

POSIX Threads

Unix Systems Programming

CSPP 51081

What is a POSIX Thread?

What is a Thread?

A *thread* is a single execution stream in a process which can be independently scheduled by the kernel and shares the same addressable space with any other thread.

Independence

Since threads are independently scheduled threads act concurrently with each other and may be executed in parallel in a multi-processor system. This means that each thread must have the following resources of its own:

- Program Counter
- Stack Space
- Register Set (i.e. space to store register values when not on the CPU)
- Priority (used for scheduling of CPU time)

Concurrency

Operations are *concurrent* if they may be arbitrarily interleaved so that they make progress independently.

- Threads can be scheduled to use the CPU in any order with any other thread. A thread may be removed from the CPU at any point of execution and replaced by another thread.
- Operations which cause one thread to suspend execution do not cause the execution other threads to be suspended.
- If the order that threads execute matter, or a thread must fully complete a task before other threads can execute, then the thread execution must be synchronized to coordinate action

Thread Addressable Memory

Any data whose address may be determined by a thread are accessible to all threads in the process.

- This includes static variables, automatic variables, and storage obtained via `malloc`.
- This *does not* mean that a thread cannot have private data, and we will see that POSIX includes a way for a thread to have data which only it can access. But it does mean that that the thread does not access this data through some addressable location

Benefits of Multithreaded Programs

- Utilizes program parallelism on a multiprocessor architecture for faster execution.
- Performance gains on single processor architectures by allowing processes to make computational progress even while waiting on slow I/O operations or blocking operations.
- Quicker responsiveness in interactive and real-time systems that must respond to asynchronous events.

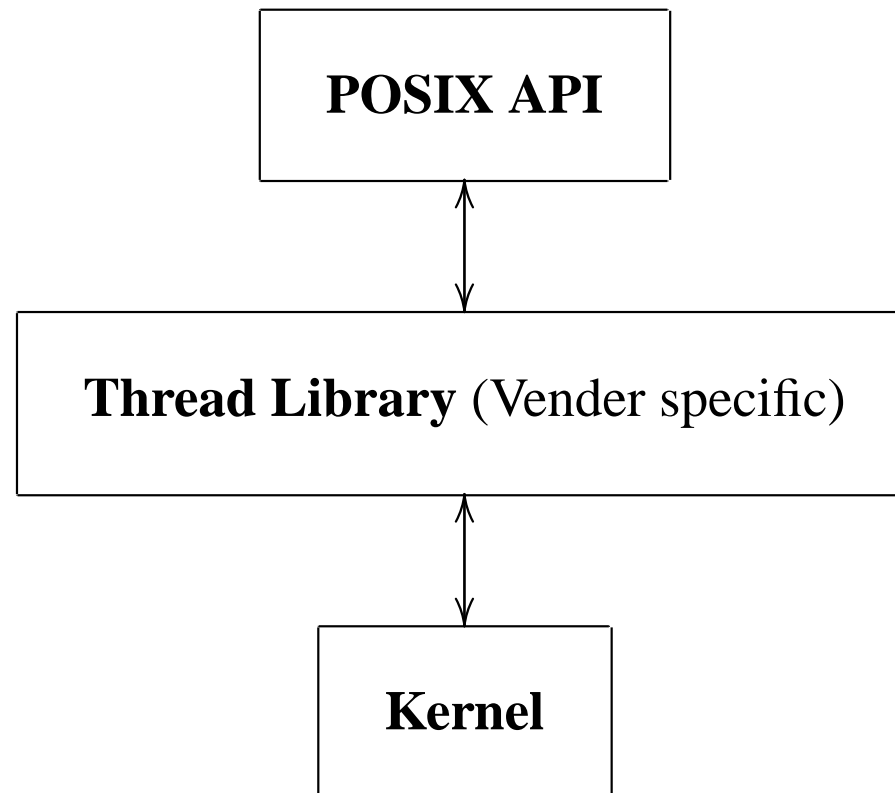
Problems of Multithreaded Programs

- Computational overhead of thread synchronization and scheduling overhead, which may be intolerable on programs with little parallelism.
- Greater programming discipline to plan and coordinate different execution sequences
- Bad code breaks in ways that can be very hard to locate and repair.

