

# POSIX Threads and Synchronization

CSE 220: Systems Programming

Ethan Blanton, Carl Alphonse, & Eric Mikida

Department of Computer Science and Engineering

University at Buffalo



# POSIX Threads

The [POSIX threads API](#) adds [threading](#) to Unix.

You will also see this API called [Pthreads](#) or [pthreads](#).

Early Unix provided only the [process model](#) for concurrency.

POSIX threads [look like processes](#), but [share more resources](#).

Every POSIX thread starts with a [function](#).

# POSIX Synchronization

Pthreads also provides [synchronization mechanisms](#).

In fact, it provides a [rather rich set](#) of options!

- Mutexes
- Semaphores
- Condition variables
- Thread joining
- Memory barriers<sup>1</sup>

Only [semaphores](#) are covered in detail in CS:APP.

---

<sup>1</sup>We won't talk about these.

# Compilation with Pthreads

Pthreads may **require extra compilation options**.

On modern Linux, **use `-pthread` both when compiling and linking**.

On some other systems, other options may be required:

- Provide a different compiler or linker option (such as `-pthread`)
- Compile with some preprocessor define (e.g., `-DPTHREAD`, `-D_REENTRANT`)
- Link with a library (e.g., `-lpthread`)
- ...read the documentation!

# Thread Creation

Threads are created with the `pthread_create()` function:

```
#include <pthread.h>

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

The created thread will:

- **begin** at the given `start_function` **function argument**
- have the given data `arg` **passed in as an argument**

# Pthread Object Declarations

Threads (and other Pthread objects) are **declared as values**.

They are often **used as pointers**.

For example:

```
pthread_t thread;  
pthread_create(&thread, NULL, thread_function, NULL);
```

This allows them to be created **without dynamic allocation**.

# Thread Functions

The thread `start function` has the following signature:

```
void *(*start_function)(void *);
```

This is a `function` that:

- Accepts a single `void *` argument
- Returns `void *`

Example:

```
void *thread_main(void *arg) {  
    return NULL;  
}
```

# Thread Semantics

When `pthread_create()` is called, it:

- Creates a new **execution context**, including **stack**
- Creates a **concurrent flow** using that stack and context
- Causes the new flow to **invoke the provided function** and **passes the provided argument**

The separation of **thread start function** and its **argument** allows one function to **perform multiple tasks** based on its argument.

The new thread **appears to be scheduled independently**.

It can do **anything the original thread could**.



# Thread Attributes

The function `pthread_create()` accepts a [thread attribute object](#).

This object has type `pthread_attr_t`.

Passing `NULL` for this argument will use [default attributes](#).

Thread attributes include:

- Processor [affinity](#)
- The desired [scheduler](#) for the thread and its [configuration](#)
- The [detach state](#) of the new thread
- The thread's [stack location and size](#)

We will not use thread attributes this semester.

# Thread Termination

POSIX threads can terminate in several ways:

- The **application can exit**
- By calling `pthread_exit()`
- By **returning** from the thread start function
- It can be **canceled** by another thread using `pthread_cancel()`

# Joining

A thread can be **joined**, which is a **synchronous operation**.

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

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

**Joining** a thread:

- **blocks** the caller until the thread exits
- **retrieves** the thread's exit status

# Examples

- `counter.c` - mutexes protecting critical section
- `deadlock.c` - deadlock scenario
- `odds_evens.c` - condition variables
- `printer.c` - thread scheduling and joining

# POSIX Mutexes

POSIX mutexes are of type `pthread_mutex_t`.

They provide basic mutex functionality with several features:

- Optional recursive lock detection
- A try lock operation that will return immediately whether or not the mutex could be locked

It is an error to unlock a POSIX mutex on a different thread than the thread that locked it.

# Mutex Initialization

POSIX mutexes have [static](#) and [dynamic](#) initializers:

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

```
pthread_mutex_t fastmutex =  
    PTHREAD_MUTEX_INITIALIZER;
```

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
    const pthread_mutexattr_t *mutexattr);
```

In [older POSIX specifications](#), the static initializer could be used [only for compile-time initializers](#).

The [dynamic initializer](#) accepts [attributes](#) to configure the mutex. (Pass NULL to get default behavior.)

# Mutex Operations

A mutex can be **locked** or **unlocked**:

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

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_trylock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

The **lock** and **unlock** functions operate exactly as expected.

`pthread_mutex_trylock()` will **always return immediately**.

- If the mutex is **already locked**, it will return EBUSY.
- If the mutex is **unlocked**, it will lock it and return 0.

# Destroying Mutexes

When you are finished with a mutex, you should **destroy it**.

On Linux, **destroying a mutex is essentially no-op**.

However, other platforms **may associate resources** with a mutex.

Destroying the mutex allows those resources to be **released**.

Destroying a **locked mutex is an error**.

Destroying a **mutex being waited upon<sup>2</sup> is an error**.

---

<sup>2</sup>More on this later...



