Programming with POSIX Threads

Parallel Programming

2024-11-25

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

Outline

- What is the difference between kernel mode and user mode?
    1. Kernel mode can only execute instructions from the kernel binary
    2. Kernel mode has unrestricted access to the hardware
    3. Kernel mode is slower than user mode due to overhead

- Why should system calls be avoided in HPC applications?
  1. System calls are a legacy approach
  2. Interrupts are better suited for HPC applications
  3. System calls can cause the application to lose their processor allocation
  4. System calls are slow due to management overhead

- How are thread-safety and reentrancy related?
  1. Both describe the same concept
  2. Thread-safety implies reentrancy
  3. Reentrancy implies thread-safety
  4. Neither implies the other

- Which function allows starting threads?
    1. `fork`
    2. `exec`
    3. `clone`
    4. `open`

Outline

Motivation

- 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

Motivation…

- 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

Motivation…

- 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

- 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.

Outline

Overview

- 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

Overview…

- 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

Overview…

- 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.

Implementations

- 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

Importance

- 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

Importance…

- 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

Benefits and Drawbacks

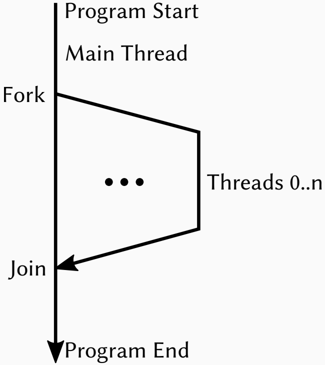
- POSIX Threads allow covering a wider range of use cases than OpenMP
- Applications have to be designed for threading from the start
  - There is no support for incremental parallelization
  - Refactoring existing applications is more complicated
- There is no special compiler support for POSIX Threads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

Usage

- 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

Outline

- 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



Creating Threads

- 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

```
int main(void) {
    pthread_t threads[10];

    for (uint64_t i = 0; i < 10; i++) {
        pthread_create(&threads[i],
            NULL, thread_func,
            (void*)i);
    }

    for (uint64_t i = 0; i < 10; i++) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

- 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

```
int main(void) {
    pthread_t threads[10];

    for (uint64_t i = 0; i < 10; i++) {
        pthread_create(&threads[i],
            NULL, thread_func,
            (void*)i);
    }

    for (uint64_t i = 0; i < 10; i++) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

Creating Threads

- 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

```c
void* thread_func(void* data) {
    uint64_t id = (uint64_t)data;

    sleep(1);
    printf("Hello world from "
        "thread %ld.\n", id);

    return NULL;
}
```

Attributes

- 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

```
int main(void) {
    pthread_t threads[10];
    pthread_attr_t attr[1];

    pthread_attr_init(attr);
    pthread_attr_setdetachstate(attr,
        PTHREAD_CREATE_DETACHED);

    for (uint64_t i = 0; i < 10; i++) {
        pthread_create(&threads[i],
            attr, thread_func,
            (void*)i);
    }
```

Attributes...

- 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

```
    for (uint64_t i = 0; i < 10; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_attr_destroy(attr);

    return 0;
}
```

Quiz

- 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

```c
pthread_attr_setdetachstate(attr,
    PTHREAD_CREATE_DETACHED);

for (uint64_t i = 0; i < 10; i++) {
    pthread_create(&threads[i],
        attr, thread_func,
        (void*)i);
}

for (uint64_t i = 0; i < 10; i++) {
    pthread_join(threads[i], NULL);
}

return 0;
```

Scheduling

- 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…

- 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

Current Thread

- 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

```
void* thread_func(void* data) {
    (void)data;

    sleep(1);
    printf("Hello world from "
        "thread %p.\n",
        (void*)pthread_self());

    return NULL;
}
```

Current Thread

- 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

```c
int main(void) {
    pthread_t thread;

    pthread_create(&thread, NULL,
        thread_func, NULL);
    printf("Started thread %p.\n",
        (void*)thread);

    pthread_join(thread, NULL);

    return 0;
}
```