Implementing Threads

Thread Implementation: Layers of Abstraction

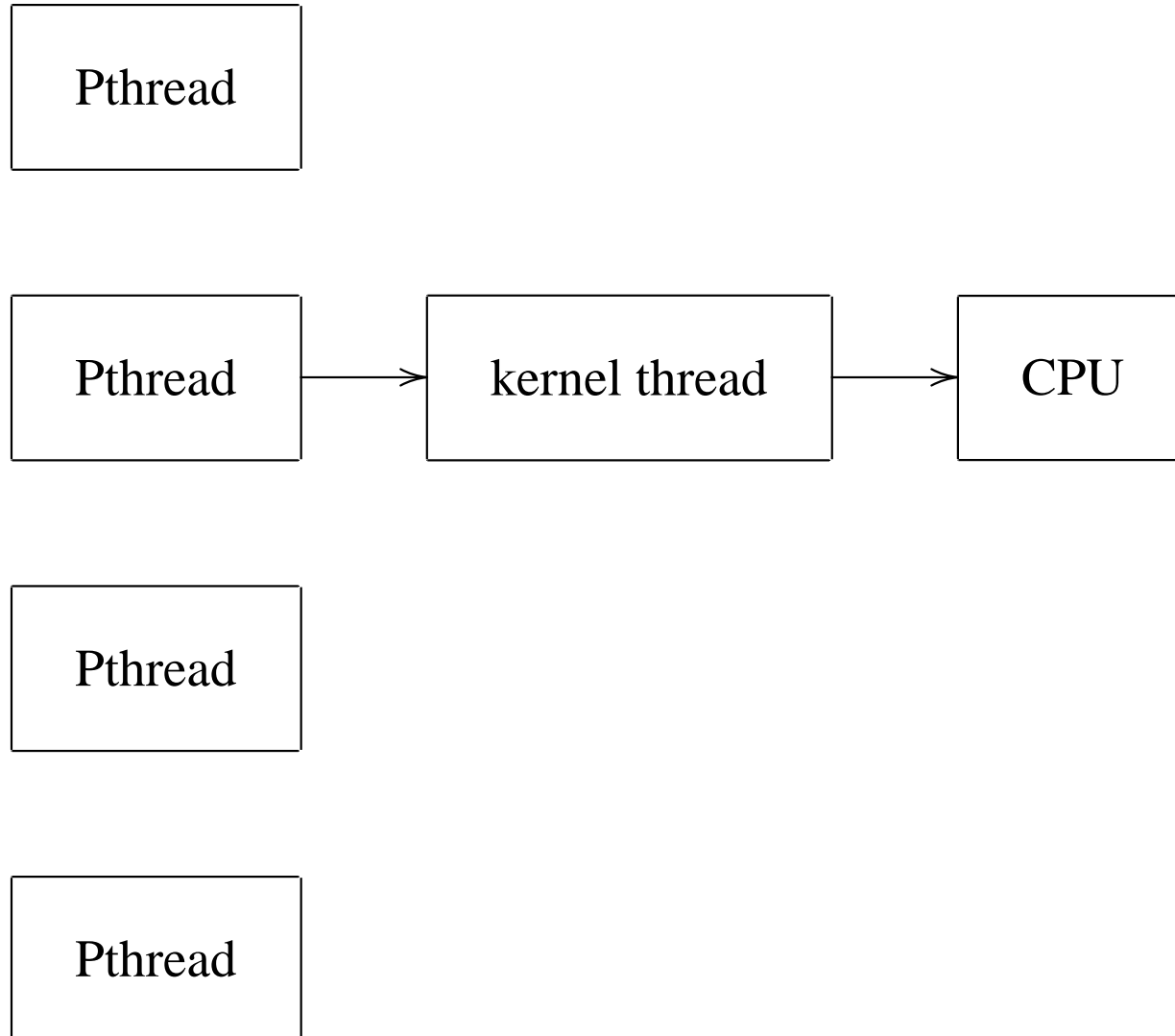
Implementing the POSIX API occurs in three layers of abstraction



Layers of Abstraction

- A *Pthread* is the POSIX specification for thread behavior, so is an abstraction, which the systems programmer sees and directly uses.
- A *kernel thread* is the entity actually created, scheduled and managed by the kernel as the context to execute a thread in a process. What it is and how it is managed depends on the operating system.
- The thread library may provide no more than a simple wrapper routine to a kernel system call to implement the POSIX specification; or it may provide full support for thread creation, scheduling and management, independently of the kernel. In this latter case, the entity created by the library to implement the POSIX specification is called a *user thread*.

Three models of thread implementation.

