Tutorial 1: Threads

CS486 - Principles of Distributed Computing

Papageorgiou Spyros

Outline

- Processes
- Parallelism
- Threads overview
- `Pthreads` library
- Thread creation: `pthread_create`
- Waiting for threads: `pthread_join`
- Mutexes
- Barriers
- Working examples

Processes

- Carry out tasks within the operating system
- Programs
  - A set of machine code instructions and data stored on disk
- Process: a program in **action**
- Multiple instances of a program can exist => multiple processes
  - Each process is independent
- Have their own separate resources
  - Address space (a “chunk” of memory..)
  - Process descriptor in OS (unique identifier -> PID)

(simple_process.c)

Parallelism

- Why is it important?
- Multicore architectures prominent
  - Allow multiple contexts to run simultaneously (e.g. multiple process)
- Contexts work together in order to solve problems **faster**
- Can we use processes then?
- Yes, **but**
  - Process creation is costly (prefer long running tasks)
  - Inter-process communication (**IPC**) is slow/hard (shared memory segments, sockets, files etc)
- Ideally we need
  - A “lighter” version of a process
  - Easier communication between contexts
- ...**threads** do that!

Threads overview

- A *lightweight* process that lives within a process
  - Lightweight => threads use far less resources
  - If **parent** process dies, all threads die
- Threads share the same **address space**
  - e.g. changes to global variables are visible to all threads
  - **Low communication overhead**
  - ..still must synchronize though
- Multiple threads can exist within the same process
- Threads are scheduled independently

The `pthreads` library

- Standardized programming interface
- Library that allows us to spawn threads
- UNIX implementation of POSIX threads => `pthreads`
- To use the library
  - `#include <pthread.h>`
  - Compile with `-lpthread`
- Basic API
  - `pthread_create` `create a thread`
  - `pthread_join` `wait for a thread`
  - `pthread_mutex_init` `initialize a mutex`
  - `pthread_mutex_lock` `attempt to acquire mutex (blocking)`
  - `pthread_mutex_unlock` `unlock mutex`
  - `pthread_mutex_destroy` `destroy mutex`

Thread creation: `pthread_create` (1 / 2)

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start routine) (void *), void *arg);`
- `pthread_t *thread`
  - Thread descriptor, kept by the main process to use `pthread_join` later on
- `const pthread_attr_t *attr`
  - Control thread attributes
  - Usually `NULL`
  - **NOTE:** can be used to control thread **affinity** (the CPU to run on)
- `void *(*start routine) (void *)`
  - Function to run when the thread is created
  - Think of it as the "`main`" of the thread
  - example: `void* worker_func(void *data)`

Thread creation: `pthread_create` (2 / 2)

- `void *arg`
  - Pass arguments to a thread
  - Can be NULL if no arguments are needed
  - Cast to void * first
  - Pass multiple arguments by creating a struct
  - **NOTE:** thread is responsible to free memory
- Return value
  - Returns 0 on success
  - Otherwise, non-zero value..

Waiting for threads: `pthread_join`

- If parent process dies (or exits) threads also exit
- Need to wait for threads to finish
- Use **`pthread_join`**
  - `int pthread_join(pthread_t *thread, void **retval)`
- Returns immediately if thread already exited
- If `pthread_exit` is used, threads can return meaningful values
- Return value
  - Returns 0 on success
  - Otherwise error number
- For multiple threads => loop over array holding thread descriptors passed to `pthread_create` previously

Mutexes

- Simple lock primitive
  - `pthread_mutex_t lock;`
- Initialize before use
  - `pthread_mutex_init(&lock);`
- Attempt to acquire lock. This call is blocking
  - `pthread_mutex_lock(&lock);`
- Release the lock
  - `pthread_mutex_unlock(&lock);`
- Free resources
  - `pthread_mutex_destroy(&lock);`
  - Make sure lock is **not** used after this call!

Barriers (1 / 2)

- Used to synchronize threads
  - e.g. make sure all threads start at the same time
- Threads are allowed to progress only when all threads have reached the same point in execution
- `pthread_barrier_t barrier;`
- Initialization
  - ```
    int pthread_barrier_init(pthread_barrier_t  *barrier,
                const pthread_barrierattr_t *restrict attr,
                unsigned count);
    ```
  - `count` controls how many calls t
  - e.g. `pthread_barrier_init(&barrier, NULL, 4);`
    ```

Barriers (2 / 2)

- Waiting on a barrier
  - `pthread_barrier_wait(&barrier)`
  - Once `count` number of threads have called `pthread_barrier_wait`, they are allowed to progress
- Destroying a barrier
  - pthread_barrier_destroy(&barrier)
- See `simple_barriers.c`

Working examples

- Threads
  - `simple_threads.c`
- Barriers
  - `simple_barriers.c`
- Simple single threaded process vs multithreaded process
  - `md5_file.c`
  - `md5_file_multithreaded.c`

Thoughts on parallelization

- Amdahl’s law for dummies
  - *“You are as slow as the slowest (usually sequential) part of your program”*
- Different degrees of parallelization
  - *Embarrassingly parallel*
  - Middle-ground...
  - *Inherently serial*
- Organizing data in a smart way can make things things trivially parallel
  - e.g. md5 sum calculation
  - Need lock for directory pointer (shared)
  - Multiple directories -> divide files equally -> no locking needed
- Thread placement **matters**
  - Cache pollution