NULL **value_ptr** argument, the value passed to **pthread_exit()** by the terminating thread is stored in the location referenced by **value_ptr**.
- If successful, **pthread_join()** will return zero. Otherwise, an error number will be returned to indicate the error.


Pthread workflow
main()
When a process starts execute in the main() function, there is one thread of control.
pthread_create()
Creates a new thread and gives it a start routine (callback) to execute.
pthread_join()
Wait for the new thread to terminate.
start_routine()
The new thread execute the start routine (callback).
pthread_exit()
The new threads calls pthread_exit() to terminate.

pthreads_create_exit_null_join.c

This program creates four threads and wait for all of them to terminate.

```
$ ./bin/pthreads_create_exit_null_join
main() - before creaing new threads
  thread 0 - hello
  thread 1 - hello
  thread 2 - hello
  thread 3 - hello
main() - thread 0 terminated
main() - thread 1 terminated
main() - thread 2 terminated
main() - thread 3 terminated
main() - all new threads terminated
$
```

```c
void* hello(void* arg) {
  int i = *(int*) arg;
  printf("  thread %d - hello\n", i);
  pthread_exit(NULL);
}
```

This is the start routine each of the threads will execute.

Every start routine must take **`void*`** as argument and return **`void*`**.

When creating a new thread we will use a pointer to an integer as argument, pointing to an integer with the thread number.

Here we first cast from **`void*`** to **`int*`** and then dereference the pointer to get the integer value.

Terminate the thread by calling **`pthread_exit(NULL)`**. Here **`NULL`** means we don't specify a termination status.

```c
/* An array of thread identifiers, needed by
   pthread_join() later. */
pthread_t tid[NUM_OF_THREADS];

/* An array to hold argument data to the hello()
   start routine for each thread. */
int arg[NUM_OF_THREADS];

/* Attributes (stack size, scheduling information
   etc) for the new threads. */
pthread_attr_t attr;

/* Get default attributes for the threads. */
pthread_attr_init(&attr);
```

Declaration of arrays used to store thread IDs and arguments for each threads start routine, the `hello()` function.

Use default attributes when creating new threads.

```c
/* Create new threads, each executing the
    hello() function. */
 for (int i = 0; i < NUM_OF_THREADS; i++) {
   arg[i] = i;
   pthread_create(&tid[i], &attr, hello, &arg[i]);
 }
                        1         2       3        4
```

1) Pass in a pointer to `tid_t`. On success `tid[i]` will hold the thread ID of thread number `i`.

2) Pass a pointer to the default attributes.

3) The start routine (a function pointer).

4) A pointer to the argument for the start routine for thread number `i`.

```c
/* Wait for all threads to terminate. */
for (int i = 0; i < NUM_OF_THREADS; i++){
  if (pthread_join(tid[i], NULL) != 0) {
    perror("pthread_join");
    exit(EXIT_FAILURE);
  }
  printf("main() - thread %d terminated\n", i);
}

printf("main() - all new threads terminated\n");
```

1) Wait for thread with thread ID `tid[i]` to terminate.

2) Pass `NULL` here means we don't care about the exit status of the terminated thread.

`pthreads_and_fork.c`

Does `fork()` duplicate only the calling thread or all threads?

- ★ Let’s write a small program to find out.

```c
void* worker(void *arg) {
  int id = *(int*) arg;
  printf("PID = <%ld> Thread %d says: Hello!\n", (long) getpid(), id);
  for (int i = 2; i > 0; i--) {
    printf("PID = <%ld> Thread %d says: I've got %d seconds to live.\n",
           (long) getpid(), id, i);
    sleep(1);
  }
  return NULL;
}
```

This is the start routine each of the threads will execute.

Every start routine must take `void*` as argument and return `void*`.

When creating a new thread we will use a pointer to an integer as argument, pointing to an integer with the thread number.

Here we first cast from `void*` to `int*` and then dereference the pointer to get the integer value.

Each thread sleeps for 2 seconds before terminating.

```c
/* Create threads. */
for (int i = 0; i < NUM_OF_THREADS; i++) {
  targ[i] = i;
  pthread_create(&tid[i], &attr, worker, &targ[i]);
}

/* Create a new process - will the threads be duplicated or not? */
switch(fork()) {
case -1:
  perror("Forked failed.");
  exit(EXIT_FAILURE);
case 0:
 printf("PID = <%ld> ==> Child after fork()!\n", (long) getpid());
 break;
default:
  printf("PID = <%ld> ==> Parent after fork()!\n", (long) getpid());
}
```

First create the threads, then `fork()`.

```
/* Wait for all threads to terminate. */
 for (int i = 0; i < NUM_OF_THREADS; i++){
1  if (pthread_join(tid[i], NULL) == 0)
     printf("PID = <%ld> ==> Joinig thread %d.\n", (long) getpid(), i);
   else {
     printf("PID = <%ld> ==> No thread to join.\n", (long) getpid());
   }
 }
```

