Pthreads

Overview A process can contain multiple threads, where each thread independently executes the same program, while sharing global memory, initialized and uninitialized data, and heap segments. Stacks and some thread-specific data are not shared between threads. *Advantages of using threads instead of processes* include:

- It's easier to share information between threads. We don't need complex interprocess communication primitives.
- Process creation with `fork()` is expensive, while thread creation is based on `clone()`, which is more efficient since we don't need to duplicate shared data such as page tables, pages of memory, etc.

Disadvantages of threads over processes include:

- We need to ensure that the functions we call are **thread-safe** or are called in a thread-safe manner.
- A bug in one thread can damage all the thread in the process since they share the same address space, etc.
- Each thread is competing for use of the finite virtual address space of the host process. For example, in x86-32, 4GB address space can be shared by potentially huge number of threads.
- Dealing with signals requires careful design. Usually, avoiding signals in threads is desirable.
- In multithreaded applications, all threads run the *same* program, though at different program points.

Information shared between threads

- process ID, parent process ID, process group ID, session ID
- **page table, pages of memory**
- open file descriptors and file system-related information
- controlling terminal, signal disposition, etc.

Information not shared between threads

- thread ID, **stack**, **thread-specific data**
- signal mask, alternate signal stack (`sigaltstack()`)
- the `errno` variable
- realtime scheduling policy and priority, etc.

Pthreads datatypes There are several essential datatypes used in pthreads: `pthread_t` (thread identifier), `pthread_mutex_t` (mutex), `pthread_mutexattr_t` (mutex attributes object), `pthread_cond_t` (condition variable), `pthread_condattr_t` (condition variable attributes object), `pthread_key_t` (key for thread-specific data), `pthread_once_t` (one-time initialization control context), `pthread_attr_t` (thread attributes object).

Thread creation

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

Thread termination Note that `retval` should be allocated on heap not stack since after thread termination, its thread stack is no longer available.

```
int pthread_exit(void *retval);
```

Thread ID On Linux, `pthread_t` happens to be defined as `unsigned long`, however, in general, it can be any type such as `struct`. So, when handling thread IDs, we should not assume anything about the `pthread_t` type.

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

Joining terminated threads The `pthread_join()` function waits for the thread identified with `thr` to terminate. If `retval` is a non-NULL pointer, then it receives a copy of the terminated thread's return value. Only non-detached threads are joinable.

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

Detaching a thread By default, a thread is *joinable*, meaning that another thread can obtain its return status using `pthread_join()`. If we don't care about the thread's return status, we can simply tell the system to automatically clean up and remove the thread when it terminates by *detaching* them.

```
int pthread_detach(pthread_t thr);
```

A thread can detach itself using the following call.

```
pthread_detach(pthread_self());
```

Thread attributes Thread attributes include information such as the location and size of the thread's stack, thread's scheduling policy and priority, and whether the thread is joinable or detached. The following is an example of setting attributes.

```
pthread_t thr;
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
  PTHREAD_CREATE_DETACHED);
pthread_create(&thr, &attr, foo, (void *) 1;
pthread_attr_destroy(&attr);
```

Thread synchronization using a mutex A *mutex* prevents multiple threads from accessing a shared variable at the same time.

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

Behavior of mutexes can be adjusted by using `pthread_mutexattr_t`. An example which uses mutex for synchronization is given in a *producer-consumer* code below.

```
int avail = 0;  // shared variable

/* producer */
pthread_mutex_lock(&mtx);
avail++;
pthread_mutex_unlock(&mtx);

/* consumer */
for (;;) {
  pthread_mutex_lock(&mtx);
  while (avail > 0) {
    avail--; // also consume the unit
  }
  pthread_mutex_unlock(&mtx);
}
```

Signaling change of state using condition variables A *condition variable* allows one thread to inform other threads about changes in the state of a shared variable (or other shared resource) and allows the other threads to wait (block) for such notification.

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

When multiple threads are blocked on a condition variable in `pthread_cond_wait`, broadcasting wakes up all such threads.

```
int avail = 0; // shared variable

/* producer */
pthread_mutex_lock(&mtx);
avail++;
pthread_mutex_unlock(&mtx);
pthread_cond_signal(&cond);

/* consumer */
pthread_mutex_lock(&mtx);
while (avail == 0)
  pthread_cond_wait(&cond, &mtx);
avail--; // also consume the unit
pthread_mutex_unlock(&mtx);
```

Thread safety and reentrancy A function is said to be **thread-safe** if it can safely be invoked by multiple threads at the same time. To make a thread-safe function, there is one common method:

> *Whenever it accesses a shared global variable, use mutex around the access. However, too large critical section causes* serialization *of threads, resulting in loss of concurrency.*

Also, be careful when using a non-thread-safe C library function (e.g. `strtok()`).

A **reentrant** function achieves thread safety without the use of mutexes. It does this by avoiding the use of *global and static variables*. Any information that must be returned to the caller, or maintained between calls to the function, is stored in buffers allocated by the caller.

One-time initialization In multithreaded application, we want some initialization to occur only once regardless of how many threads are created.

```
int pthread_once(pthread_once_t *once_ctrl,
                 void (*init)(void));
```

Thread-specific data Thread-specific data is a technique for making an existing function *thread-safe* without changing its interface. This allows a function to maintain a *separate copy of a variable for each thread* that calls the function, which is *persistent* between thread's invocations of the function.

```
int pthread_key_create(pthread_key_t *key,
                       void (*destructor)(void *));
int pthread_setspecific(pthread_key_t key,
                        const void *Value);
int pthread_getspecific(pthread_key_t key);
```

Thread-local storage Like thread-specific data, thread-local storage provides persistent per-thread storage, but in a more simpler way. To create a thread-local variable, we simply include `__thread` specifier in the declaration of a *global or static* variable.

```
static __thread buf[MAX_LEN];
```

Thread cancellation

```
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int stat, int *oldstat);
int pthread_setcanceltype(int type, int *oldtype);
```

Thread implementation models In **M:1 implementations (user-level threads)**, all of the details of thread creation, scheduling, and synchronization are handled entirely with the process by a user-space threading library. This model is fast but the kernel doesn't know anything about the threads. So, any system call in one thread can block the entire threads.

In **1:1 implementations (kernel-level threads)**, each thread maps onto a separate kernel scheduling entity (KSE). LinuxThreads and NPTRL employ this model.

In **M:N implementations (two-level model)**, it aims to combine above to models.