Current Thread

- `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.
```

Cancellation

- 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

```
void* thread_func(void* data) {
    pthread_t thread = pthread_self();

    (void)data;

    pthread_cancel(thread);
    printf("Hello world from "
        "thread %p.\n",
        (void*)thread);
    printf("Bye world from "
        "thread %p.\n",
        (void*)thread);

    return NULL;
}
```

Cancellation

- `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

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

Quiz

- 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

```
void* thread_func(void* data) {
    uint64_t id = (uint64_t)data;

    sleep(1);
    printf("Hello world from "
        "thread %ld.\n", id);

    return NULL;
}
```

Initialization

- 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

```c
static pthread_once_t once =
    PTHREAD_ONCE_INIT;

void once_func(void) {
    printf("Hello once.\n");
}

void* thread_func(void* data) {
    (void)data;
    pthread_once(&once, once_func);
    return NULL;
}
```

Outline

Barrier

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

```c
int main(void) {
    pthread_t threads[10];
    pthread_barrier_init(barrier,
        NULL, 10);
    for (uint64_t i = 0; i < 10; i++) {
        pthread_create(&threads[i],
            NULL, thread_func,
            (void*)i);
    }
    for (uint64_t i = 0; i < 10; i++) {
        pthread_join(threads[i], NULL);
    }
    pthread_barrier_destroy(barrier);
    return 0;
}
```

- 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

```c
static pthread_barrier_t barrier[1];

void* thread_func(void* data) {
    (void)data;

    printf("Hello world.\n");

    if (pthread_barrier_wait(barrier) ==
        PTHREAD_BARRIER_SERIAL_THREAD)
        printf("I am the one.\n");

    printf("Bye world.\n");

    return NULL;
}
```

Barrier

- `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.
```

Mutex

- 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

```c
static int counter = 0;
static pthread_mutex_t mutex =
    PTHREAD_MUTEX_INITIALIZER;

void* thread_func(void* data) {
    (void)data;
    for (int i = 0; i < 1000; i++) {
        pthread_mutex_lock(&mutex);
        counter++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

Mutex…

- 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

- 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`

```
static int counter = 0;
static pthread_cond_t cond =
    PTHREAD_COND_INITIALIZER;
static pthread_mutex_t mutex =
    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

```c
static int counter = 0;
static pthread_cond_t cond =
    PTHREAD_COND_INITIALIZER;
static pthread_mutex_t mutex =
    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

```c
static int counter = 0;
static pthread_cond_t cond =
    PTHREAD_COND_INITIALIZER;
static pthread_mutex_t mutex =
    PTHREAD_MUTEX_INITIALIZER;
```

```c
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);
    }
}
```

```c
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);
    }
}
```

Condition Variables...

```
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
```

Condition Variables...

```
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);
    }
}
```

Condition Variables...

```
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
...
```

Condition Variables…

- `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

Thread-Local Storage

- 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

```
int main(void) {
    pthread_t threads[10];

    pthread_key_create(&key, NULL);
    for (uint64_t i = 0; i < 10; i++) {
        pthread_create(&threads[i],
            NULL, thread_func,
            (void*)(i + 1));
    }
    for (uint64_t i = 0; i < 10; i++) {
        pthread_join(threads[i], NULL);
    }
    pthread_key_delete(key);
    return 0;
}
```

Thread-Local Storage

- 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

```c
static pthread_key_t key;

void* thread_func(void* data) {
    void* mykey;
    pthread_setspecific(key, data);
    mykey = pthread_getspecific(key);
    printf("key=%p, mykey=%p\n",
        (void*)&key, mykey);

    return NULL;
}
```

Thread-Local Storage

- `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
```

Outline

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, 2024] Barney, B. (2024). **POSIX Threads Programming.** https://hpc-tutorials.llnl.gov/posix/.

[Linux man-pages project, 2024] Linux man-pages project (2024). **pthreads(7).** https://man7.org/linux/man-pages/man7/pthreads.7.html.