Many-to-one

Many-to-one: Advantages

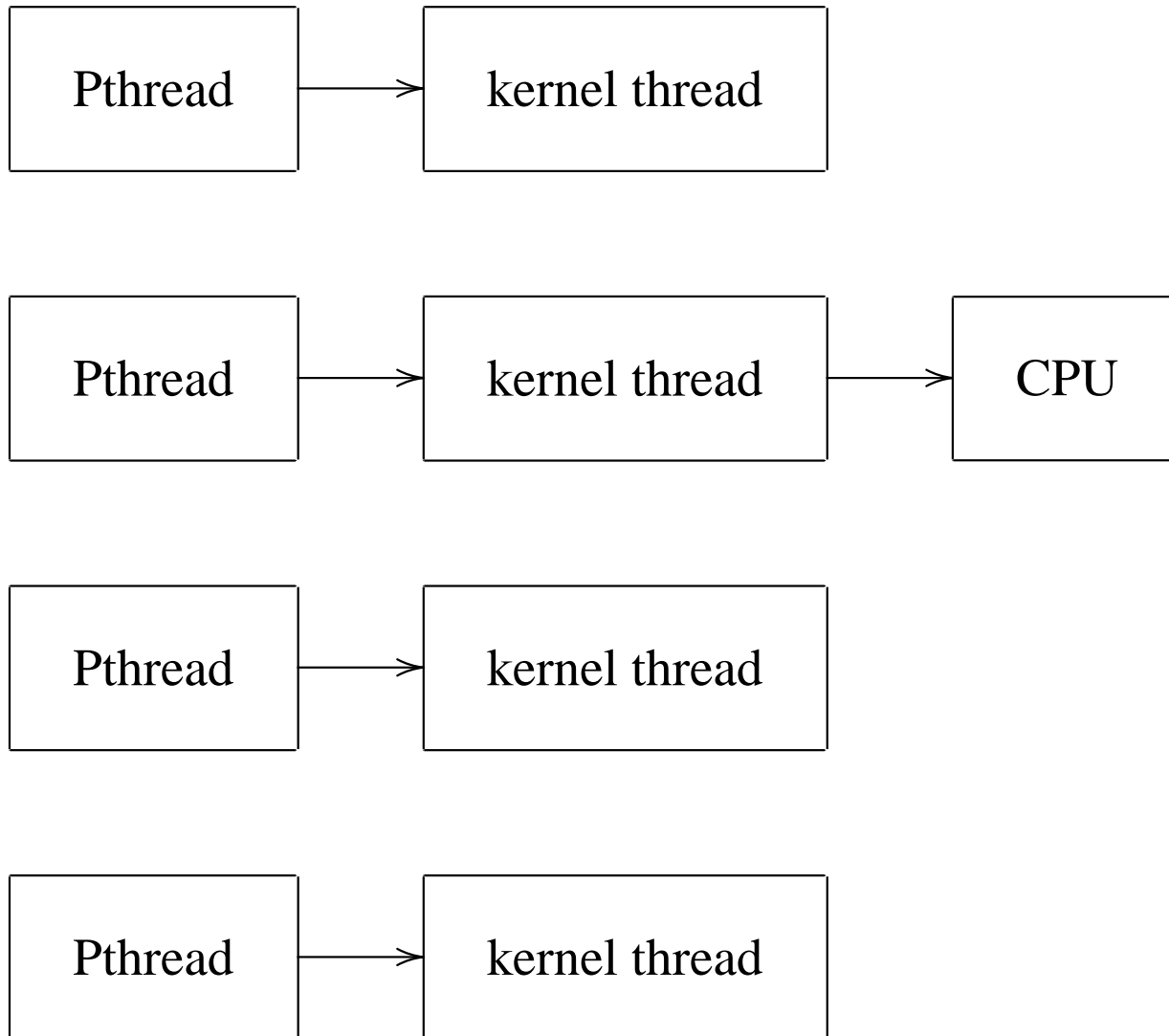
Pthreads are implemented as user threads by the runtime library.

- Most portable implementation since it requires no kernel support.
- Fast context switches between user threads because it is handled entirely in user space without the overhead of the kernel implementing a process-level context switch. This is especially valuable for processes which require much synchronization between threads using mutexes or condition variables.

Examples include Mach C-threads, BSD, Solaris UI-threads and DCE-threads.

Many-to-one: Disadvantages

- Cannot take advantage of multiprocessor architecture.
- Potentially long latency during system service blocking. The user thread library must replace blocking system calls with non-blocking versions, to prevent a blocking call from blocking all threads of the process. Some devices do not support non-blocking system calls.

