Thread Creation

```
int pthread_create(pthread_t *ID,
                   pthread_attr_t *attr,
                   void *(*function)(void *),
                   void *arg);
```

Return Values

UNIX system calls:

- Return 0 on success, -1 on any error
- `errno` specifies which error

Pthread library calls:

- Return 0 on success, error code otherwise
- No concept like `errno` — static variable foils reentrancy

Thread ID, I

UNIX: process ID is an `int` (though called "`pid_t`")

ID of thread is a `pthread_t`

Usually, `pthread_t` is pointer to thread's control block

But: `pthread_t` is "opaque" – cannot make any assumption about implementation

Thread ID, II

For example, cannot write:

```
pthread_t t1;
pthread_t t2;
    ...
if (t1 == t2) {
  // code that depends on t1 and t2 being same thread
}
```

even though this may/probably work as intended

Instead must write:

```
if (pthread_equal(t1, t2)) {
  // code that depends on t1 and t2 being same thread
}
```

Thread ID, III

To get thread's own ID:

```
pthread_t pthread_self()
```

Analogous to `getpid(2)`, which gets process ID

Thread Creation Revisited

```
int pthread_create(pthread_t *ID,
                   pthread_attr_t *attr,
                   void *(*function)(void *),
                   void *arg);
```

ID is out parameter

Skip "`pthread_attr_t *attr`" for now

Signature of "`function`:"

```
void *function(void *arg);
```

Aside: Void *

Type "`void *`" means "any type of pointer"

So

```
void *function(void *arg);
```

means "accepts any type of pointer, returns any type of pointer"

What to do: write function body using desired pointer types, casting to/from `void *` at beginning/end

Example

```
void *function(void *arg) {
  argument_t *input;  /* argument_t is
                         actual type of argument */
  return_t *ret;      /* return_t is
                         actual type of return value */

  input = (argument_t *) arg;
            ...
  ret = ... some return_t value ...;
            ...
  return (void *) ret;
}
```

Aside: How to Pass Arbitrary Arguments

`pthread_create`'s argument function takes only 1 argument

Q: What if it should take >1 argument?

A: Make single `void *` argument point to a `struct` that contains all the real arguments

Example

```
/*
 *  function takes 2 arguments, int and string
 */
struct argument_t {
    int arg1;
    char *arg2;
}

void *function(void *arg) {
    struct argument_t *input;
    int x;
    char *y;

    input = (struct argument_t *) arg;
    x = input->arg1;
    y = input->arg2;
}
```

Of course caller must pack the `struct` and pass pointer to it in call to `pthread_create`

Thread Start

First thread of process starts executing `main()`

Later threads start with function passed as argument to `pthread_create`

Termination

Three ways for thread to terminate:

1. Start function returns — similar to `main` returning in UNIX process
2. Call `pthread_exit` — similar to calling `_exit(2)` to terminate process
3. Call `pthread_cancel`

`void *` value is returned by thread that returns or exits — similar to child process exit code

Join, I

In UNIX, parent process often waits for child termination using calls like `waitpid(2)`

Similarly, one thread can wait for another to terminate:

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

Note "`void **result`" means "pointer to area where a `void *` value exists"

Join, II

Also similar to UNIX: Pthreads implementation saves return value of terminated thread in case another thread later decides to join

BUT: *unlike with UNIX* `fork/exit`, *there need be no parent-child relationship between these threads!*

ANY thread can call `pthread_join` with ANY other thread as argument

Detach

A "detached" thread can never be joined — Pthreads implementation throws away its return value

A thread can be forcibly detached by another:

```
pthread_t ID;
    ...
pthread_detach(ID);
```

or can detach itself:

```
pthread_detach(pthread_self());
```

Benefit of detaching: saves resources, since entire thread data structure can be reclaimed when it terminates

Gotcha

There is one special value that a thread should never return: `PTHREAD_CANCELED`

A "canceled" thread is one that was killed before it had chance to terminate itself

From `/usr/include/pthread.h`:

```
#define PTHREAD_CANCELED        ((void *) -1)
```

This value is returned to a thread that calls `pthread_join` with cancelled thread as argument

Thread Attributes, I

Aspects of a thread's behavior or resource usage called "attributes"

`pthread_attr_t` is `struct` containing all this info

Common attributes:

- Size of stack
- Location of stack
- Is it detached or still joinable?
- Is it cancelable?
- Scheduling policy
- Many others, incl. vendor-specific attributes

Thread Attributes, II

Implementation may choose not to implement some attributes

If attribute is implemented, compile-time constant will be defined; e.g.,

```
_POSIX_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_ATTR_STACKADDR
_POSIX_THREAD_PRIORITY_SCHEDULING
```

Therefore:

```
#ifdef _POSIX_THREAD_ATTR_STACKSIZE
    ... code to set thread stack size ...
#endif
```

Thread Attributes, III

Implemented attributes have default values that user can change

There are lots of calls to read/write individual attributes

To accept all defaults, pass `NULL` argument to `pthread_create`

To NOT accept defaults:

1. Create `pthread_attr_t` object
2. Pass it to `pthread_attr_init()` — to initialize to defaults
3. Make calls to change individual attribute values
4. Pass modified `pthread_attr_t` object to `pthread_create`

Example: Typical Use

```
pthread_t TID;

void *func(void *) {
    ... function body ...
    return NULL;     /* or non-NULL "void *" value */
}

struct argument_t {
    int actual_arg;
} arg;
arg.actual_arg = 12345;

int rc;                                 /* return code */

rc = pthread_create(&TID,               /* TID is out parameter */
                    NULL,               /* no attributes */
                    func,               /* function */
                    (void *)&arg); /* function argument */
```

Thread Stacks

Setting stack size/location is obviously non-portable — do you really want to do this???

Default thread stack much smaller than default process stack segment

Only 1st thread has stack allocated in process stack segment

Later threads have stack allocated from "heap" segment of process address space (e.g., allocated by `malloc`)

Minimum guaranteed stack size given by `PTHREAD_STACK_MIN`