

# Programming with POSIX Threads

Parallel Programming for Engineers

2026-05-18



---

Prof. Dr. Michael Kuhn

michael.kuhn@ovgu.de

Parallel Computing and I/O

Institute for Intelligent Cooperating Systems

Faculty of Computer Science

Otto von Guericke University Magdeburg

<https://parcio.ovgu.de>

## Programming with POSIX Threads

Review

Introduction

Basics

Thread Management

Synchronization

Summary

- How many threads run in this example?
  1. Twelve (nproc output)
  2. Four
  3. Three
  4. Two
  5. One

```
1  int main(void) {
2      omp_set_num_threads(3);
3
4      #pragma omp parallel
5          ↪ num_threads(4) if(1)
6      printf("Hello world from "
7             "thread %02d/%02d.\n",
8             omp_get_thread_num(),
9             omp_get_num_threads());
10
11     return 0;
12 }
```

- How many threads run in this example?
  1. Twelve (nproc output)
  2. Four ✓
  3. Three
  4. Two
  5. One

```
1  int main(void) {
2      omp_set_num_threads(3);
3
4      #pragma omp parallel
5          ↪ num_threads(4) if(1)
6      printf("Hello world from "
7             "thread %02d/%02d.\n",
8             omp_get_thread_num(),
9             omp_get_num_threads());
10
11     return 0;
12 }
```

- What happens in this example?
  1. The same as with `parallel for`
  2. Compiler exits with an error
  3. Both threads calculate the whole loop
  4. Race condition

```
1  int main(void) {
2      int i;
3
4      omp_set_num_threads(2);
5
6      #pragma omp parallel
7      for (i = 0; i < 10; i++) {
8          printf("i=%d, id=%d\n",
9              i, omp_get_thread_num()
10             );
11     }
12
13     return 0;
14 }
```

- What happens in this example?
  1. The same as with `parallel for`
  2. Compiler exits with an error
  3. Both threads calculate the whole loop
  4. Race condition ✓

```
1  int main(void) {
2      int i;
3
4      omp_set_num_threads(2);
5
6      #pragma omp parallel
7      for (i = 0; i < 10; i++) {
8          printf("i=%d, id=%d\n",
9              i, omp_get_thread_num()
10             );
11     }
12
13     return 0;
14 }
```

- What is the fastest synchronization construct for incrementing a variable?
  1. `critical`
  2. `atomic`
  3. `reduction`
  4. `omp_lock_t`

- What is the fastest synchronization construct for incrementing a variable?
  1. critical
  2. atomic ✓
  3. reduction ✓
  4. omp\_lock\_t

## Programming with POSIX Threads

Review

**Introduction**

Basics

Thread Management

Synchronization

Summary

- OpenMP provides a convenient interface for thread programming
  - Support depends on the compiler and is tuned towards parallel applications
- POSIX Threads are a low-level approach for threads
  - Allows covering more use cases than high-level approaches
  - Might be available on more systems, providing improved portability
- Fine-grained control over threads allows performance tuning
  - For instance, it is possible to control when threads are started and terminated

- Threads can be used to cover a wide range of use cases
  - Reducing latency for servers by preempting long requests
  - Improve throughput by overlapping system calls for I/O and communication
  - Handle asynchronous events by spawning threads to handle input etc.
  - Real-time applications via high priority threads
  - Separation of concerns in applications
- OpenMP is tuned for numerical computations
  - Sections and tasks provide a more generic interface

- Modern computers always feature multiple cores
  - Applications should be designed with concurrency and parallelism in mind
  - Non-numerical applications can also benefit from threads
- Modern operating systems can deal with threads
  - Threads are mapped to available cores according to scheduling policy
- We have to take care that used libraries are thread-safe
  - Thread-safe functions from libc are listed in [Linux man-pages project, 2024]

- Thread-safety means that multiple threads can call a function at the same time
  - There is also reentrancy, which is different from thread-safety
  - Reentrancy is mainly used in the context of signal handling and interrupts
- We are mainly interested in thread-safety for normal applications
  - Reentrancy becomes important if code can be executed in kernel mode
- Own code and used libraries have to be thread-safe
  - Otherwise, it is necessary to manually take care of locking etc.

