CSC / CPE 357

Systems Programming

Chapter 11 in Advanced Programming
in the UNIX Environment

Threads

- Typical UNIX process do only one thing at a time
  - One "thread of control"

- With multiple threads of control (often just: **threads**) a single process can do more than one thing at a time; each thread can handle a separate task

- We will discuss the UNIX thread standard known as "pthreads" or "POSIX threads"
  - POSIX.1-2001 Standard

Threads

- Handle asynchronous events by assigning a separate thread to process each event
- Threads within a process automatically have access to the same memory address space and file descriptors
- Improved response time by using multiple threads to separate handling of user input/output from the other parts of the program

Multithread vs Multiprocessor Multicore

- A program can be simplified using threads regardless of the number of processors
  - Number of processors doesn't affect program structure.

- Any activities that block (wait) can still see improvements in response time and throughput when using threads on a single processor
  - Some threads might be able to run while others are blocked.

Thread ID / Data

- A thread consists of the information necessary to represent an execution context within a process:
  - Thread ID that identifies the thread within a process
  - Set of register values
  - Stack
  - Scheduling priority and policy
  - Signal mask
  - Errno variable

- Everything within a process is sharable among the threads in a process:
  - Text of the executable program
  - Global and heap memory
  - Stacks
  - File descriptors

Thread Creation

```
#include <pthread.h>

int pthread_create(pthread_t *restrict tidp,
                   const pthread_attr_t *restrict attr,
                   void *(*start_rtn)(void *), void *restrict arg);
```

Returns: 0 if OK, error number on failure

To install manual pages for pthread_ functions:

```
sudo apt-get install manpages-posix manpages-posix-dev
```

Thread Example

pthread_intro.c

Compile with:

```
gcc -pthread pthread_intro.c
```

(Adds support for multithreading with the pthreads library)

Thread Example

`pthread_intro.c`

Two notes:

1. `sleep()` in the main thread. Without this, the main thread might exit, terminating the entire process before the new thread gets a chance to run
2. New thread obtains its thread ID by calling `pthread_self`. Main thread stores this ID in variable `ntid`, but the new thread can't safely use it. New thread could run before the main thread returns from `pthread_create` (seeing the uninitialized contents of `ntid` instead of the thread ID)

Thread Termination

If any thread within a process calls exit, _Exit, or _exit, then the entire process terminates.

A single thread can exit in three ways (without terminating the entire process.

1. `return` from the start routine (return value is the thread's exit code)
2. A thread can be canceled by another thread in the same process
3. A thread can call `pthread_exit`

```
#include <pthread.h>

void pthread_exit(void *rval_ptr);
```