One-to-one

One-to-one: Advantages

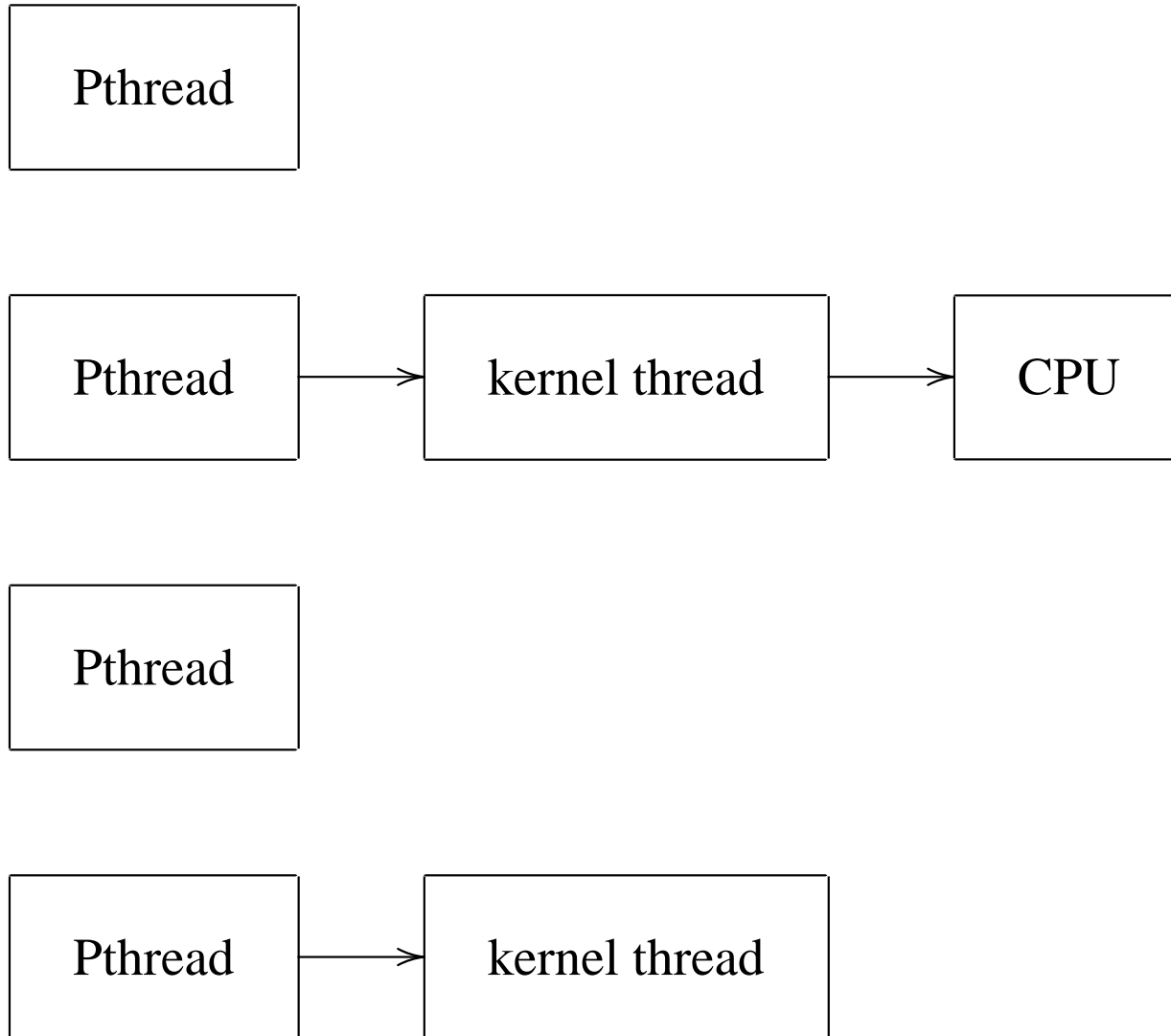
Pthreads are implemented as kernel threads through the kernel. Thread library may still be extensively involved in synchronizing threads, but the kernel handles thread creation and scheduling.

- Can take full advantage of multiprocessor architecture within a single process.
- No latency during system service blocking (any more than between processes.)

