



NTNU – Trondheim
Norwegian University of
Science and Technology

Creating and removing pthreads

There's an old joke...

- *A programmer had a problem, and thought
"I know, I'll solve it with threads!"*
- *has Now problems. two programmer the*
- We have already seen similar issues using parallel processes
- They appear perhaps even more easily with threads, because there are no explicit messages in the code that synchronize it



Enabling pthreads

- Unlike MPI, it is not necessary to install any additional software in order to start using pthreads
 - The P stands for “POSIX”
 - “POSIX” stands for “Portable Operating System Interface” (...X?)
 - If your O/S supports them, you already have everything needed
 - If your O/S *doesn't* support them, it must be a very exotic one
- Your compiler should come with a header file, so that
 - #include <pthread.h>**
inserts prototypes for all the functions, and putting
 - pthread**
among the compiler flags handles correct linking



Creating a function to use

- As we have spoken about, threads are closely related to function calls
 - They have their own call stack, but share everything else
- Spawning a thread basically amounts to writing
“start this function call in the background, and return immediately”
...so we're going to need a function to call.
- Thread-able functions have this type signature in C:
`void * my_function (void *argument);`
i.e. it accepts a **void *** argument, and returns a **void *** value



Into the void

- ‘void’ means different things in C, depending on where it appears
 - When it stands in for an argument list it means “exactly zero arguments”, as in

```
int make_random_number ( void ); // No input needed
```
 - When it stands in for a type, it means “no particular type”, as in

```
void just_do_something ( int x ); // No return value needed
```
- Pointers-to-void (void *) aren’t “pointers to nothing”
 - They **are** pointers to something, but
 - They don’t know *what type of value* they point at



Pointers with types

- Connecting a type to a pointer allows a C compiler to check that you're using it correctly

- If we define

- `void do_something_useful (int x);`

- and call

- `double pi = 3.141593;`

- `do_something_useful (pi);`

- we get a warning that there is probably some mistake



Pointers with types

- Connecting a type to a pointer also defines how to handle arithmetic and dereferencing

If we define

```
int16_t *ten_sixteen_bit_ints = malloc ( 10 * sizeof(int16_t) );
```

then

```
ten_sixteen_bit_ints[7]
```

and

```
*(ten_sixteen_bit_ints+7)
```

both mean “*addr. ‘ten_sixteen_bit_ints’ plus 7 times sizeof(int16_t)*”

because the pointer is pointing at 16-bit ints



Pointers without types

- A (void *) is just a memory address that could point at anything
 - We can't do anything directly with it, because it doesn't say anything about how to interpret the data it points at
 - We can't add and subtract with it, because it doesn't say how big an address change "+1" should correspond to
 - We **can** turn any type of pointer into a (void *), just discard the type
 - We can also turn a (void *) into any other type of pointer, just add the desired type
- This basically removes all protections from C's type system
 - On the assumption that you know exactly what you're doing when you explicitly choose to disable them



Case in point:

```
% cat unsafe.c
#include <stdio.h>
#include <stdint.h>

int main () {
    double hello = 1.81630607015975e-310;
    printf ( "%s\n", (char *)&hello );
}
% make unsafe
cc -O2 -pipe unsafe.c -o unsafe
% ./unsafe
Hello!
% █
```

- This program works because the bit pattern of that horrible floating point number coincides with the string “Hello!”
- It “*shouldn’t*” work, because it reinterprets the address of a number as the address of some text, and numbers aren’t text
- It **won’t** work (the same way) on a CPU with a different internal floating point representation
- The C compiler just shuts up and generates code when we tell it to



Back to the threads

```
void * my_function ( void *argument );
```

- This function takes an address to an un-checked thing as its argument
 - The body of this function will have to cast the argument into whatever type of thing it expects to receive
- It also returns an address to some un-checked thing
 - The caller of this function will have to cast the return value into whatever type of thing it expects to get back
- Unexpected events will occur if you pass it the wrong type of argument, or mis-interpret the answer
 - The compiler will be oblivious to what the mistake is



This is a recurring theme

- The whole design of pthreads is permeated by this idea of a “*gentleman’s agreement*” between calling functions and called functions:
 - Callers have to ensure that the arguments are correct, threads have to trust that the caller will interpret the answer correctly
 - Threads can’t actually prevent each other from entering a critical section, you have to program them so that they all respect the state of a locking mechanism everyone can see
 - Threads can’t forcibly exclude each other from overwriting shared values, you have to program them so that they don’t try to

Creating a thread

(finally!)

