8. Threads

Outline

- Principles and motivating examples
- Thread creation and synchronization
- Threads and signals
- Thread-level parallelism for shared-memory parallel computing

Facilitating Shared Memory Concurrency and Parallel Computing

Motivation

- Fine-grain concurrency to reduce process creation and context switch overhead
  $\longrightarrow$ Lightweight processes
- Implement shared-memory parallel applications
  $\longrightarrow$ Take advantage of cache-coherent parallel processing hardware: SMT (simultaneous multi-threaded or hyper-threaded), CMP (chip multi-processor or multi-core), SMP (symmetric multi-processor), or NUMA (non-uniform memory architecture)

Principles

- *Thread-level concurrency*: a single process may contain multiple *control threads*, or simply, *threads*
  - Share the *same global memory*: code, data and heap
  - Distinct, *separate stack*
- `$ man 7 pthreads`

Multi-Threaded Applications

Thread-Level Concurrency

- Many algorithms express more naturally with independent computation flows
  - Reactive and interactive systems: safety critical controller, graphical user interface, web server, etc.
  - Large applications with modular structure: distributed component engineering (CORBA), remote function/method call/invocation, etc. (combine processes and threads)

Thread-Level Parallelism

- Found largely in server (database, web server, etc.) and computational (numerical simulation, signal processing, etc.) applications
- Goals
  - Tolerate latency (I/O or memory), e.g., creating more logical threads than hardware threads
  - More scalable usage of hardware resources, due to physical limitations of nanometric circuit technologies, e.g., multi-core processors

Threads vs. Processes

Shared Attributes

- PID, PPID, PGID, SID, UID, GID
- Current and root directories, controlling terminal, open file descriptors, record locks, file creation mask (`umask`)
- Timers, signal settings, priority (`nice`), resource limits and usage

Threads vs. Processes

Shared Attributes

- PID, PPID, PGID, SID, UID, GID
- Current and root directories, controlling terminal, open file descriptors, record locks, file creation mask (`umask`)
- Timers, signal settings, priority (`nice`), resource limits and usage

Distinct Attributes

- Thread identifier: `pthread_t` data type
- Signal mask (`pthread_sigmask()`)
- `errno` variable
- Scheduling policy and real-time priority
- CPU affinity (NUMA machines)
- Capabilities (Linux only)

Threads vs. Processes

Shared Attributes

- PID, PPID, PGID, SID, UID, GID
- Current and root directories, controlling terminal, open file descriptors, record locks, file creation mask (`umask`)
- Timers, signal settings, priority (`nice`), resource limits and usage

Distinct Attributes

- Thread identifier: `pthread_t` data type
- Signal mask (`pthread_sigmask()`)
- `errno` variable
- Scheduling policy and real-time priority
- CPU affinity (NUMA machines)
- Capabilities (Linux only)

To use POSIX threads, link with `-lrt` or compile with `-pthread`

Thread-Local Storage

Thread Static Data (TSD)

- Private memory area associated with each thread
- Some static and global variables "want to be private"
- Example: `errno`
- More examples: OpenMP programming language extensions
  General compilation method: *privatization*

Finalization Functions

- Privatization of non-temporary data may require
  - *Copy-in*: broadcast shared value into multiple private variables
  - *Copy-out*: select a private value to update a shared variable upon termination
- Memory management (destructors) for dynamically allocated TSD

System Call: pthread_create()

Create a New Thread

```
#include <pthread.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

Semantics

- The new thread calls the function `start_routine` passing it `arg` as first argument
- The `attr` argument corresponds to thread attributes, e.g., it can be *detached* or *joinable*, see `pthread_attr_init()` and `pthread_detach()` for details
  If `NULL`, default attributes are used (it is *joinable* (i.e., not *detached*) and has default (i.e., non *real-time*) scheduling policy
- Returns **0** on success, or a non-null error condition (not `errno`); stores identifier of the new thread in the location pointed to by the `thread` argument

System Call: pthread_exit()

Terminate the Calling Thread

```
#include <pthread.h>

void pthread_exit(void *retval);
```

Semantics

- Terminates execution
  - After calling cleanup handlers (set with pthread_cleanup_push()
  - Then calling finalization functions for thread-specific data (see pthread_key_create())
- The retval argument (an arbitrary pointer) is the return value for the thread; it can be consulted with pthread_join()
- The function is called implicitely if the thread routine returns (using its return value)
- pthread_exit() never returns

System Call: pthread_join()

Wait For Termination of Another Thread

```
#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return);
```

Semantics

- Suspends execution of the calling thread until the th terminates or is *canceled* (see pthread_cancel())
- If thread_return is not null
  - It stores the pointer returned upon termination of th
  - Or it stores PTHREAD_CANCELED if th was canceled
- Thread th must be in the *joinable* state (i.e., not *detached*, e.g. calling pthread_detach())
- Thread resources are *not* freed upon termination, only when calling pthread_join() (except if detached); watch out for memory leaks!
- Important: *at most one thread may wait for the termination of a given one*
- Returns **0** on success, or a non-null error condition (not errno)

Threads and Signals

Sending a Signal to A Particular Thread

→ `pthread_kill()`
Behaves like `kill()`, but *signal actions and handlers are global to the process*

Blocking a Signal in A Particular Thread

→ `pthread_sigmask()`
Behaves like `sigprocmask()`

Suspending A Particular Thread Waiting for Signal Delivery

→ `sigwait()`
Behaves like `sugsuspend()`, with a hybrid of thread-local — suspending thread execution — and process-global behavior — blocking a set of signals.

Example: Typical Thread Creation/Joining

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <sys/times.h>

#define NTHREADS 5

void *thread_fun(void *num) {
  int i = *(int *)num;

  printf("Thread %d\n", i);          // Or: pthread_self()

  // ...
  // More thread-specific code
  // ...

  pthread_exit(NULL);          // Or simply: return NULL
}
```

Example: Typical Thread Creation/Joining

```
pthread_t threads[NTHREADS];

int main(int argc, char *argv[]) {
  pthread_attr_t attr;
  int i, error;
  for (i = 0; i < NTHREADS; i++) {
    pthread_attr_init(&attr);
    int *ii = malloc(sizeof(int));
    *ii = i;
    error = pthread_create(&threads[i], &attr, &thread_fun, ii);
    if (error != 0) {
      fprintf(stderr, "Error in pthread_create: %s \n", strerror(error));
      exit(1);
    }
  }
  for (i=0; i < NTHREADS; i++) {
    void *ptr;
    int error = pthread_join(threads[i], &ptr);
    if(error != 0) {
      fprintf(stderr, "Error in pthread_join: %s \n", strerror(error));
      exit(1);
    }
  }
}
```