## Programming with POSIX Threads

Review

Introduction

**Basics**

Thread Management

Synchronization

Summary

- Threads are available using different interfaces
  - OpenMP covers many numerical use cases
  - `clone` allows starting threads but is very complex
    - Requires in-depth Linux knowledge and is not portable
- `fork` can be used to spawn multiple processes for arbitrary applications
  - Requires using shared memory objects to exchange data
  - Overhead is too high for many use cases
- POSIX Threads provide a standardized interface for thread programming

- Vendors shipped their own proprietary implementations of threads
  - Bad for portability, custom operating systems are common in HPC
  - POSIX Threads are standardized in POSIX 1003.1c (1995)
- POSIX Threads are available on many systems, not only Linux
  - Native support on Linux, BSD, Android, macOS etc.
  - Windows support via mapping to existing Windows API
- Other thread implementations are often very similar
  - See C11 threads, which cover a reduced feature set

- POSIX Threads cover multiple aspects
  1. Thread management and miscellaneous functionality
  2. Mutexes (mutual exclusion via locks)
  3. Condition variables (communication between threads)
  4. Synchronization (barriers, read/write locks etc.)
- Semaphores are part of a different standard (POSIX 1003.1b, 1993)
- Implementations might still differ in certain details
  - Maximum number of threads, allowed stack size etc.

- There have been two major POSIX Threads implementations

## 1. LinuxThreads

- Original implementation that is unsupported since glibc 2.4
- Threads do not share the same process ID but have individual PIDs

## 2. Native POSIX Threads Library (NPTL)

- Current implementation that is closer to POSIX compliance
  - Still not fully compliant: Threads do not share a common nice value
- Better performance with large numbers of threads
- Requires newer features from Linux 2.6 (CLONE\_THREAD)
  - Threads in a process share the same process ID

- Threads allow overlapping work
  - For instance, computation with I/O or communication
- Threads have their own control flow
  - Separate stack, registers, scheduling, signals and thread-local storage
- Operating systems use threads extensively
  - More than 150 kernel threads on a typical Linux system

- Threads can be mapped to schedulable tasks in various ways
- 1:1 mapping
  - Each thread created by the developer corresponds to one task in the kernel
  - Used on Linux, macOS, iOS, Solaris, various BSDs etc.
- n:1 mapping
  - Several user-level threads map to one kernel task
  - Allows switching between threads without context switches
  - Does not offer true parallelism due to limited scheduling
- m:n mapping
  - Maps several user-level threads to several kernel tasks
  - Requires coordination between threading library and operating system

- POSIX Threads allow covering a wider range of use cases than OpenMP
- Applications have to be designed for threading from the start
  - There is no support for incremental parallelization
  - Refactoring existing applications is more complicated
- There is no special compiler support for POSIX Threads
  - Developers have to manage threads explicitly
  - No automatic distribution of computation via work sharing directives

- POSIX Threads functions and data structures all start with `pthread_`
  1. Thread management: `pthread_` and `pthread_attr_`
  2. Mutexes: `pthread_mutex_` and `pthread_mutexattr_`
  3. Condition variables: `pthread_cond_` and `pthread_condattr_`
  4. Synchronization: `pthread_barrier_` etc.
  5. Locking: `pthread_rwlock_`, `pthread_spin_` etc.
  6. Thread-local storage: `pthread_key_`
- Applications have to be adapted
  - Header `pthread.h` has to be included
  - Compiler flag `-pthread` has to be used (automatically links with `libpthread`)
- Some features require preprocessor macros to be set
  - For instance, barriers require `_POSIX_C_SOURCE` with a value of at least `200112L`

## Programming with POSIX Threads

Review

Introduction

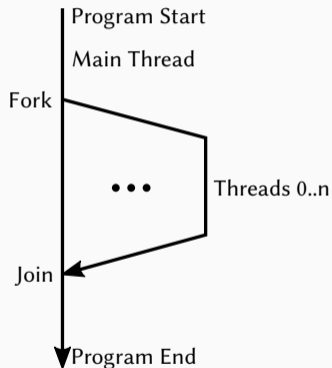
Basics

**Thread Management**

Synchronization

Summary

- When starting a process, there is one main thread
  - Starting new threads forks the control flow
  - Terminating them joins it again
  - Process ends when main thread terminates
- Fork and join have to be performed manually
  - OpenMP used to take care of this for us
  - We have to manage overhead ourselves now