1) Check the return value from `pthread_join()` if there are threads to joint with.

pthreads_and_fork.c

Does **fork()** duplicate only the calling thread or all threads?

```
$ make
gcc -std=gnu99 -Werror -Wall -O2 src/pthreads_and_fork.c -o bin/
pthreads_and_fork
$> ./bin/pthreads_and_fork
PID = <48270> ==> Lets create some threads.
PID = <48270> Thread 0 says: Hello!
PID = <48270> Thread 0 says: I've got 2 seconds to live.
PID = <48270> Thread 1 says: Hello!
PID = <48270> Thread 1 says: I've got 2 seconds to live.
PID = <48270> ==> Parent after fork()!
PID = <48271> ==> Child after fork()!
PID = <48271> ==> No thread to join.
PID = <48271> ==> No thread to join.
PID = <48270> Thread 1 says: I've got 1 seconds to live.
PID = <48270> Thread 0 says: I've got 1 seconds to live.
PID = <48270> ==> Joining thread 0.
PID = <48270> ==> Joining thread 1.
$
```

Semantics of `fork()` when using threads

Does `fork()` duplicate only the calling thread or all threads?

- ★ A process shall be created with a single thread.
- ★ If a multi-threaded process calls fork(), the new process shall contain a replica of the calling thread and its entire address space

pthreads_unsynchronized_concurrency.c

Given a string, write a program using Pthreads to concurrently:

- ★ calculate the length of the string.
- ★ calculate the number of spaces in the string.
- ★ change the string to uppercase.
- ★ change the string to lowercase.

Header files and global data

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h> // sleep()

#define NUM_OF_THREADS 4

/* A global string for the threads to work on. */
char STRING[] = "The string shared among the threads.";

/* Global storage for results. */
int LENGTH;
int NUM_OF_SPACES;
```

Start routines (1)

```
void* length(void *arg) {





}

void* num_of_spaces(void *arg) {







}
```

The implementation details of these functions are not important for the purpose of this exercise.

But, note that to for Pthreads to be able to use these functions as start routines for the threads, they must all be declared **`void*`** and take a single argument of type **`void*`**.

Start routines (1)

```
void* length(void *arg) {
  char *ptr = (char*) arg;
  int i = 0;
  while (ptr[i]) i++;
  LENGTH = i;
}

void* num_of_spaces(void *arg) {
  char *ptr = (char*) arg;
  int i = 0;
  int n = 0;

  while (ptr[i]) {
    if (ptr[i] == ' ') n++;
    i++;
  }
  NUM_OF_SPACES = n;
}
```

The implementation details of these functions are not important for the purpose of this exercise.

But, note that to for Pthreads to be able to use these functions as start routines for the threads, they must all be declared **`void*`** and take a single argument of type **`void*`**.

Start routines (2)

```
void* to_uppercase(void *arg) {
  char *ptr = (char*) arg;
  int i = 0;

  while (ptr[i]) {
    /* Defer execution just a bit to get a more random thread interleaving. */
    sleep(1);
    if (ptr[i] <= 'z' && ptr[i] >= 'a') {
      ptr[i] -= 0x20;
    }
    i++;
  }
}
```

```
void* to_lowercase(void *arg) {
  char *ptr = (char*) arg;
  int i = 0;

  while (ptr[i]) {
    /* Defer execution just a bit to get a more random thread interleaving. */
      sleep(1);
    if (ptr[i] <= 'Z' && ptr[i] >= 'A') {
      ptr[i] += 0x20;
    }
    i++;
  }
}
```

main() - step 1

```
int main(int argc, char *argv[]) {
  /* An array of thread identifiers, needed by pthread_join() later... */
  pthread_t tid[NUM_OF_THREADS];

  /* Attributes (stack size, sche
  pthread_attr_t attr;

  /* Get default attributes for the thre . */
  pthread_attr_init(&attr);
```

We could simply call **pthread_create()** four times using the four different string functions:

- ★ `length()`
- ★ `num_of_spaces()`
- ★ `to_upppercase()`
- ★ `to_lowercase()`

, for example like this.

```
pthread_create(&tid[i], &attr, length, STRING);
```

But, it is more practical (and fun) to collect pointers to all the functions in an array.

Array of function pointers in C

The type of a function pointer is just like the function declaration, but with "(*)" in place of the function name. So a pointer to:

```
int foo( int )
```

would be:

```
int (*)( int )
```

In order to name an instance of this type, put the name inside (*), after the star, so:

```
int (*foo_ptr)( int )
```

declares a variable called foo_ptr that points to a function of this type.

Arrays follow the normal C syntax of putting the brackets near the variable's identifier, so:

```
int (*foo_ptr_array[2])( int )
```

declares a variable called foo_ptr_array which is an array of 2 function pointers.

The syntax can get pretty messy, so it's often easier to make a typedef to the function pointer and then declare an array of those instead:

```
typedef int (*foo_ptr_t)( int );
foo_ptr_t foo_ptr_array[2];
```

Source: http://stackoverflow.com/questions/5488608/how-define-an-array-of-function-pointers-in-c

main() - step 2

```
int main(int argc, char *argv[]) {
  /* An array of thread identifiers, needed by pthread_join() later... */
  pthread_t tid[NUM_OF_THREADS];

  /* An array of pointers to the callback functions. */
  void* (*callback[NUM_OF_THREADS]) (void* arg) =
    {length,
     to_uppercase,
     to_lowercase,
     num_of_spaces};

  /* Attributes (stack size, scheduling information) for the threads. */
  pthread_attr_t attr;

  /* Get default attributes for the threads. */
  pthread_attr_init(&attr);

  /* Create one thread running each of the callbacks. */
  for (int i = 0; i < NUM_OF_THREADS; i++) {
    pthread_create(&tid[i], &attr, *callback[i], STRING);
  }

  /* Wait for all threads to terminate. */
  for (int i = 0; i < NUM_OF_THREADS; i++){
    pthread_join(tid[i], NULL);
  }

  /* Print results. */
  printf("       lenght(\"%s\") = %d\n", STRING, LENGTH);
  printf("num_of_spaces(\"%s\") = %d\n", STRING, NUM_OF_SPACES);
}
```

Array of callbacks

```
/* Pointer to pthread start function. */
typedef void* (*callback_ptr_t)(void* arg);


int main(int argc, char *argv[]) {
  /* An array of thread identifiers, needed by pthread_join() later... */
  pthread_t tid[NUM_OF_THREADS];

  /* An array of pointers to the callback functions. */
  callback_ptr_t callback[NUM_OF_THREADS] = {
    length,
    to_uppercase,
    to_lowercase,
    num_of_spaces
  };
```

Array of callbacks - the rest of main()

```
  /* Attributes (stack size, scheduling information) for the threads. */
  pthread_attr_t attr;

  /* Get default attributes for the threads. */
  pthread_attr_init(&attr);

  /* Create one thread running each of the callbacks. */
  for (int i = 0; i < NUM_OF_THREADS; i++) {
    pthread_create(&tid[i], &attr, callback[i], STRING);
  }

  /* Wait for all threads to terminate. */
  for (int i = 0; i < NUM_OF_THREADS; i++){
    pthread_join(tid[i], NULL);
  }

  /* Print results. */
  printf("        lenght(\"%s\") = %d\n", STRING, LENGTH);
  printf("num_of_spaces(\"%s\") = %d\n", STRING, NUM_OF_SPACES);

}
```

Writing and reading results

```
int main(int argc, char *argv[]) {
  /* An array of thread identifiers, needed by pthread_join() later... */
  pthread_t tid[NUM_OF_THREADS];

  /* An array of pointers to the callback functions. */
  void* (*callback[NUM_OF_THREADS]) (void* arg) =
    {length,
     to_uppercase,
     to_lowercase,
     num_of_spaces};

  /* Attributes (stack size
  pthread_attr_t attr;

  /* Get default attributes for the threads. */
  pthread_attr_init(&attr);

  /* Create one thread running each of the callbacks. */
  for (int i = 0; i < NUM_OF_THREADS; i++) {
    pthread_create(&tid[i], &attr, *callback[i], STRING);
  }

  /* Wait for all threads to terminate. */
  for (int i = 0; i < NUM_OF_THREADS; i++){
    pthread_join(tid[i], NULL);
  }

  /* Print results. */
  printf("       lenght(\"%s\") = %d\n", STRING, LENGTH);
  printf("num_of_spaces(\"%s\") = %d\n", STRING, NUM_OF_SPACES);
}
```

In this example, each thread simply writes their results directly to the shared variables **`STRING`**, **`LENGTH`** and **`NUM_OF_SPACES`**.

Test runs

Terminal — a.out — 74×17

```
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: gcc -std=c99 pthreads.c
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: ./a.out
       lenght("tHE STRING SHared among the threads.") = 36
num_of_spaces("tHE STRING SHared among the threads.") = 5
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: ./a.out
       lenght("THE STRING SHARED AMONG THE THREADS.") = 36
num_of_spaces("THE STRING SHARED AMONG THE THREADS.") = 5
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: ./a.out
       lenght("THE STRING SHARED among the threads.") = 36
num_of_spaces("THE STRING SHARED among the threads.") = 5
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: ./a.out
       lenght("THE STRING SHARED AMONG THE THREADS.") = 36
num_of_spaces("THE STRING SHARED AMONG THE THREADS.") = 5
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: ./a.out
       lenght("tHe string shared amOng the threads.") = 36
num_of_spaces("tHe string shared amOng the threads.") = 5
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: 
```

Because the threads execute and operate on the same data concurrently, the result of **to_uppercase()** and **to_lowercase()** will be unpredictable due to **data races**.