

Part IV

Other Systems: III

Pthreads: A Brief Review

An algorithm must be seen to be believed.

1

The POSIX Standard: 1/2

- **POSIX (P**ortable **O**perating **S**ystem **I**nterfaces) is a family of standards for maintaining compatibility between operating systems.
- **POSIX is a Unix-like operating system environment and is currently available on Unix/Linux, Windows, OS/2 and DOS).**

The POSIX Standard: 2/2

- **Pthreads (POSIX Threads) is a POSIX standard for threads.**
- **The standard, POSIX.1c thread extension, defines thread creation and manipulation.**
- **This standard defines thread management, mutexes, conditions, read/write locks, barriers, etc.**
- **Except for the monitors, all features are available in Pthreads.**

Thread Creation

- Always includes the `pthread.h` header file.

```
int pthread_create(  
    pthread_t          *tid,  
    const pthread_attr_t *attr,  
    void               *(*start)(void *),  
    void               *arg);
```

- `pthread_create()` creates a thread and runs function `start()` with argument list `arg`.
- `attr` specifies optional creation attributes.
- The ID of the newly created thread is returned with `tid`.
- Non-zero return value means creation failure.

Thread Join

- Use `pthread_join()` to join with a thread.
- The following waits for `thread` to complete, and returns `thread`'s exit value if `value_ptr` is not `NULL`. Use `NULL` if you don't use exit value.
- Join failed if `pthread_join()` returns a non-zero value.

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

Thread Exit

- Use `pthread_exit()` to terminate a thread and return the value `value_ptr` to any joining thread.
- Exit failed if `pthread_exit()` returns a non-zero value.
- Use `NULL` for `value_ptr` if you don't use exit value.

```
int pthread_exit(  
    pthread_t  thread,  
    void      *value_ptr) ;
```

Mutex: 1/2

- A mutex has a type `pthread_mutex_t`.
- Mutexes initially are unlocked.
- Only the owner can unlock a mutex.
- Since mutexes cannot be copied, use pointers.
- Use `pthread_mutex_destroy()` to destroy a mutex. Make sure no thread is blocked inside.

```
pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
int pthread_mutex_init(  
    pthread_mutex_t      *mutex,  
    pthread_mutexattr_t  *attr);  
int pthread_mutex_destroy(  
    pthread_mutex_t      *mutex);
```

Mutex: 2/2

- If `pthread_mutex_trylock()` returns `EBUSY`, the lock is already locked. Otherwise, the calling thread becomes the owner of this lock.
- With `pthread_mutexattr_settype()`, the type of a mutex can be set to allow recursive locking or report deadlock if the owner locks again.

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


Condition Variables: 1/2

- Conditions in Pthreads are usually used with a mutex to enforce mutual exclusion.

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

```
int pthread_cond_init(  
    pthread_cond_t          *cond,  
    const pthread_condattr_t *attr) ;  
int pthread_cond_destroy(  
    pthread_cond_t          *cond) ;  
int pthread_cond_wait(  
    pthread_cond_t          *cond,  
    pthread_mutex_t         *mutex) ;  
int pthread_cond_signal(  
    pthread_cond_t          *cond) ;  
int pthread_cond_broadcast(  
    pthread_cond_t          *cond) ;
```

Condition Variables: 2/2

- `pthread_cond_wait()` and `pthread_cond_signal()` are the `wait()` and `signal()` methods in **ThreadMentor**, and are `wait()` and `notify()` in Java.
- `pthread_cond_signal()` uses Mesa type and the released thread must recheck the condition.

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

Simulating a Mesa Monitor: 1/2

- Use a mutex for protecting the monitor.
- Lock and unlock this mutex upon entering and exiting the monitor.
- When a thread calls a condition wait, it relinquishes the monitor mutex. Once blocked, the monitor mutex becomes available to other threads.
- The released thread (from a condition wait) becomes the new owner of the monitor mutex.

Simulating a Mesa Monitor: 2/2

```
pthread_mutex_t MonitorLock = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;
```

monitor procedure

```
pthread_mutex_lock(&MonitorLock); // enter the monitor  
    // other statements  
    while (condition is not met)    // this is a Mesa type  
        pthread_cond_wait(&cond, &MonitorLock);  
    // other statements  
pthread_mutex_unlock(&MonitorLock); // exit monitor
```

monitor procedure

```
pthread_mutex_lock(&MonitorLock); // enter the monitor  
    // other statements  
    // cause condition to happen  
    pthread_cond_signal(&cond);  
    // other statements  
pthread_mutex_unlock(&MonitorLock); // exit monitor12
```

Simulating a Hoare Monitor

- **Simulating a Hoare type monitor requires the use of general semaphores.**
- **The Pthreads standard does not have semaphores. Instead, POSIX.1b standard has the Unix semaphores.**
- **With POSIX.1b semaphores, it is easy to simulate a Hoare type monitor. Many OS textbooks discuss such a simulation. Also see our reading lists for such a solution.**

Languages vs. Libraries: 1/2

- Libraries are extension to a sequential language.
- Programmers may try various approaches that fit his/her needs. Programs can be deployed without requiring any changes in the tools (e.g., compiler).
- Libraries may not be well-defined and completely portable. Some features may be difficult to define and/or implement (e.g., Hoare type monitors).
- Programs may be difficult to understand because API function calls can scatter everywhere and sometimes cryptic.

Languages vs. Libraries: 2/2

- With the language-based approach, the intent of the programmer is easier to express and understand, both by other programmers and by program analysis tools.
- Languages usually require the standardization of new constructs and perhaps new keywords.
- Language features are fixed. Each language may only support one or a few concurrent programming models, and may not be very flexible.

The End