- `pthread_create`
  - Thread identifier (opaque)
  - Attributes (scheduling etc.)
  - Thread routine (function)
  - Argument (function argument)
- Creates a new thread
  - Maximum number set by `ulimit`
  - No distinction between processes and threads in Linux
  - Maximum is typically not a problem nowadays (125,835 per process)
  - Threads can create other threads

```
1  int main(void) {
2      pthread_t threads[10];
3
4      for (uint64_t i = 0; i < 10; i++) {
5          pthread_create(&threads[i],
6                        NULL, thread_func,
7                        (void*)i);
8      }
9
10     for (uint64_t i = 0; i < 10; i++) {
11         pthread_join(threads[i], NULL);
12     }
13
14     return 0;
15 }
```

- `pthread_join`
  - Thread identifier
  - Return value
  - Cleans up resources
    - Otherwise, zombies are created
- Main thread has to wait for others
  - `pthread_join` synchronizes
  - `pthread_exit` waits for threads

```
1  int main(void) {
2      pthread_t threads[10];
3
4      for (uint64_t i = 0; i < 10; i++) {
5          pthread_create(&threads[i],
6                        NULL, thread_func,
7                        (void*)i);
8      }
9
10     for (uint64_t i = 0; i < 10; i++) {
11         pthread_join(threads[i], NULL);
12     }
13
14     return 0;
15 }
```

- Thread termination can vary
  - `pthread_exit` with return value
  - Return value from routine
    - Implicit `pthread_exit` for all non-main threads
  - `pthread_cancel` to terminate
  - Any thread calls `exit`
  - Main thread returns from `main`

```
1 void* thread_func(void* data) {
2     uint64_t id = (uint64_t)data;
3
4     sleep(1);
5     printf("Hello world from "
6           "thread %ld.\n", id);
7
8     return NULL;
9 }
```

- Threads can be influenced using attributes
  - Detach state
    - Determines whether threads can be joined to get return value
  - Stack size (and more)
    - Stack size is implementation-specific and not standardized (usually 2 MiB)
  - Scheduling and priority
    - Priority of specific threads can be adapted to provide real-time behavior
  - Affinity (not portable)
    - Thread migrations could cause performance degradation due to cache invalidation

- `pthread_attr_t`
  - Opaque data structure
  - Has to be initialized and destroyed
  - Set attributes using specific functions
- Detach state determines whether joining is possible
  - Detached cannot return value
  - Resources will be cleaned up automatically after termination
  - Can be set via `pthread_detach`
  - Joining synchronizes threads

```
1  int main(void) {
2      pthread_t threads[10];
3      pthread_attr_t attr[1];
4
5      pthread_attr_init(&attr);
6      pthread_attr_setdetachstate(&attr,
7                                  PTHREAD_CREATE_DETACHED);
8
9      for (uint64_t i = 0; i < 10; i++) {
10         pthread_create(&threads[i],
11                       &attr, thread_func,
12                       (void*)i);
13     }
```

- pthread\_attr\_t
  - Opaque data structure
  - Has to be initialized and destroyed
  - Set attributes using specific functions
- Detach state determines whether joining is possible
  - Detached cannot return value
  - Resources will be cleaned up automatically after termination
  - Can be set via pthread\_detach
  - Joining synchronizes threads

```
1   for (uint64_t i = 0; i < 10; i++) {  
2       pthread_join(threads[i], NULL);  
3   }  
4  
5   pthread_attr_destroy(attr);  
6  
7   return 0;  
8 }
```

- How does the previous example behave?
  1. All threads print a hello world message
  2. No output is produced and process terminates immediately
  3. Application crashes in pthread\_join
  4. Compiler produces an error message

```
1 pthread_attr_setdetachstate(attr,  
2     PTHREAD_CREATE_DETACHED);  
3  
4 for (uint64_t i = 0; i < 10; i++) {  
5     pthread_create(&threads[i],  
6         attr, thread_func,  
7         (void*)i);  
8 }  
9  
10 for (uint64_t i = 0; i < 10; i++) {  
11     pthread_join(threads[i], NULL);  
12 }  
13  
14 return 0;
```

- How does the previous example behave?
  1. All threads print a hello world message
  2. No output is produced and process terminates immediately ✓
  3. Application crashes in pthread\_join
  4. Compiler produces an error message

```
1 pthread_attr_setdetachstate(attr,  
2     PTHREAD_CREATE_DETACHED);  
3  
4 for (uint64_t i = 0; i < 10; i++) {  
5     pthread_create(&threads[i],  
6         attr, thread_func,  
7         (void*)i);  
8 }  
9  
10 for (uint64_t i = 0; i < 10; i++) {  
11     pthread_join(threads[i], NULL);  
12 }  
13  
14 return 0;
```

