Reasons to Use Threads Instead of Processes

1. Need many schedulable units — threads faster to create, destroy, switch

2. Much shared data — inefficient to pass it around using system calls like `read`, `write`

Thread Creation

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

Pthread Return Values

UNIX:

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

Pthreads:

- 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 cannot make any assumption about implementation

Thread ID, II

Cannot write:

```
pthread_t t1;
pthread_t t2;
    ...
if (t1 == t2) {
    // do something
}
```

Instead must write:

```
if (pthread_equal(t1, t2)) {
    // do something
}
```

Thread ID, III

To get thread's own ID:

`pthread_t pthread_self()`

Much like `getpid(2)`

Thread, like process, starts not knowing its 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 *);
```

Aside: Void *

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

So

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

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

Aside: Example

```
void *function(void *arg) {
    argument_t *input;
    return_t *ret;

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

Thread Start

First thread of process starts executing `main()`

Later threads start with function passed as argument

Single "`void *`" argument can be used to pass arbitrary arguments — pack arguments into a `struct` then create a pointer to the `struct`

Termination

Two ways for thread to terminate:

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

Either way, a `void *` value is returned by terminated thread

Join, I

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

Similarly, one thread can wait for another to terminate:

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

(Note "`void **result`" – pointer to storage for `void *` return value)

Join, II

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

BUT: *there is no parent-child relationship among 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 thread should never return: `PTHREAD_CANCELED`

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

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

```
/*
 * POSIX 1003.1-2001, section 2.5.9.3:
 * ``The symbolic constant PTHREAD_CANCELED
 * expands to a constant expression of type (void *)
 * whose value matches no pointer
 * to an object in memory nor the value NULL.''
 */
#define PTHREAD_CANCELED        ((void *) 1)
```

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:

- Detached or joinable
- Size of stack
- Location of stack base
- Scheduling policy
- Custom vendor-specific attributes

Thread Attributes, II

Implementation may choose not to implement some attributes

If attribute is implemented, compile-time constant will be defined:

```
_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 all defaults:

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

Thread Stacks

Setting stack size or location 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 (e.g., by `malloc`)

Minimum guaranteed stack size given by `PTHREAD_STACK_MIN`

Problems Caused by Threads

1. Thread-safe libraries
2. Cancel-safe libraries
3. Adapting UNIX semantics
4. Synchronization

Thread-safe Libraries

For library to be "thread safe" means: any number of threads may be executing functions from this library simultaneously

Library functions must be *reentrant*:

- Function does not return pointer to static
- Function does not write to errno
- Function does not use globals OR function gets lock before accessing globals

Cancelation

A thread may be canceled:

```
int pthread_cancel(pthread_t target)
```

A thread has:

- Cancel state – enabled or disabled
- Cancel type — asynchronous or deferred

Deferred cancelation means: thread may be canceled only at certain "cancelation points" where implementation checks "should I kill this thread?"

Cancelation Points

4 cancelation points in Pthreads implementation:

1. `pthread_testcancel`
2. `pthread_join`
3. `pthread_cond_wait`
4. `pthread_cond_timed_wait`

POSIX states vendors *must* implement cancelation points in 23 specific library functions — roughly, those that may block

POSIX states vendors *may* implement cancelation points in approximately 50 other specific library functions

Cancel-safe Libraries

If thread is cancelable and "type" is asynchronous, thread will be killed immediately

Q: How to ensure that thread won't be half-done with some crucial operation (e.g., has locked a file)?

A: Can't

Therefore, set thread cancel state to deferred before any "dangerous" operation

Adapting UNIX Semantics

Process management

Signals — to which thread is a signal delivered?

Blocking & scheduling

Adapting UNIX Process Management

Does `fork` of N-thread process create another N-thread process or a 1-thread process?

Does `_exit` terminate just one thread or whole process?

What happens to threads when `exec` is called?

Fork

When thread calls `fork` ...

- New 1-thread process is created
- Thread is replica of thread in parent process that called `fork`
- Address space of child duplicates that of parent — including all state created by other threads in parent

Atfork

`pthread_atfork` function exists to help manage potential mess:

```
int pthread_atfork(void (*prepare)(void),
                   void (*parent)(void),
                   void (*child)(void));
```

`prepare` function called in parent before fork

`parent` function called in parent after fork

`child` function called in child after fork

Exit

Process terminates when any of these events occurs:

- Any thread calls `_exit(2)`
- Thread running `main` terminates
- Fatal signal is delivered

Exec

When `exec(2)` is called:

- All existing threads terminated
- New thread created to run `main` of new executable

Synchronization

Next week ...

Threads and Signals, I

POSIX added:

- Notion of per-thread signal mask
- Thread analogues of signal system calls: `pthread_kill(pthread_t, int)`, `pthread_sigmask`, etc.

Each thread can mask signals individually

How signal is handled depends on how signal was generated

Threads and Signals, II

If signal was generated by hardware or software exception (e.g., `SIGILL` or `SIGPIPE`)

...

then "effective target" of signal is thread that caused exception, so ...

signal is delivered to offending thread

Threads and Signals, III

If signal was generated by `pthread_kill` ...

then "effective target" of signal is specific thread, so ...

signal is delivered to targeted thread

Threads and Signals, IV

If signal was generated by external process

...

then "effective target" is whole process, so

...

signal is delivered to *arbitrary thread that does not have signal blocked*