"C - Grundlagen und Konzepte"

Threads

Marcel Hellwig
1hellwig@informatik.uni-hamburg.de

Universität Hamburg

5. Juli 2013
SOSE 13
This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
Questions?
   Just raise your hand and ask (in German, if you like).

Fragen?
   Einfach die Hand heben und fragen (gerne auch in Deutsch).
# Table of contents

1. Introduction
2. PThreads
3. Examples
4. Pitfalls and Solutions
5. C11
6. C++11
Why run things in parallel

- computing in parallel may be faster
- take advantage of multi-core systems
- use of different resources of the system
- because we can 😊
Ways to run things in parallel

- Forks
- Threads
(Process) Forks

- call `fork()` in C program
- inherits the code and the IP of parent process
- but not the memory
- child starts where `fork()` has been called
- parent and child processing independent from each other
- simple example: call “ls” in bash
(Process) Forks

- call `fork()` in C program
- inherits the code and the IP of parent process
- but not the memory
- child starts where `fork()` has been called
- parent and child processing independent from each other
- simple example: call “ls” in bash

Horror of every unix user:

`:(){ :|:&};:`
Threads

- often called light-weight process
- share resources with parent (e.g. file descriptor, PID, signals)
- much cheaper to create than forks
- starts not at the same point, but in a (user-)defined function
### Battle: Forks vs Threads

50000 thread/fork creation that won’t do anything

<table>
<thead>
<tr>
<th>It#</th>
<th>Fork</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5830</td>
<td>750</td>
</tr>
<tr>
<td>2</td>
<td>5680</td>
<td>730</td>
</tr>
<tr>
<td>3</td>
<td>5840</td>
<td>750</td>
</tr>
<tr>
<td>4</td>
<td>5620</td>
<td>740</td>
</tr>
<tr>
<td>5</td>
<td>5660</td>
<td>770</td>
</tr>
<tr>
<td>6</td>
<td>5580</td>
<td>750</td>
</tr>
<tr>
<td>7</td>
<td>5780</td>
<td>780</td>
</tr>
<tr>
<td>8</td>
<td>5890</td>
<td>750</td>
</tr>
<tr>
<td>9</td>
<td>5810</td>
<td>760</td>
</tr>
<tr>
<td>10</td>
<td>5670</td>
<td>770</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>5736</strong></td>
<td><strong>750</strong></td>
</tr>
</tbody>
</table>
Why threads?

- light-weight
- easier to manage than forks
- start at a defined function, not at the thread_create point
Why not threads?

- high complexity
- very hard to debug
- produce errors more often (than ST application)
- not a speed up every time
Where do we use threads?

- GUI-Application
- Network-application
- Garbagecollection
- “Divide and Conquer”
PThreads

- short for POSIX Threads
- API for thread using in C (before C11)
- defined for all POSIX systems
- therefore highly portable
- “Native POSIX Thread Library”
#include <pthread.h>

<table>
<thead>
<tr>
<th>Pthread_</th>
<th>main functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_attr_</td>
<td>thread attribute objects</td>
</tr>
<tr>
<td>pthread_mutex_</td>
<td>mutexes</td>
</tr>
<tr>
<td>pthread_mutexattr_</td>
<td>mutex attributes objects</td>
</tr>
<tr>
<td>pthread_cond_</td>
<td>condition variables</td>
</tr>
<tr>
<td>pthread_condattr_</td>
<td>condition attributes objects</td>
</tr>
<tr>
<td>pthread_rwlock_</td>
<td>read/write locks</td>
</tr>
</tbody>
</table>
PThread API

```c
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void **start_routine)(void *), void *arg);
int pthread_join(pthread_t thread, void **status);
int pthread_cancel(pthread_t thread);
void pthread_exit(void *status);
```

