Architecture et Systèmes

Stefan Schwoon

Cours L3, 2014/15, ENS Cachan

Threads

Besides processes, threads are another way to implement concurrency.

Recall: Processes are units of execution with certain attributes and resources (id, memory, signals, . . .) that belong to the process itself. From the point of view of the system, processes function independently from one another.

Threads are units of execution *within* one process. They share certain resources (see next slides).

Threads are also called *lightweight processes*.

Private and shared resources

A thread owns its own

- program counter and call stack
- local (non-static) variables
- return value

All threads (of a process) share, e.g.,

- global (static) memory, heap
- open files (will discuss later)

Signals are “partially” shared (see later).

Working with threads

Standard library: Posix threads (Pthreads), defined in 1995

To compile programs with threads, include `pthreads.h` and use `gcc -pthread`.

See manpage `pthreads(7)` for overview

General notes about threads

A process can be thought of as a collection of threads
(by default, only one).

No hierarchy between threads inside a process.

Threads are “units of execution” scheduled by the kernel.

Each thread has a numerical identifier (type `pthread_t`).
This identifier can only be used within pthread functions.

In Linux at least, threads are also associated with a PID.

Creation of a thread

Any thread can create another thread, using the function `pthread_create`.

Unlike `fork`, the current thread is not duplicated, instead `pthread_create` works almost like a procedure call (specify a procedure and one argument).

The starting procedure of a thread must be of type `void*` and accept one argument of type `void*`.

Termination of a thread

The thread lives until it terminates the procedure it started in, or calls `phread_exit`.

Similarly to processes, a thread may return an exit value.

Any thread can wait for termination of another thread (and query its value) using `pthread_join`.

A thread can be killed by another thread using `pthread_cancel`. Even the main thread can be terminated in this way!

Diverse notes

`pstree` shows threads in curly braces.

`pthread_detach` will make a thread non-joinable (it will not enter ‘zombie’ state and hence cannot return a value).

Lifetime of process and threads

The process lives as long as at least one thread is alive.

Also, any of the following terminates the process along with all its threads:

- a call to `exit` (by any thread);
- the main thread terminates the main procedure (with implicitly calls `exit`);
- the process or any thread is terminated by a signal.

Threads and Signals

Signals can be sent to an individual thread.

However, all threads of a process share the signal disposition (Term, Ignore, Stop/Cont, handler routine, . . .).

Each thread has its own signal mask.

Notice that certain terminating signals are sent by the kernel in response to certain actions of a thread. Thus, a signal like `SIGFPE`, `SIGSEV`, . . . will terminated the entire process unless caught.

Pros and cons

Advantages of threads over processes:

- Easier communication between threads (via variables, not files or signals).
- Thread creation/switching less costly
- Better use of system resources, e.g., two threads can operate on two cores.
- In short, more efficient.

Disadvantages:

- Less secure - a crash in one thread can terminate the entire program.
- Race conditions (conflicting write access).
- Certain system functions are not thread-safe (see *pthreads(7)* for a list).

Input/output

Input/output operations transfer data between memory and other processes or external devices.

In Unix/Posix, input/output is done through files.

A file is an abstract concept that has at least a *read* and a *write* operation (and possibly others).

Different implementations, e.g.:

- files on hard disk;
- temporary buffers (console, pipes);
- kernel structures (procfs);
- network sockets, external devices.

Basic ideas

Data is read/written to a file with `read`/`write`(2).

Certain files (e.g. files on disk) are random-access, i.e.

data is read or written at the *current position*;

the position is changed upon read/write or a *seek* operation.

A file may be opened in a certain access mode (read-write, read-only, . . .)

Files are identified within a process by a *file descriptor*.

File descriptors

Each process has its set of file descriptors.

The file descriptors owned by a process (at a given moment) are the files to which the process can input/output.

Open files are created by Unix functions such as `creat`, `open`, `pipe`, which return file descriptors to the caller.

`open`(2) opens a file, e.g. on the disk.

`fork` duplicates a process *and its file descriptors*.

`dup` duplicates a file descriptor within the same process.

`close` removes a file descriptor from the process.

Open file table

File descriptors within a process refer to a table of “open files”, maintained by the kernel.

An open file is a data structure that permits access to a file:
type, access mode, position in file, buffered data, . . .

This is a system-wide structure (there’s only one of it).

Multiple open file entries may access the same data (but, e.g., with different access modes, positions, . . .).

An open-file entry persists as long as there is a file descriptor in some process that references it.