Return Value from a Thread

```
#include <pthread.h>

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

Returns: 0 if OK, error number on failure

- `pthread_join` allows us to wait for a specified thread
- The calling thread will block until specified thread completes or is canceled.
- `rval_ptr` will contain the return code
- If the thread was canceled (`pthread_cancel` function) `rval_ptr` is set to `PTHREAD_CANCELED`

Thread Join / Exit Code

`pthread_exit.c`

- Pointer passed to `pthread_create` and `pthread_exit` can be used to pass more than a single value.
  - For example, pointer can be used to pass the address of a structure containing more complex information.
- Caution: the memory used for the structure must remain valid when the caller has completed.
  - If the structure was allocated on the caller's stack, for example, the memory contents might have changed by the time the structure is used.

`pthread_exit2.c`

Process vs Thread Functions

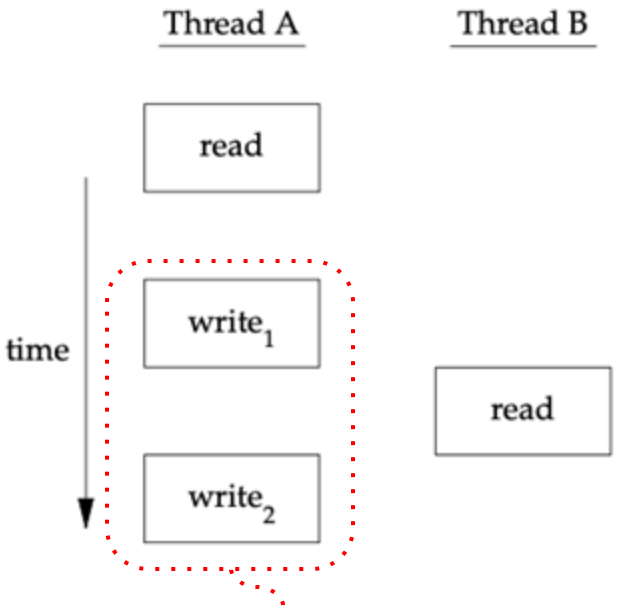
Process primitive	Thread primitive	Description
`fork`	`pthread_create`	create a new flow of control
`exit`	`pthread_exit`	exit from an existing flow of control
`waitpid`	`pthread_join`	get exit status from flow of control
`atexit`	`pthread_cleanup_push`	register function to be called at exit from flow of control
`getpid`	`pthread_self`	get ID for flow of control
`abort`	`pthread_cancel`	request abnormal termination of flow of control

Figure 11.6 Comparison of process and thread primitives

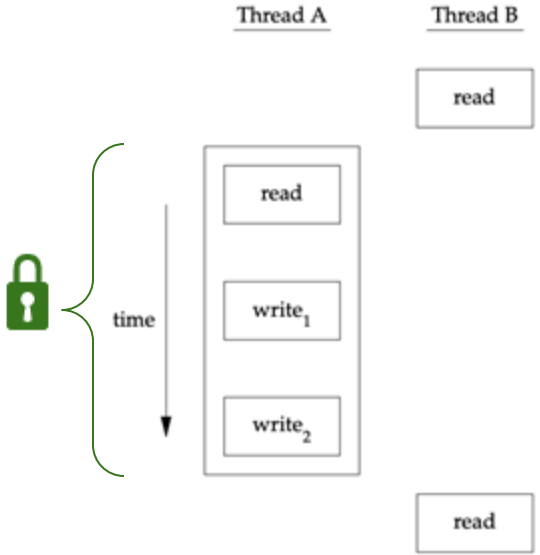
Thread Synchronization

- Important to make sure that each thread sees a **consistent** view of data
- If each thread uses variables that other threads don't read or modify, no consistency problems will exist.
- If a variable is read-only, there is no consistency problem with more than one thread reading its value at the same time.
- When one thread can modify a variable that other threads can read or modify, we need to synchronize the threads to ensure that they don't use an invalid value when accessing the variable's memory contents.

Thread Synchronization / Locks



In this example, the write operation takes two cycles. Thread B may see an inconsistent value



Synchronized using a lock that will allow only one thread to access the variable at a time

Mutexes

Mutual-exclusion interfaces (**mutexes**) ensure access by only one thread at a time.

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Both return: 0 if OK, error number on failure

Mutexes

A mutex is a lock that we set (lock) before accessing a shared resource and release (unlock) when we're done. While it is set, any other thread that tries to set it will block (pause execution) until we release it.

```
#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);
```

All return: 0 if OK, error number on failure

Other Sycnhronization Mechanisms

- Reader–writer locks (`pthread_rwlock_*`) allow three states: locked in read mode, locked in write mode, and unlocked
  - Only one thread at a time can hold a reader–writer lock in write mode, multiple threads can lock in read mode at the same time
- A spin lock (`pthread_spin_*`) is like a mutex, except that instead of blocking a process by sleeping, the process is blocked by busy-waiting (spinning).
  - Useful in situations where locks are held for short periods of times and threads don't want to incur the cost of being descheduled.
- A barrier (`pthread_barrier_*`) allows each thread to wait until all cooperating threads have reached the same point, and then continue executing from there
  - `pthread_join` function acts as a barrier, waiting until another thread exits

Thread-Safe (Reentrant) Functions

- Multiple threadscan potentially call the same function at the same time

- If a function can be safely called by multiple threads at the same time, we say that the function is **thread-safe** (or reentrant)

- Most library functions we have seen are thread-safe, with several exceptions:
  - `getenv`
  - `setenv`
  - `strtok`
  - (plus a few dozen lesser-used functions)
- Thread-safe variations exist, with the suffix `_r` (for reentrant)
  - `strtok_r`, `getenv_r`, etc.