- When you have defined your

```
void * my_function ( void *arg );
```

it can be launched in a thread like this:

```
thread_t my_thread;
```

```
pthread_create ( &my_thread, NULL, &my_function, NULL );
```

Thread handle

Thread attributes

Pointer to function

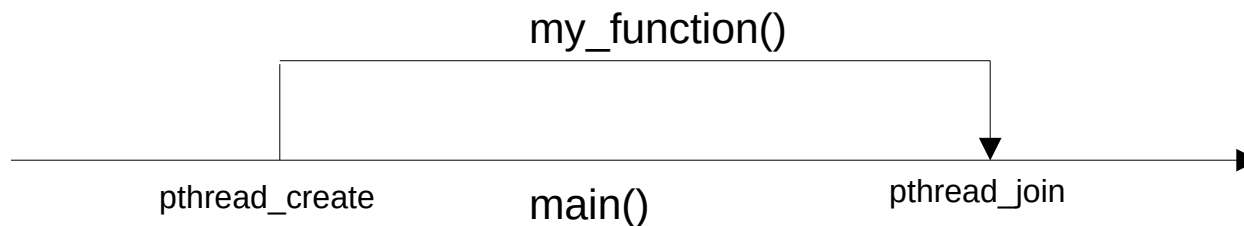
(void *) to the
argument



NTNU – Trondheim
Norwegian University of
Science and Technology

Waiting for a thread to finish

- When we need that thread to finish its work, we can wait for it like this:
`pthread_join (my_thread, NULL);`
- Those were a lot of NULLs, we'll get back to them momentarily
- What we did so far was to make control flow split and merge over time:



(time goes from left to right)



NTNU – Trondheim
Norwegian University of
Science and Technology

Arguments and return values

(two of those NULLs)

- We can now try it out with passing arguments and obtaining return values
 - Today's example archive subdirectory '01_hello_world' contains a program 'arguments_and_return.c' which utilizes the in/out arguments of `pthread_{create/join}` to pass some values
 - It's a little redundant to return the result-pointer when there is only one value it could ever have, but we can imagine a function that chooses between several places to put the answer
 - It serves mostly as an illustration, anyway



The last of the NULLs

- The second argument to `pthread_create` is a pointer to a struct that has the type `pthread_attr_t`
- Such structs can be created and deleted with `pthread_attr_init` and `pthread_attr_destroy`
- It lets you control O/S-dependent info about the thread
 - Maximal lifetime
 - Size of its stack
 - Scheduling priority
 - *etc.*
- The defaults we get by not using this will suffice



Now you've seen it

- The arguments/return thing is important for programs that aim to be modular
 - Pthreads were originally introduced for *concurrent* programs
 - e.g. programs that spawn lots of threads with separate tasks, and mostly wait around for something to do
- Most parallel HPC programs have a different purpose
 - We usually don't want more threads than we have CPU cores
 - All the threads tend to do the same thing
 - They tend to do it to large amounts of shared data, making it impractical/pointless to pass copies of every value in and out of function calls



HPC-style pthreads

- As with MPI programs, most of our needs are covered if we can
 - Work on some huge, global problem
 - Give each thread a unique index among the total number of threads
 - Calculate its part of the global problem from those two numbers
- A very common trick is to use the (void *) argument as an integer instead
 - We can store the total number of threads in a global variable
 - We don't need the argument to represent a memory address, so we can just use the fact that a memory address is an integer

This is terrible software design

- I know, right?
- The variables don't have descriptive names
 - The argument pointer isn't a pointer, and it doesn't point at any arguments
- The whole program state is global
 - That's where the threads can access it directly
 - Individual blocks of local work-shares would double the total memory requirement, and require a lot of copying
- I didn't invent this pattern, you will find that our book uses it as well



Hello, threads

- In the same example directory, there is also a program called 'hello_pthreads.c'
- It works in pretty much the same way as our MPI hello world program, except that
 - The collective isn't there from the start, the main function has to launch every co-worker
 - They don't all last until the end of the program, threads disappear when they are joined on completion
 - Threads that don't join before the program exits will just vanish without a trace



We have been here before

- We now have the tools to parallelize the same kind of tasks that we could handle with only `MPI_{ Init, Finalize, Comm_rank, Comm_size }`
 - *Embarrassingly parallel* problems don't need the threads to synchronize/communicate
 - If a sub-problem is only a function of a worker's index, it can be handled with just `pthread_{ create, join }`
- Next lecture, we will look at synchronization and communication, and solve a problem that requires it