man 7 pthreads
quicksort

Classic “Divide and Conquer” ¹

Quicksort

¹please don’t mix up with “Multiply and Surrender”
## quicksort

<table>
<thead>
<tr>
<th>50 \cdot 10^6 Elements</th>
<th>time in ms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>std::sort()</td>
</tr>
<tr>
<td>1</td>
<td>7309</td>
</tr>
<tr>
<td>2</td>
<td>6977</td>
</tr>
<tr>
<td>3</td>
<td>7180</td>
</tr>
<tr>
<td>4</td>
<td>6933</td>
</tr>
<tr>
<td>5</td>
<td>7189</td>
</tr>
<tr>
<td>6</td>
<td>7040</td>
</tr>
<tr>
<td>7</td>
<td>7040</td>
</tr>
<tr>
<td>8</td>
<td>7187</td>
</tr>
<tr>
<td>9</td>
<td>7145</td>
</tr>
<tr>
<td>10</td>
<td>6898</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>7089.8</td>
</tr>
</tbody>
</table>
**simple example**

```c
void *par(void *par) {
    int i;
    for(i = 0; i < 10; i++) {
        printf("%d is here.\n", i);
        sched_yield();
    }
}

int main() {
    int i;
    pthread_t t;
    pthread_create(&t, NULL, par, NULL);
    for(i = 0; i < 10; i++) {
        printf("%d was here.\n", i);
        sched_yield();
    }
    pthread_join(t, NULL);
}
```
simple example

Possible output:

0 was here. 6 was here.
0 is here. 7 was here
1 was here. 4 is here.
2 was here. 5 is here.
3 was here. 6 is here.
1 is here. 7 is here.
2 is here. 8 is here.
3 is here. 9 is here.
4 was here. 8 was here.
5 was here. 9 was here.
int i;

void *par(void *par) {
    for(i = 0; i < 10; i++) {
        printf("%d is here.\n", i);
        sched_yield();
    }
}

int main() {
    pthread_t t;
    pthread_create(&t, NULL, par, NULL);
    for(i = 0; i < 10; i++) {
        printf("%d was here.\n", i);
        sched_yield();
    }
    pthread_join(t, NULL);
}
pitfalls

0 was here.
0 is here.
1 was here.
3 was here.
4 was here.
5 was here.
6 was here.
7 was here.
8 was here.
9 was here.
2 is here.

What happened here?
pitfalls

0 was here.
0 is here.
1 was here.
3 was here.
4 was here.
5 was here.
6 was here.
7 was here.
8 was here.
9 was here.
2 is here.

What happend here? ⇒ lost update ∧ race-condition
### race-conditions

<table>
<thead>
<tr>
<th>time</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>reads 100€</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>reads 100€</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>withdraw 75€</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>write new balance: 25€</td>
</tr>
<tr>
<td>5</td>
<td>deposits 50€</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>write new balance: 150€</td>
<td></td>
</tr>
</tbody>
</table>

Balance: 100€

new balance: 150€ ⇒ Wrong!
## race-conditions

A balance: 100€

<table>
<thead>
<tr>
<th>time</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>reads 100€</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>reads 100€</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>withdraw 75€</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>write new balance: 25€</td>
</tr>
<tr>
<td>5</td>
<td>deposits 50€</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>write new balance: 150€</td>
<td></td>
</tr>
</tbody>
</table>

New balance: 150€ ⇒ Wrong!

Is there a way out of this dilemma?
Locks

- grant exclusive access to certain regions in code/resource
- provide at least two operations
  - lock
  - unlock
Sorts of locks

- semaphore
- mutex-lock
- read-write-lock
- monitoring
# semaphore

```c
struct Semaphor {
    int counter;
    Queue queue;
};

void Lock (Semaphor s) {
    s.counter = s.counter - 1;
    if (s.counter < 0)
        self_block(s.queue);
}

void Unlock (Semaphor s) {
    s.counter = s.counter + 1;
    if (s.counter <= 0)
        unblock(s.queue);
}
```
Mutex

- in fact a binary semaphore
- has concept of an owner
- recursive locking is possible
Mutex

mutex a;
int i;

void inc()
{
    lock(a);
    i++;
    unlock(a);
}

void dec()
{
    lock(a);
    i--;
    unlock(a);
}
## Problems while Locking

```c
mutex a;
int i;

void incAndPrint()
{
    lock(a);
    inc();
    print();
    unlock(a);
}
```

```c
void inc()
{
    lock(a);
    i++;
    unlock(a)
}
```

```c
void print()
{
    lock(a);
    printf("%d", i);
    unlock(a);
}
```
Recursive Locking

```c
mutex a;
int i;

void incAndPrint()
{
    rec_lock(a);
    inc();
    print();
    rec_unlock(a);
}

void inc()
{
    rec_lock(a);
    i++;
    rec_unlock(a)
}

void print()
{
    rec_lock(a);
    printf("%d", i);
    rec_unlock(a);
}
```
deadlock
A deadlock is a state where is no back and no forth.