- Scheduling can be affected in a variety of ways
  - Need to be set via attributes when thread is created
- Contention scope
  - Defines which other threads the thread competes against
  - System: Compete with all other threads on the system
  - Process: Compete with other threads within same process
    - Unspecified how they compete system-wide
  - Linux supports only system-wide contention scope

- Scheduling policy
  - Supports a subset of Linux's scheduling policies
  - FIFO: First-in, first-out (run until blocked, preempted or thread yields)
  - RR: Round-robin (FIFO with maximum time slice)
  - Other: Default time-sharing policy
- Processor affinity
  - Allows setting which processors/cores a thread can run on
  - Non-portable extension but important for performance

- `pthread_self`
  - Returns the current thread's ID
- ID is an opaque data structure, additional functions are needed
  - `pthread_equal` can be used to compare two IDs
- Necessary for some functionality
  - Not easily possible to pass ID via `pthread_create`

```
1 void* thread_func(void* data) {
2     (void)data;
3
4     sleep(1);
5     printf("Hello world from "
6           "thread %p.\n",
7           (void*)pthread_self());
8
9     return NULL;
10 }
```

- `pthread_self`
  - Returns the current thread's ID
- ID is an opaque data structure, additional functions are needed
  - `pthread_equal` can be used to compare two IDs
- Necessary for some functionality
  - Not easily possible to pass ID via `pthread_create`

```
1  int main(void) {
2      pthread_t thread;
3
4      pthread_create(&thread, NULL,
5                    thread_func, NULL);
6      printf("Started thread %p.\n",
7            (void*)thread);
8
9      pthread_join(thread, NULL);
10
11     return 0;
12 }
```

- `pthread_self`
  - Returns the current thread's ID
- ID is an opaque data structure, additional functions are needed
  - `pthread_equal` can be used to compare two IDs
- Necessary for some functionality
  - Not easily possible to pass ID via `pthread_create`

```
Started thread 0x7fd846781640.  
Hello world from thread 0x7fd846781640.
```

- `pthread_cancel`
  - Sends cancellation request to thread
- Cancelability state and type
  - State can be enabled or disabled
  - Type is asynchronous or deferred
    - Asynchronous: At any time
    - Deferred: At cancellation points
- Deferred cancellation by default
  - Only specific functions are cancellation points
  - `printf` may be a cancellation point

```
1 void* thread_func(void* data) {
2     pthread_t thread = pthread_self();
3
4     (void)data;
5
6     pthread_cancel(thread);
7     printf("Hello world from "
8         "thread %p.\n",
9         (void*)thread);
10    printf("Bye world from "
11        "thread %p.\n",
12        (void*)thread);
13
14    return NULL;
15 }
```

- `pthread_cancel`
  - Sends cancellation request to thread
- Cancellability state and type
  - State can be enabled or disabled
  - Type is asynchronous or deferred
    - Asynchronous: At any time
    - Deferred: At cancellation points
- Deferred cancellation by default
  - Only specific functions are cancellation points
  - `printf` may be a cancellation point

```
Started thread 0x7f05b12dc640.  
Hello world from thread 0x7f05b12dc640.
```

- What happens with `pthread_exit` instead of return for a detached thread?
  1. Main thread waits for termination
  2. The same as with return
  3. The whole process is terminated

```
1 void* thread_func(void* data) {
2     uint64_t id = (uint64_t)data;
3
4     sleep(1);
5     printf("Hello world from "
6           "thread %ld.\n", id);
7
8     return NULL;
9 }
```

- What happens with `pthread_exit` instead of return for a detached thread?
  1. Main thread waits for termination
  2. The same as with return ✓
  3. The whole process is terminated

```
1 void* thread_func(void* data) {
2     uint64_t id = (uint64_t)data;
3
4     sleep(1);
5     printf("Hello world from "
6           "thread %ld.\n", id);
7
8     return NULL;
9 }
```

- Need ways to initialize data structures
  - Static variable for serial applications
- pthread\_once
  - Control structure tracks initialization
  - Calls given routine exactly once
- Safely initialize multi-threaded applications and libraries

```
1  static pthread_once_t once =
2      PTHREAD_ONCE_INIT;
3
4  void once_func(void) {
5      printf("Hello once.\n");
6  }
7
8  void* thread_func(void* data) {
9      (void) data;
10     pthread_once(&once, once_func);
11     return NULL;
12 }
```

