Threads 1: Introduction

1. **Introduction**
   - **threads vs. processes**
   - **what is shared and what is not**
   - **POSIX threads (Pthreads)**
   - **the main thread**
   - **Pthreads and attributes**
   - **Pthreads and I/O**
   - **pread() and pwrite()**
   - **Pthreads and signals**
2. Creation and Termination
3. Synchronization (part a)
4. Synchronization (part b)
5. Miscellaneous

Threads vs. Processes

A traditional program has a *single* **thread of execution** active within a *single* **process**.

Most modern operating systems also support **multithreading**: the ability to have *multiple concurrent threads of execution* running within the address space of a *single process*.

I.e., the ability to be concurrently executing different sequences of code from within a *single program*.

The OS mechanism that supports multiple threads of execution is also typically called a **thread**.

An **OS thread** encapsulate one of possibly multiple *threads of execution* running within the context of a *single process*.

Threads vs. Processes (contd.)

Recall from the **Concurrent Computing lectures** that there are two alternative approaches for building **concurrent programs**:

- **multi-process programs** – composed of multiple *processes*, each with completely *separate address spaces* (memory)
- **multithreaded programs** – composed of multiple *OS threads*, sharing the address space of a *single process*

An OS thread is sometimes described as a **lightweight process** because the multiple threads within a process all share the same *address space* (process memory).

As a result, creating a new OS thread requires much less additional memory and less time to create than a new process.

Threads vs. Processes (contd.)

Testing by M. Kerrisk for his 2010 Linux book shows that creating a new *process* (with `fork()`) takes on the order of *ten times as long* as creating a new OS thread (with `clone()`).

However, the time to create a fairly large process is less than 10^{-3} sec., so the difference between creating a few subprocesses vs. a few OS threads is rarely going to be noticeable.

Likewise, while multiple threads require less *address space* than multiple processes, the use of **paged virtual memory** means this may have little practical effect.

Except for extreme cases, one should generally choose between using multiple processes or multiple threads based on the need for data sharing among the threads and the potential for unwanted interactions.

Threads vs. Processes: Sharing

Multithreaded programs make it *easier to share data* between the different threads of execution than do multi-process programs.

The two main program memory components that can be shared among OS threads are:

- **global variables**
- **heap memory**, i.e., dynamic memory

A **global variable** is a variable that is declared outside of `main` or any function body.

Memory for *global variables* is allocated in the **initialized data** or **uninitialized data** memory **segments** (where `static` data is also stored).

Threads vs. Processes: Sharing (contd.)

By default, a global variable's **scope** is the *entire program*, so all threads can access it.

(If one uses the `static` keyword when declaring the variable, its scope is limited to functions defined in the *same file* only.)

Threads do *not* require the use of **IPC mechanisms** since data can be shared by simply updating a global variable or dynamic memory.

Not shared among threads:

- **local** (**automatic**) **variables** in functions
  (unique to each thread since stored on thread's stack)
- **errno** variable giving syscall error code

POSIX Threads (Pthreads)

UNIX did not originally support multithreadeding.

As multithreading was added, different UNIXes adopted slightly different threading models.

Multithreading for UNIX was eventually standardized under POSIX, and this model is known as **POSIX Threads** or **Pthreads**.

This is the threading model supported by modern Linux systems.

Multithreaded programs can be written in C using the Pthreads system calls (which is what will be covered in these lectures).

C11 added support for writing multithreaded programs to C, but C11 threads are still not widely available (e.g., in glibc).

POSIX Threads (Contd.)

The Pthreads API is much more complicated than the API for multiple processes.

There are currently around 50 Pthreads system calls in Linux!

One can get more information about Linux Pthreads with the following commands:

```
man 7 pthreads
```

```
apropos pthread_
```

Note that when compiling with GCC, one must link with `libpthread` by including the option `-lpthread` (or just `-pthread`).

Pthreads API Overview

The Pthreads syscalls all start with the prefix "`pthread_`", e.g., `pthread_create`.

The entire set of calls can be divided into categories based on the types of thread operations they involve:

- creation and termination
- synchronization
- scheduling and other attributes
- thread IDs
- threads and signals

The Main Thread

When a *process* is created (with `fork()`), there is automatically a single running thread.

This is known as the **main** or **initial** thread in documentation.

In the modern Linux Pthreads implementation, the main thread has no special powers or importance.

Because of Linux' particular approach to **thread IDs** (TIDs), it is possible to determine whether a thread is the main thread (`syscall(SYS_gettid) == getpid()`).

Not all UNIXes allow the main thread to be identified.

Pthreads and Attributes

Many **attributes** are *process-wide*, so shared by all the Pthreads in a process, including:

- PID and PPID
- process group ID, session ID, controlling terminal
- UIDs and GIDs
- open file descriptors
- file mode creation mask
- current working directory (CWD)
- resource limits
- signal dispositions

Pthreads and Attributes (contd.)

Other Pthread attributes are *thread-specific*, including:

- thread ID (TID)
- stack
- program counter
- other registers
- signal mask
- `errno`

Pthreads and I/O

File descriptors are a *process-wide* resource.

Thus, threads in the same process share the *same file descriptors* in the process' **file descriptor table**.

This can cause problems, because when one thread carries out an I/O operation on an open file, it affects the file's **offset** for *every thread*.

This is not too different from what happens by default when a *child process* is created with `fork()`, though there are differences.

`fork()` creates a *copy* of a process, so the child gets a copy of the parent's *file descriptor table*.

Pthreads and I/O(contd.)

After the `fork()` the two processes have separate *file descriptor tables*, but initially the (now separate) file descriptors point to the *same entry* in the kernel's **open files table**.

Since an open file's **offset** is stored in the *open files table*, this means that when one process performs an I/O operation, it will affect the file offset for the other process as well.

Of course since processes have *separate file descriptor tables*, one process can close an FD and reopen the file, and from then the operations in one process will not affect the other's offset.

To have separate offsets, Pthreads would have to open the same file multiple times, creating unique FDs pointing to separate *open files table* entries, with separate *offsets*.

pread() and pwrite()

There are two I/O syscalls that are particularly useful when doing I/O in multithreaded programs: `pread()` and `pwrite()`.

Unlike `read()` and `write()`, these syscalls do not modify the file's *offset* (in the kernel *open files table*).

This means that they can be used to perform I/O on the *same file descriptor* in different threads, without the potentially serious interactions that `read()` and `write()` would have.

pread() and pwrite() (contd.)

`pread()` is like `read()`, but takes an *offset* argument telling it where to start reading:

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset)
```

`pwrite()` is similar to `write`, but with an offset argument:

```
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset)
```

- `offset` is the position in *bytes* from the *start of the file* at which reading or writing is to take place

Pthreads and Signals

The POSIX **signal model** was developed long before **POSIX Threads**, so interaction of signals with Pthreads can be somewhat complicated.

Nonetheless, it is possible to use signals in multithreaded programs.

In fact, dedicating a thread to handling particular signals can be very effective.

The **Signals lectures** include detailed information about the use of signals with Pthreads.