Examples include Windows operating system and Linux Threads. (Linux has taken the approach of making process creation fast through the `clone` system call. Essentially, the process data structure in the kernel does not contain data directly, but stores pointers to process data, making copying faster.)

One-to-one: Disadvantages

- Poor scaling when many threads are used, as each thread takes kernel resources from the system.
- Generally, slow thread context switches since these require the kernel to intervene. This means that programs which frequently require threads to block for synchronization will run slower than in the other models.

Many-to-few

Many-to-few: Advantages

Tries to merge the advantages of the one-to-one and many-to-one models. Requires extensive coordination between the user-level thread library and the kernel. They share scheduling responsibilities.

- Can take full advantage of multiprocessor architecture within a single process.
- Most context switches are in user mode, switching user threads without switching kernel threads.
- Scales well: a process does not need as many kernel threads to run
- Little latency during system service blocking
- Excels in most real-world applications since these have a mix of blocking I/O system calls, blocking synchronization and computation intensive operations.

Solaris, HP, AIX all use this model.

Many-to-few: Disadvantages

- More complicated implementation to coordinate user-level library and kernel.

Common Features of All Models

Regardless of the model, all threads share text, static and heap memory, and each thread has its own

- Thread ID
- Scheduling priority and policy
- `errno` value
- signal mask
- stack memory (although this is addressable by any other thread), register set and program counter
- Thread specific data (this is not addressable by other threads.)

Pthread API

Object-Oriented

The Pthread API is *object-oriented*. The API consists of ten *objects* (types in C) which are *opaque*, so that initializing, destroying, obtaining attributes and modifying instances of any object must be performed through one of sixty plus functions.

You must not make any assumptions about the implementation details of any type.

Pthread API Objects

Type	Description
<code>pthread_t</code>	thread identifier
<code>pthread_attr_t</code>	thread attribute
<code>pthread_cond_t</code>	condition variable
<code>pthread_condattr_t</code>	condition variable attribute
<code>pthread_key_t</code>	access key for thread-specific data
<code>pthread_mutex_t</code>	mutex
<code>pthread_mutexattr_t</code>	mutex attribute
<code>pthread_once_t</code>	one time initialization
<code>pthread_rwlock_t</code>	read-write lock
<code>pthread_rwlockattr_t</code>	read-write lock attribute

POSIX Thread API

- Each thread is *uniquely* identified by a thread ID of type.
- POSIX thread functions return 0 on success and a non-zero error code on failure.
- POSIX thread functions do not set `errno`, so you cannot use `perror` to report errors, but you can use `strerror` (although there are dangers, as we will see in thread safety.)
- POSIX thread functions are automatically restarted if interrupted by a signal.
- You can determine which functions modify which objects by dropping the trailing `_t` from the type. Ex. `pthread_mutex_init` initializes a `pthread_mutex_t` object; `pthread_create` creates (and starts running) a `pthread_t` object.

Pthread naming convention

Pthread functions consist of a prefix giving the object type the function modifies and a suffix giving the operation

- The prefix is determined by dropping `_t` from the object type.
- The suffix may be one of
 - `_init` to initialize object
 - `_destroy` to destroy object
 - `_getattr` get attribute value from the attribute object
 - `_setattr` set attribute value from the attribute object

Pthread Attributes

Pthread objects are designed to be *modular* and *portable*.

- Many traditional POSIX functions use flags to set attributes when initializing a new object. This means that to modify the attributes of all objects of a type across a program requires finding every initialization.
- Pthreads allow the programmer to centralize attribute creation to a single place in the code by treating the attributes themselves as objects.
- Many attributes produce implementation dependent behavior, and some may not be universally implemented.

Pthread Objects

Pthread objects are designed to have *fast* implementations. Pthread functions cut corners to ensure fast implementation. This has two consequences:

- Pthread functions are not *async-signal safe*. They must not be called in signal handlers (discussed in more detail later.)
- Pthread synchronization objects with default properties may be vulnerable to problems such as priority inversion (when a low priority thread locks a mutex required by a high priority thread) and recursive deadlock (behavior when a thread tries to relock a mutex.)