## Programming with POSIX Threads

Review

Introduction

Basics

Thread Management

**Synchronization**

Summary

- `pthread_barrier_init`
  - Initialized for a number of threads
  - Attributes to share across processes

```
1  int main(void) {
2      pthread_t threads[10];
3      pthread_barrier_init(&barrier,
4                          NULL, 10);
5      for (uint64_t i = 0; i < 10; i++) {
6          pthread_create(&threads[i],
7                        NULL, thread_func,
8                        (void*)i);
9      }
10     for (uint64_t i = 0; i < 10; i++) {
11         pthread_join(threads[i], NULL);
12     }
13     pthread_barrier_destroy(&barrier);
14     return 0;
15 }
```

- `pthread_barrier_init`
  - Initialized for a number of threads
  - Attributes to share across processes
- `pthread_barrier_wait`
  - All threads have to enter barrier
  - One thread gets special return value
  - Others do not wait for serial thread

```
1  static pthread_barrier_t barrier[1];
2
3  void* thread_func(void* data) {
4      (void)data;
5
6      printf("Hello world.\n");
7
8      if (pthread_barrier_wait(barrier) ==
9          PTHREAD_BARRIER_SERIAL_THREAD)
10         printf("I am the one.\n");
11
12         printf("Bye world.\n");
13
14         return NULL;
15     }
```

- `pthread_barrier_init`
  - Initialized for a number of threads
  - Attributes to share across processes
- `pthread_barrier_wait`
  - All threads have to enter barrier
  - One thread gets special return value
  - Others do not wait for serial thread

```
Hello world.  
...  
Hello world.  
Bye world.  
...  
I am the one.  
...  
Bye world.
```

- pthread\_mutex\_t
  - Implements mutual exclusion
  - Similar to a critical region in OpenMP
  - Can be initialized statically
- Allows setting attributes
  - Only via pthread\_mutex\_init
- Locks block by default
  - trylock returns immediately

```
1 static int counter = 0;
2 static pthread_mutex_t mutex =
3     PTHREAD_MUTEX_INITIALIZER;
4
5 void* thread_func(void* data) {
6     (void) data;
7     for (int i = 0; i < 1000; i++) {
8         pthread_mutex_lock(&mutex);
9         counter++;
10        pthread_mutex_unlock(&mutex);
11    }
12    return NULL;
13 }
```

- Mutex attributes allow changing behavior
  - Priority ceiling: Maximum priority, only for FIFO scheduling
  - Protocol: Priority changes if blocking more important threads
  - Process-shared: Whether mutexes can be shared across processes
  - Robustness: Behavior if owner terminates without unlocking
  - Type: Normal, error-checking or recursive

- Condition variables allow implementing efficient condition checking
  - Usually, a thread would have to check the condition regularly (spinlock)
- Condition variables support waiting and signaling
  - Thread can sleep until another thread signals that condition is met
  - Allows synchronization based on the value of data

- `pthread_cond_t`
  - Condition variables require a mutex
  - Can have attributes via `pthread_cond_init`

```
1  static int counter = 0;
2  static pthread_cond_t cond =
3      PTHREAD_COND_INITIALIZER;
4  static pthread_mutex_t mutex =
5      PTHREAD_MUTEX_INITIALIZER;
```

- `pthread_cond_t`
  - Condition variables require a mutex
  - Can have attributes via `pthread_cond_init`
- `pthread_cond_wait`
  1. Unlocks mutex
  2. Sleeps until condition is met
  3. Locks mutex

```
1 static int counter = 0;
2 static pthread_cond_t cond =
3     PTHREAD_COND_INITIALIZER;
4 static pthread_mutex_t mutex =
5     PTHREAD_MUTEX_INITIALIZER;
```

