9. Threads

- Applications
- Principles
- Programmer Interface
- Threads and Signals
- Example
- Threads and Mutual Exclusion
- Logical Threads vs. Hardware Threads

Lightweight Shared Memory Concurrency

Motivations

- Finer-grain concurrency than processes
  - Reduce cost of process creation and context switch
  - $\approx$ lightweight processes (save the process state)
- Implement shared-memory parallel applications
  - Take advantage of cache-coherent parallel processing hardware

9. Threads

Multi-Threaded Applications

Thread-Level Concurrency

- Many algorithms can be expressed more naturally with independent computation flows
- Reactive and interactive systems: safety critical controller, graphical user interface, web server, etc.
- Client-server applications, increase modularity of large applications without communication overhead
- Distributed component engineering (CORBA, Java Beans), remote method invocation, etc.

Multi-Threaded Applications

Thread-Level Parallelism

- Tolerate latency (I/O or memory), e.g., creating more logical threads than hardware threads
- Scalable usage of hardware resources, beyond instruction-level and vector parallelism
- Originate in server (database, web server, etc.) and computational (numerical simulation, signal processing, etc.) applications
- Now ubiquitous on multicore systems: Moore's law translates into performance improvements through thread-level parallelism only

9. Threads

Principles

Thread-Level Concurrency and Parallelism

- A single process may contain multiple *POSIX threads*, a.k.a. *logical threads*, or simply, *threads*
  - Share a *single memory space*
    - Code, static data, heap
  - Distinct, *separate stack*
- Impact on operating system
  - Schedule threads and processes
  - Map POSIX threads to hardware threads
  - Programmer interface compatibility with single-threaded processes
- `$ man 7 pthreads`

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, `$ man 7 capabilities`)

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, `$ man 7 capabilities`)

To use POSIX threads, compile with `-pthread`

9. Threads

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

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()
- Called implicitely if the thread routine returns
- pthread_exit() never returns

System Call: pthread_join()

Wait For Termination of Another Thread

```
#include <pthread.h>

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

Semantics

- Suspend execution of the calling thread until thread *terminates* or is *canceled*, see pthread_cancel()
- If thread_return is not null
  - Its value is the pointer returned upon termination of thread
  - Or PTHREAD_CANCELED if thread was canceled
- thread must not be *detached*, see pthread_detach()
- Thread resources are *not* freed upon termination, only when calling pthread_join() of pthread_detach(); watch out for memory leaks!
- Return **0** on success, or a non-null error condition
- Note: errno is *not* set

Thread-Local Storage

Thread-Specific Data (TSD)

- *Private memory* area associated with each thread
- Some global variables need to be private
  - Example: `errno`
  - More examples: OpenMP programming language extensions
  - General compilation method: *privatization*
- Implementation: `pthread_key_create()`

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

9. Threads

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()`, suspending thread execution (thread-local) and blocking a set of signals (global to the process).

9. Threads

Example: Typical Thread Creation/Joining

```
#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++) {
    error = pthread_join(threads[i], NULL);
    if (error != 0) {
      fprintf(stderr, "Error in pthread_join: %s \n", strerror(error));
      exit(1);
    }
  }
}
```

9. Threads

System Call: pthread_mutex_init()

Initialisation of a mutex

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *attr);
```

Semantics

- Perform `mutex` initialization
- The `mutex` variable has to be shared among the threads willing to use the same lock; initialization has to occur exactly one time
  - For re-using an already initialized mutex see `pthread_mutex_destroy`
- The `attr` argument is the mutex type attribute: it can be *fast*, *recursive* or *error checking*; see `pthread_mutexattr_init()`
  - If `NULL`, *fast* is assumed by default
- Return **0** on success, or a non-null error condition
- Initialization can also be performed statically with default attributes by using:
  `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`

System Call: pthread_mutex_*un*lock()

Acquiring/Releasing a lock

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Semantics of pthread_mutex_lock

- Block the execution of the current thread until the lock referenced by mutex becomes available
  - Attemtping to re-lock a mutex after acquiring the lock leads to different behaviour depending on mutex attributes (see previous slide)
- The system call is *not* interrupted by a signal
- Return **0** on success, or a non-null error condition

Semantics of pthread_mutex_unlock

- Release the lock (if acquired by the current thread)
- The lock is passed to a blocked thread (if any) depending on schedule
- Return **0** on success, or a non-null error condition

System Call: pthread_mutex_*try/timed*lock()

Acquiring a lock without blocking

```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t * mutex);
int pthread_mutex_timedlock(pthread_mutex_t * mutex,
                            struct timespec * abs_timeout);
```

Semantics of pthread_mutex_trylock

- Try to acquire the lock and return immediately in case of failure
- Return **0** on success, or a non-null error condition

Semantics of pthread_mutex_timedlock

- Block the execution of the current thread until the lock becomes available or until abs_timeout elapses
- Return **0** on success, or a non-null error condition

Read/Write Locks

Principles

- Allow concurrent read and guarantee excluse write
- Similar API to regular mutexes
  - pthread_rwlock_init() – initialize a read/write lock
  - pthread_rwlock_rdlock() – get a shared read lock
  - pthread_rwlock_wrlock() – get an exclusive write lock
  - pthread_rwlock_unlock() – unlock an exclusive write or shared read lock
  - pthread_rwlock_tryrdlock() – get a shared read lock w/o waiting
  - pthread_rwlock_trywrlock() – get an exclusive write lock w/o waiting
  - pthread_rwlock_timedrdlock() – get a shared read lock with timeout
  - pthread_rwlock_timedwrlock() – get an exclusive write lock with timeout