Pthread Object Initialization

- Pthreads provides `_init` operations for all objects (called `_creat` for `pthread_t`) which sets the attributes for the object created. Declaring a name of an object (ex. `pthread_t tid;`) only sets aside space, it does not create an object of this type.
- Reinitializing an object is dangerous and must be avoided. Pthreads provides two methods to avoid this:
 - Use `pthread_once_t` to ensure the initialization routine is performed at most once.
 - For default synchronization objects (mutexes, condition variables and read-write locks) a static initializer, `_INITIALIZER` is provided:

```
pthread_mutex_t  mtx = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t   cnd = PTHREAD_COND_INITIALIZER;  
pthread_rwlock_t rwl = PTHREAD_RWLOCK_INITIALIZER;
```

Compiling with Pthreads

- Use header `<pthread.h>` for all files using pthread routines. It is recommended in some implementations (ex. AIX) to place this header *first* since it defines macros for using thread-safe functions when these are available.
- When compiling you need to specify the *libpthread.a* library (on some systems this is included in the standard C library, so this is redundant):

```
gcc -lpthread
```

Thread Management

Thread Management v. Process Control

Threads are to functions what processes are to programs.

- Threads are identified by a process unique thread ID.
- When threads are created they begin executing a function, whose parameter is passed during creation. (Compare process creation with `fork` and `exec`, when an argument list is passed.)
- Threads can exit or be terminated by other threads. (Compare process `exit` and `kill`.)
- A thread can wait for another thread and collect its return value. (Compare process `wait`.)
- Threads have modifiable attributes like priority. (Compare process `nice`.)
- Threads have their own signal masks. (Compare process `sigprocmask`)

POSIX Thread Management Functions

Function	Description
<code>pthread_cancel</code>	terminate another thread
<code>pthread_create</code>	create a thread
<code>pthread_detach</code>	set thread to release resources
<code>pthread_equal</code>	test two thread IDs for equality
<code>pthread_exit</code>	exit a thread w/o exiting process
<code>pthread_kill</code>	send a signal to a thread
<code>pthread_join</code>	wait for a thread
<code>pthread_self</code>	find out own thread ID

Getting thread ID and comparing thread IDs

SYNOPSIS

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

```
pthread_t pthread_self(void);
```

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

Return:

0 on success non-zero error code on error

The only error for `pthread_equal` is `ESRCH` for an invalid thread ID.

You must use this function for comparing thread values.

pthread_create : create a thread

SYNOPSIS

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

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

Return:

0 on success non-zero error code on error

pthread_create: Possible Errors

error	cause
EAGAIN	The system lacked the necessary resources to create another thread, or the system-imposed limit on the total number of threads a process may have is exceeded. The maximum number of threads allowed to a process is given by <code>PTHREAD_THREADS_MAX</code>
EINVAL	<code>attr</code> parameter is invalid
EPERM	Caller does not have the appropriate permissions to set scheduling policy or parameters specified by <code>attr</code>

Usage of `pthread_create`

- The thread created by `pthread_create` is runnable without requiring a separate start code. It begins execution on the first line of the function routine provided in the third argument.
- The `thread` parameter points to the thread ID of the newly created thread.
- The `attr` parameter is usually set to `NULL` so the new thread has default attributes.
- The `routine` parameter is a function which has a single argument and returns a single value (both generic `void *`). To pass multiple values you will need to create a structure or array.
- The `arg` parameter is the argument to be passed to `routine`.

Example of Usage of `pthread_create`

```
void *routine(void *);

int         error;
int         arg[2]; //pass 2 int args to thread
pthread_t   tid;

//args given some value to be passed to thread
arg[0] = 0;
arg[1] = 1;
if (error = pthread_create(&tid, NULL, routine, &arg))
    fprintf(stderr, "Failed to create thread: %s\n",
            strerror(error));
```

A Dangerous Pitfall in `pthread_create`

Warning: the fourth parameter, `arg` is a pointer to data, and a common mistake is to pass this parameter and then overwrite it when calling a new thread. You must make sure a thread will no longer access this data before overwriting it with a new value to be passed to another thread.

Exiting a thread: Terminating the process

- Calling `exit` in any thread forces the process to terminate.
- Returning from `main` or running out of code in `main`
- Any signal whose default action is to kill the process and is uncaught kills the entire process, even if it is directed at a particular thread (using `pthread_kill`).
- Last thread in process calls `pthread_exit`, in which case the process return value is zero.
- Cancelling the main thread using `pthread_cancel`

If the main thread has no work to do it should block until all threads have completed (perhaps using `pthread_join`.)