- `pthread_cond_t`
  - Condition variables require a mutex
  - Can have attributes via `pthread_cond_init`
- `pthread_cond_wait`
  1. Unlocks mutex
  2. Sleeps until condition is met
  3. Locks mutex
- `pthread_cond_signal`
  - Signals condition is met
  - Wakes up at least one thread

```
1  static int counter = 0;
2  static pthread_cond_t cond =
3      PTHREAD_COND_INITIALIZER;
4  static pthread_mutex_t mutex =
5      PTHREAD_MUTEX_INITIALIZER;
```

```
void* producer(void* data) {
    (void)data;
    while (1) {
        pthread_mutex_lock(&mutex);
        while (counter >= 10)
            pthread_cond_wait(
                &cond, &mutex);

        counter++;
        printf("p=%d\n", counter);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}
```

```
void* consumer(void* data) {
    (void)data;
    while (1) {
        pthread_mutex_lock(&mutex);
        while (counter == 0)
            pthread_cond_wait(
                &cond, &mutex);

        counter--;
        printf("c=%d\n", counter);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}
```

```
void* producer(void* data) {
    (void)data;
    while (1) {
        pthread_mutex_lock(&mutex);
        while (counter >= 10)
            pthread_cond_wait(
                &cond, &mutex);

        counter++;
        printf("p=%d\n", counter);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}
```

```
p=1
p=2
...
p=9
p=10
```

```
p=1
p=2
...
p=9
p=10
c=9
c=8
...
c=1
c=0
```

```
void* consumer(void* data) {
    (void)data;
    while (1) {
        pthread_mutex_lock(&mutex);
        while (counter == 0)
            pthread_cond_wait(
                &cond, &mutex);

        counter--;
        printf("c=%d\n", counter);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}
```

```
void* producer(void* data) {
    (void)data;
    while (1) {
        pthread_mutex_lock(&mutex);
        while (counter >= 10)
            pthread_cond_wait(
                &cond, &mutex);

        counter++;
        printf("p=%d\n", counter);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}
```

```
p=1
p=2
...
p=9
p=10
c=9
c=8
...
c=1
c=0
p=1
p=2
p=3
...
```

- `pthread_cond_wait` performs steps atomically
- Condition variables do not store signals
  - If no thread is waiting when signaling, nothing happens
- Signaling should be performed with a locked mutex
- Attributes can influence behavior
  - Clock: Which clock should be used for `pthread_cond_timedwait`
  - Process-shared: Whether condition variables can be used across processes

- pthread\_key\_t
  - Thread-specific data, also known as thread-local storage
  - Optional destructor
- Calls destructor on thread termination
  - For instance, per-thread hash tables

```
1  int main(void) {
2      pthread_t threads[10];
3
4      pthread_key_create(&key, NULL);
5      for (uint64_t i = 0; i < 10; i++) {
6          pthread_create(&threads[i],
7                        NULL, thread_func,
8                        (void*)(i + 1));
9      }
10     for (uint64_t i = 0; i < 10; i++) {
11         pthread_join(threads[i], NULL);
12     }
13     pthread_key_delete(key);
14     return 0;
15 }
```

- `pthread_key_t`
  - Thread-specific data, also known as thread-local storage
  - Optional destructor
- Calls destructor on thread termination
  - For instance, per-thread hash tables
- `pthread_setspecific`
  - Initializes thread-specific data
- `pthread_getspecific`
  - Returns thread-specific data

```
1  static pthread_key_t key;
2
3  void* thread_func(void* data) {
4      void* mykey;
5      pthread_setspecific(key, data);
6      mykey = pthread_getspecific(key);
7      printf("key=%p, mykey=%p\n",
8             (void*)&key, mykey);
9
10     return NULL;
11 }
```

- `pthread_key_t`
  - Thread-specific data, also known as thread-local storage
  - Optional destructor
- Calls destructor on thread termination
  - For instance, per-thread hash tables
- `pthread_setspecific`
  - Initializes thread-specific data
- `pthread_getspecific`
  - Returns thread-specific data

```
key=0x404058 , mykey=0x1
key=0x404058 , mykey=0x2
key=0x404058 , mykey=0x6
key=0x404058 , mykey=0x3
key=0x404058 , mykey=0x4
key=0x404058 , mykey=0x5
key=0x404058 , mykey=0x8
key=0x404058 , mykey=0x7
key=0x404058 , mykey=0x9
key=0x404058 , mykey=0xa
```

## Programming with POSIX Threads

Review

Introduction

Basics

Thread Management

Synchronization

Summary

- POSIX Threads are a standard for thread programming
  - Available on most major operating systems
- Includes thread management, mutexes, condition variables and synchronization
  - Most behavior can be influenced using attributes
- Allows fine-grained control and tuning of threads
  - Requires manual thread management and work sharing
- Covers a wider range of use cases than OpenMP
  - Threads can be used for structuring applications, not only parallelism

## References

[Barney, 2025] Barney, B. (2025). **POSIX Threads Programming.**

<https://hpc-tutorials.llnl.gov/posix/>.

[Linux man-pages project, 2024] Linux man-pages project (2024). **pthread(7).**

<https://man7.org/linux/man-pages/man7/pthreads.7.html>.