# Default Mutex Behaviors

The default mutex **may not allow recursive locks**.

The following code could **deadlock** (and **will** on Linux!):

```
void deadlock() {  
    pthread_mutex_t mutex =  
        PTHREAD_MUTEX_INITIALIZER;  
    pthread_mutex_lock(&mutex);  
    pthread_mutex_lock(&mutex);  
}
```

Mutexes can be initialized with a **recursive attribute**.

Recursive mutexes maintain a **lock count**, and the above would simply require **unlocking twice**.

# Condition Variables

POSIX **condition variables** work **in conjunction with mutexes**.

A thread **must hold a mutex** to wait on a condition variable.

Waiting on a condition variable **atomically**:

- **Unlocks** the mutex
- Puts the thread **to sleep** until the condition is signaled

A thread can signal **one** or **all** threads sleeping on a condition variable.

# Creating a Condition Variable

Condition variables are created **like mutexes**:

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

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;  
int pthread_cond_init(pthread_cond_t *cond,  
    pthread_condattr_t *cond_attr);
```

The Linux implementation of Pthreads **recognizes no condition variable attributes**.

# Waiting on Condition Variables

A thread can **wait** on a condition variable.

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

```
int pthread_cond_wait(pthread_cond_t *cond,  
pthread_mutex_t *mutex);
```

Note that there is **an associated mutex**.

The mutex should **protect the condition state**.

As previously discussed, threads can **spuriously wake**.

# Waiting Example

```
extern pthread_mutex_t lock;
extern pthread_cond_t cond;
extern bool done;

void *block_until_done(void *ignored) {
    pthread_mutex_lock(&lock);
    while (!done) {
        pthread_cond_wait(&cond, &lock);
    }
    pthread_mutex_unlock(&lock);
    return ignored;
}
```

# Signaling Condition Variables

Condition variables can signal:

- **one** waiting thread
- **all** waiting threads

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

```
int pthread_cond_signal(pthread_cond_t *cond);  
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Signaling a variable if **no threads are waiting** does nothing.

The **mutex protecting shared state** should be used appropriately!

# Signaling Example

```
extern pthread_mutex_t lock;
extern pthread_cond_t cond;
extern bool done;

void signal_done() {
    pthread_mutex_lock(&lock);
    done = true;
    pthread_mutex_unlock(&lock);
    pthread_cond_signal(&cond);
}
```

## Putting it Together

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;  
bool done;
```

```
int main(int argc, char *argv[]) {  
    pthread_t t;  
  
    pthread_create(&t, NULL, block_until_done, NULL);  
    usleep(100000);  
    signal_done();  
    pthread_join(t, NULL);  
  
    return 0;  
}
```



# Destroying Condition Variables

Like mutexes

- Condition variables **should be destroyed**
- Destroying condition variables **does nothing on Linux**

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

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Destroying a condition variable **with waiting threads is an error.**

# POSIX Semaphores

POSIX semaphores can operate between either threads or processes.

They provide counting semaphore semantics.

They obsolete System V semaphores, which you may also see.

POSIX semaphores:

- Do not begin with `pthread_`
- Are not found in `pthread.h`

# POSIX Semaphore Creation

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared,
             unsigned int value);
```

There **is no static initializer** for POSIX semaphores.

If **pshared is true**:

- The semaphore **can be used between processes**
- **Must be located in shared memory** for this to work

The given value **is its initial count**.

# POSIX Semaphore Manipulation

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
```

The wait operation corresponds to Dijkstra's P(), and post to V().

sem\_trywait() is like pthread\_mutex\_trylock():

- It will return immediately **even if it cannot decrement the semaphore**
- If it succeeds **it returns zero**
- If it does not, **it returns EAGAIN**

# Summary

- The **POSIX threads** (pthreads) API provides a **thread abstraction** on Unix
- POSIX provides many **synchronization primitives**:
  - Mutexes
  - Semaphores
  - Condition variables
  - Thread joining
- CS:APP covers semaphores in detail

# References I

## Required Readings

- [1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Chapters 26, 27. Arpaci-Dusseau Books. URL: <https://pages.cs.wisc.edu/~remzi/OSTEP/>.

## Optional Readings

- [2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Chapters 30, 31. Arpaci-Dusseau Books. URL: <https://pages.cs.wisc.edu/~remzi/OSTEP/>.
- [3] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 12: 12.3, 12.5-12.7. Pearson, 2016.

# References II

- [4] IEEE and The Open Group. *The Open Group Base Specifications Issue 7*. 2017. URL: <http://pubs.opengroup.org/onlinepubs/9699919799/>.
- [5] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., 1996.

# License

Copyright 2018–2024 Ethan Blanton, All Rights Reserved.

Copyright 2024–2024 Eric Mikida, All Rights Reserved.

Copyright 2022–2024 Carl Alphonse, All Rights Reserved.

Copyright 2019 Karthik Dantu, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see <https://www.cse.buffalo.edu/~eblanton/>.