Exiting a thread: w/o exiting process

The following assume the thread is not running main

- Thread call to `pthread_exit`
- Returning from function call (an implicit call to `pthread_exit`)
- Running out of code in the function
- Cancelling the thread using `pthread_cancel`

pthread_exit: exiting a thread

SYNOPSIS

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

```
void pthread_exit(void *valptr);
```

Return:

0 on success no error defined

Using `pthread_exit`

- A thread returns a value through `return` or `pthread_exit`
- The return value can be examined using `pthread_join`
- The return value is a pointer, so must point to data which persists after the thread exits. This means that the memory *must* be allocated in the heap through `malloc` (which is thread-safe) or a globally accessible variable.
- The return value must not be an automatic local variable.
- The return value should not be a static variable (since although it persists, if any other thread runs in the same function it will share the static variable)
- The return value is not passed to the parent process calling `wait`. If the last thread calls `pthread_exit` the return value to a waiting process is zero.

Terminating another thread

SYNOPSIS

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

```
int pthread_cancel(pthread_t thread);
```

```
void pthread_testcancel(void);
```

Return:

0 on success non-zero error value on error

The only defined error is ESRCH, for an invalid thread ID.

Terminating other threads

Calling `pthread_cancel` is a polite request that another thread shut itself down.

- The default attribute of a thread is to *enable deferred* cancellation. You can change this using `pthread_setcancelstate` (enable or disable) and `pthread_setcanceltype` (deferred or asynchronous.)
- Deferred cancellation means that a thread cancels only at specific points in its execution, called *cancellation points*. These can be specifically set by the thread using `pthread_testcancel` or checked by the Pthread library before a thread begins execution of certain slow and blocking operations (such as waiting on a mutex lock or an I/O operation.)
- A thread can set-up cancellation handlers using `pthread_cleanup_push` before any cancellation points. The thread can add any number of handlers (much like `atexit`.) To remove the last handler pushed, call `pthread_cleanup_pop`.
- Asynchronous cancellation immediately terminates another thread (if it has enabled cancellation) *wherever* it is in its execution sequence. This is very dangerous though and should be avoided.

Signaling another thread

SYNOPSIS

```
#include <pthread.h> #include <signal.h>
```

```
int pthread_kill(pthread_t thread, int signo);
```

Return:

0 on success non-zero error value on error

pthread_kill: Possible Errors

error	cause
ESRCH	No thread corresponds to specified ID
EINVAL	signo is an invalid signal number

Sending signals to other threads

Do not use signals to terminate other threads!!

- `pthread_kill` is used just like `kill`. Although `pthread_kill` sends a signal to a particular thread, the action of handling it may affect the entire process. For example, sending a `SIGKILL` will terminate the entire process, and `SIGSTOP` will stop the entire process. (Remember, neither signal can be caught.)
- To raise a signal (sending a signal to yourself), use

```
pthread_kill(pthread_self(), signo);
```

Joining and detaching a thread

SYNOPSIS

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

```
int pthread_join(pthread_t thread, void **valp);  
int pthread_detach(pthread_t thread);
```

Return:

0 on success non-zero error value on error

pthread_join, pthread_detach: Possible Errors

error	cause
ESRCH	No thread corresponds to specified ID
EINVAL	signo is an invalid signal number
EDEADLK	pthread_join: A deadlock detected or the value of the thread is the calling thread.

Joining and Detaching threads

- Deceased threads, like processes, tie-up system resources; they also count against the `PTHREAD_THREADS_MAX` for the process.
- A thread which calls `pthread_join` suspends until the specified thread terminates, and receives the location for a pointer to the return value in `valptr` (unless this is set to `NULL`).
- The `pthread_detach` function sets a thread's internal attribute to specify storage for the thread can be reclaimed when the thread exits.
- Detached threads do not report their return value.

Threading Issues

Thread Issues

Threads introduce new issues for old POSIX system calls.

- Fork/Exec
- Thread-safe functions
- Error handling
- Signals

Fork and multithread programs

Avoid `fork` in a multithread program unless you plan to call `exec` immediately afterwards.

- The calling thread exists in the child created by `fork` in the same state as in the parent.
- All other threads cease to exist.
- All other Pthread objects (mutex, condition variable, read-write lock) exist in the same state as in the parent. This means that if a mutex is locked in the parent it is locked in the child!!
- Additionally, some threads may be in the process of modifying data which does not complete in the child.
- Use `pthread_atfork` (similar to `atexit`) to gracefully enter the child.

Using `pthread_atfork`

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

- `prepare` is run before the fork: the creating thread obtains all mutex locks *in correct order* to avoid deadlocks.
- `child` is run after the fork in the child: release all mutex locks.
- `parent` is run after the fork in the parent: release all mutex locks.
- Calling `fork` in a multithreaded program is messy, unless you plan on following this call with `exec`.