Condition Variables

Overview

- Producer-Consumer synchronization mechanism
- Block the execution of a thread until a boolean predicate becomes true
- Require dedicated instructions to wait without busy-waiting

Principles

- A mutex is used to atomically test a predicate, and according to its value:
  - either the execution continues
  - or the execution is blocked until it is *signaled*
- Once signaled, the thread waiting on the condition resumes
- The mutex prevents race-conditions when a thread is going to wait while being signaled

System Call: pthread_cond_wait()

Blocking a thread according to a given condition

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond,pthread_mutex_t *mutex);
```

Semantics

- Atomically block the execution of a thread and release the mutex lock
- Once the condition variable cond is signaled by another thread, atomically reacquire the mutex lock and resume execution

- Return **0** on success, or a non-null error condition
- Like mutex variables, condition variables have to be initialized with a system call
- pthread_cond_timedwait() can also resume the execution after the end of a given timeout

System Call: pthread_cond_*signal/broadcast*()

Signaling or broadcasting a condition

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

Semantics

- Signal *one* (pthread_cond_signal) or *every* (pthread_cond_broadcast) threads waiting on the condition variable cond.
- If no thread is waiting, nothing happens. Signal is *lost*.
- Return **0** on success, or a non-null error condition

Example: Typical use of Condition Variables

```
int x, y; //  Shared variables
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void *thread_one(void *param) {
   // ...
   pthread_mutex_lock(&mutex);
   while (x <= y) {
      pthread_cond_wait(&cond, &mutex);
   }
   //  Now we can be sure that x > y
   pthread_mutex_unlock(&mutex);
   //  No more guarantee on the value of x > y
}
void *thread_two(void *param) {
    // ...
    pthread_mutex_lock(&mutex);
    // modification of x and y
    // no need to send a signal if the predicate is false
    if (x > y)
      pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&mutex);
}
```

pthread Implementation: Futexes

Futexes Overview

- Futex: fast userspace mutex
- *Low level* synchronization primitives used to program *higher-level* locking abstractions
- Appeared recently in the Linux kernel *(since 2.5.7)*
- Rely on:
  - a *shared integer in user space* to synchronize threads
  - two system calls (*kernel space*) to make a thread wait or to wake up a thread
- *Fast:* most of the time only the shared integer is required
- *Difficult to use:* no deadlock protection, subtle correctness and performance issues
- For more information: read *futexes are tricky* by Ulrich Drepper `http://people.redhat.com/drepper/futex.pdf`

9. Threads

- Applications
- Principles
- Programmer Interface
- Threads and Signals
- Example
- Threads and Mutual Exclusion
- Logical Threads vs. Hardware Threads

Logical Threads vs. Hardware Threads

Logical Thread Abstraction

Multiple *concurrent* execution contexts of the same program, cooperating over a single memory space, called *shared address space* (i.e., shared data, consistent memory addresses across all threads)

Among the different forms of logical thread abstrations, *user-level* threads do not need a processor/kernel context-switch to be scheduled

Mapping Logical to Hardware Threads

The hardware threads are generally exposed directly as operating system kernel threads (POSIX threads); these can serve as *worker threads* on which user-level threads can be mapped

Mapping strategies: one-to-one, many-to-one ("green" threads), *many-to-many*

Logical Threads vs. Hardware Threads

Thread “Weight”

1. Lightest: run-to-completion coroutines
   → indirect function call
2. Light: coroutines, fibers, protothreads, cooperative user-level threads
   → garbage collector, cactus stacks, register checkpointing
3. Lighter: preemptive user-level threads
   → preemption support (interrupts)
4. Heavy: kernel threads (POSIX threads)
   → context switch
5. Heavier: kernel processes
   → context switch with page table operations (TLB flush)

Task Pool

General approach to schedule user-level threads

- Single task queue
- Split task queue for scalability and dynamic load balancing

More than one pool may be needed to separate ready threads from waiting/blocked threads

Task Pool: Single Task Queue

Simple and effective for small number of threads

Caveats:

- The single shared queue becomes the point of contention
- The time spent to access the queue may be significant as compared to the computation itself
- Limits the scalability of the parallel application
- Locality is missing all together

Task Pool: Split Task Queue

Work Sharing

Threads with more work push work to threads with less work A centralized scheduler balances the work between the threads

Work Stealing

A thread that runs out of work tries to steal work from some other thread

The Cilk Project

- Language for dynamic multithreaded applications
- C dialect
- Developed since 1994 at MIT in the group of Charles Leiserson
  `http://supertech.csail.mit.edu/cilk`
  Now part of Intel Parallel Studio (and TBB, ArBB)
- Influenced OpenMP tasks (OpenMP 3.0), and other coroutine-based parallel languages

Fibonacci in Cilk

- Tasks are (nested) coroutines
- Two keywords:
  - spawn *function*() to indicate that the function call *may* be executed as a coroutine
  - sync to implement a *synchronization barrier*, waiting for *all previously spawned tasks*

```
cilk int fib(int n) {
  if (n < 2)
    return n;
  else {
    int x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```