Pthreads Basics

Parallel Computing

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria
POSIX Threads

POSIX: Portable Operating System Interface
IEEE standards defining API of software for UNIX-like operating systems

POSIX threads (Pthreads)
standard approved 1995, amendments
functions for
  creating threads
  synchronizing threads
  thread interaction

*opaque* data types for
  thread identifiers
  synchronization constructs
  attributes
...

header file `pthread.h`
compilation: `gcc -pthread -o prog prog.c`

References:
http://opengroup.org/onlinepubs/007908799/xsh/pthread.h.html
(P)Threads in Linux

How can a thread-library be implemented?

Abstraction levels:
  - threads: created by a user program
  - kernel entity: “process”, scheduled by operating system
  - processor: physical device, gets assigned kernel entities by scheduler

Design decision: how to map threads to kernel entities?

M-to-1:
  - all threads of process mapped to one kernel entity
  - fast scheduling (in library), but no parallelism

M-to-N:
  - threads of process mapped to different kernel entities
  - two-level scheduling (library and kernel) incurs overhead, but allows parallelism

1-to-1:
  - each thread mapped to one kernel entity
  - scheduling in kernel, less overhead than in M-to-N case, allows parallelism

used in most modern Linux systems: Native POSIX Threads Library (NPTL)
Pthread Lifecycle: States

Ready

- able to run, waiting for processor

Running

- on multiprocessor possibly more than one at a time

Blocked

- thread is waiting for a shared resource

Terminated

- system resources partially released but not yet fully cleaned up
  - thread's own memory is obsolete
  - can still return value

(Recycled)

- all system resources fully cleaned up
  - controlled by the operating system

- diagram showing the lifecycle states and transitions:
  - ready
  - blocked
  - running
  - terminated
  - recycled
Pthread Creation

```c
int pthread_create(arg0, arg1, arg2, arg3)
    arg0: pthread_t *tid_ptr
        where to store thread ID of type pthread_t
    arg1: const pthread_attr_t *attr
        may set certain attributes at startup
        ignored for the moment: always pass NULL → set default attributes
    arg2: void (*)(*start)(void *)
        pointer to thread's startup function
        takes exactly one void* as argument
    arg3: void *arg
        actual parameter of thread's startup function
        returns zero on success, else error code
thread ID is stored in *tid_ptr
    pthread_t pthread_self() returns ID of current thread
    int pthread_equal(pthread_t tid1, pthread_t tid2) compares IDs
Example: helloworld
```
Process creates thread which executes `main`-function → “`main`-thread”

`main`-thread behaves slightly differently from ordinary threads:

- termination of `main`-thread by returning from `main` causes process to terminate
  - all threads of process terminate
  - Example: `helloworld`

- calling `pthread_exit(...)` in `main`-thread causes process to continue
  - all created threads continue
  - recall lifecycle: `main`-thread terminates → resources partially released
  - Attention: stack may be released!
  - memory errors: dereferencing pointers into `main`-thread's (released) stack
  - Example: `helloworld_buggy`
Pthread Termination

generally: thread terminates if startup function returns

```c
int pthread_exit(void *value_ptr)
```
causes thread to terminate (special semantics in main-thread)
implicitly called if thread's startup function returns (except in main-thread)
value_ptr is the thread's return value (see pthread_join(...))

```c
int pthread_detach(pthread_t tid)
```
resources of tid can be reclaimed after tid has terminated
default: not detached
any thread can detach any thread (including itself)

```c
int pthread_join(pthread_t tid, void **value)
```
returns when tid has terminated (or already terminated), caller blocks
optionally stores tid's return value in *value
return value from calling pthread_exit(...) or returning from startup function
joined thread will be implicitly detached
detached threads can not be joined
Pthread Termination - Examples

Example: helloworld_join

 Returning values from threads

  returning values from threads via pthread_join(...)
  example: returnval
  but: waiting for termination often not needed
  good practice to release system resources as early as possible

 alternative to pthread_join(...): custom return mechanism
  threads store their return values on the heap
  Example: returnval_heap
  problem: need to notify main-thread somehow that all threads have written results

 error: joining a detached thread
  resources are (may be or not) already released
  join should fail
  Example: returnval_buggy

 error: returning pointer to local variable
  Example: returnval_buggy
Pthread Lifecycle Revisited (1/2)

Creation

process creation ➔ main-thread creation

pthread_create(...): new threads are ready

no synchronization between pthread_create(...) and new thread's execution

Startup

main-thread's main function called after process creation

newly created threads execute startup function

Running

ready threads are eligible to acquire processor ➔ will be running

scheduler assigns timeslice to ready thread ➔ threads will be preempted

switching threads ➔ context (registers, stack, pc) must be saved

Blocking

running threads may block, e.g. to wait for shared resource

blocking threads become ready (not running) again
Pthread Lifecycle Revisited (2/2)

Termination

- generally: when thread returns from startup function
- `pthread_exit`
- can also explicitly be cancelled by `pthread_cancel(...)`
- (optional cleanup handlers are called)
- only thread's ID and return value remain valid, other resources might be released
- terminated threads can still be joined or detached
  - joined threads will be implicitly detached, i.e. all its system resources will be released

Recycling

- occurs immediately for terminated, detached threads → all resources released
Creating and Using Threads: Pitfalls

Sharing pointers into stack memory of threads
   perfectly alright, but handle with care
   passing arguments
   returning values

Resources of terminated, non-detached threads can not fully be released
   large number of threads → performance problems?
   should join or detach threads

Relying on the speed/order of individual threads
   do not make any assumptions!
   need mechanism to notify threads that certain conditions are true
      example: returnval_heap
   must prevent threads from modifying shared data concurrently
      example: sum

→ Synchronization