Thread Safety

A function is *thread-safe* if it may be safely invoked concurrently by multiple threads. Functions which use static objects are not thread safe. Most functions are thread-safe, but the following which you have probably used this quarter are not thread-safe:

- `ctime`, `asctime`
- `getenv`, `putenv`
- `getopt`
- `gethostbyname`, `gethostbyaddr`, `getservbyname`,
`getservbyport`
- `rand`
- `strtok`, `strerror`

Pthreads has defined variants which are safe, which are designated by the suffix `_r`; but these are not always available.

Strategies for thread-safety

1. Wrap function in big mutex (see next example.) This is easiest, but can slow performance by serializing code.
2. Pass thread specific static data as an argument (ex. `strtok_r`). This can be very cumbersome and exposes the internal workings of the function (as with `rand_r` below or `malloc`.)
3. Use `pthread_key_t` so each thread has its own personal data which only it can access (recall, that standard library functions are implemented in user space, so each thread executes the function as part of its execution sequence).

Ensuring Thread-Safety: An example

A thread-safe version of rand which returns 0 if no error and non-zero error code if error (locking or unlocking mutex fails.) ranp points to an integer between 0 and RAND_MAX.

```
#include <pthread.h>
#include <stdlib.h>

int rand_r(int *ranp){
    static pthread_mutex_t = PTHREAD_MUTEX_INITIALIZER;
    int error;

    if (error = pthread_mutex_lock(&lock))
        return error;
    *ranp = rand();
    return pthread_mutex_unlock(&lock);
}
```

Error Reporting

- **Behind the illusion:** Up to now you have believed `errno` was a global variable of type `int`, but such a variable would be useless for a multi-threaded program. In actuality, `errno` is a macro defined in `<bits/errno.h>` which returns thread-specific information. Each thread has a private copy of `errno`.
- To report errors: Using `perror` for system calls is OK.
- To report errors: Use protect uses of `strerror` by a mutex, or use `strerror_r` which is a function I will provide (`strerror_r` is not part of the standard library.)

Signal Handling and Threads

All threads share the process signal handlers, but each thread has its own signal mask of the signals it blocks. There are three types of signals with different methods of delivery:

- **Asynchronous:** Signals which are not delivered at any predictable time nor associated with any particular thread (e.g. SIGINT.) These are delivered to some thread which has not blocked the signal.
- **Synchronous:** Signals which are generated at the same point in a thread's execution (e.g. SIGFPE, an arithmetic error such as dividing by zero.) These are delivered to the thread which caused the signal.
- **Directed:** Signals directed at a particular thread with `pthread_kill`. These are delivered to the thread.

Signal Handling and Threads

- All threads inherit the signal mask (those signals which are blocked) from the creating thread.
- You must set the signal mask using `pthread_sigmask`.
- Signal actions are process wide. You cannot have a handler for one thread and a different handler for another thread.
- A signal is delivered to one thread. If a second signal arrives while the first signal is being processed, the signal is delivered to another thread, if there is one which isn't blocking the signal. (For asynchronous signals only.)
- *It is highly recommended that you create the signal mask and set all signal actions in the main thread, once.*

Handling signals in MT programs

Problem: Signals are always delivered as long as there are threads which do not block them—even if another thread is executing a signal handler!!

- *It is a recommended strategy to dedicate a particular thread to handle all asynchronous signals.*
- Have the main thread block all incoming signals. Since all threads inherit the signal mask, all threads will block.
- Have a dedicated signal handling thread which calls `sigwait` (new to Pthreads and is preferable to `sigsuspend` in multithreaded programs.) This call will unblock signals for which there are handlers.
- Since `sigwait` returns when a signal handler is called, the thread should call `sigwait` in a while loop.

Signal handlers and Pthreads

Never call a Pthread function in a signal handler.

- A function is *async-signal safe* if it can safely be interrupted by a signal and resumed after the signal handler returns (even if the signal handler executes the function itself.) Functions which use internal static structures are not async-safe unless they block signals from interrupting them.
- No Pthread function is signal safe. The reason is that blocking signals is time consuming (it requires a kernel system call, so a context switch). Pthread functions are designed for speed, not safety.
- If you must use a Pthread function in response to a signal, use `semwait` and a dedicated thread to handle the response. (The thread will awaken after the signal handler exits and can implement the response.)