mutex a, b;

```c
void thread1() {
    lock(a);
    lock(b);
}

void thread2() {
    lock(b);
    lock(a);
}
```
A deadlock is a state where it is no back and no forth.

```c
mutex a, b;

void thread1() {
    lock(a);
    lock(b);
}

void thread2() {
    lock(b);
    lock(a);
}
```

Solution

```c
mutex a, b;

void thread1() {
    lock(a);
    lock(b);
}

void thread2() {
    lock(a);
    lock(b);
}
```
Read-/Writelock

- grants multiple threads access to one resource
- multiple threads can read
- only one thread can write
- can gain more performance
- C++ STL is designed to have multiple read access
mutex m;
list li;

int findItem(char *item)
{
    readLock(m);
    int result = li.find(item);
    readUnlock(m);
    return result;
}

void insertItem(char *item)
{
    writeLock(m);
    li.insert(item);
    writeUnlock(m);
}
Monitor

- also used for protecting critical regions
- provide more support like mutex
  - wait
  - notify
C11 API

- copied API mostly from pthreads
- get rid of thread attributes
- common prefix is thrd_ \(^2\)
- currently no compiler supports the C11 standard
- they rely on pthreads or the windows implementation

\(^2\)are people too lazy to type in two more characters?
#include <threads.h>

```c
thrd_t t; // thread struct
thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
thrd_join(thrd_t *thr);
thrd_exit(int res);
```
C11 supports also supports

- mutex
- condition variables
- thread local storage
C++11

- RAII idiom (Resource Acquisition Is Initialization)
- C++ has support of
  - constructors
  - destructors
  - exceptions
- therefore thread things can get more complicated
- C++11 is supported by most of modern compilers (gcc, clang, msvc)
C++11 example

```cpp
#include <iostream>
#include <thread>

int main()
{
    std::thread t1([]() {
        std::cout << "Hallo" << std::endl;
    });

    t1.join();
    return 0;
}
```
std::mutex m;
int i;

void foo()
{
    m.lock();
    do_fancy_stuff_with(i);
    m.unlock();
}

void baz()
{
    m.lock();
    do_fancy_stuff_with(i);
    m.unlock();
}

Be aware that do_fancy_stuff_with may throw an exception
C++

- Remember RAII?
- C’tor creates object and locks mutex
- D’tor unlocks mutex and destroys object
- Fallsafe and easy to use!
- Works because C++ does not have GC, but call d’tor instant
- This way also unique_ptr works
```cpp
#include <iostream>

int main() {
    std::mutex m;
    int i;

    void foo()
    {
        std::lock_guard<std::mutex> l(m);
        do_fancy_stuff_with(i);
    }

    void baz()
    {
        std::lock_guard<std::mutex> l(m);
        do_fancy_stuff_with(i);
    }

    return 0;
}
```
C++11 mutexes

- mutex
- timed_mutex
- recursive_mutex
- recursive_timed_mutex
- shared_mutex
C++11 locking

- lock_guard
  - provides needed methods
  - lock and unlock
- unique_lock
  - like lock_guard but provides more methods
  - try_lock, time_based_lock, etc
- shared_lock
  - will be used for read-/writelocks
  - uses shared mutexes for this
C++11 fancy stuff

C++11 also provides some fancy stuff like:

- functional programming
- lambda expressions
- promises
- futures
- async threads

If you are interested, I will give a talk on C++11 at KBS next semester.
Conclusion

- Threads *could* make programs run faster
- increase complexity a lot
- be aware of pitfalls like race-conditions
- use different locks for different situations
- do not forget to unlock!
- use it only when necessary
References

6 - https://computing.llnl.gov/tutorials/pthreads/fork_vs_thread.txt
11 - https://computing.llnl.gov/tutorials/pthreads/
14 - http://demin.ws/blog/english/2012/04/28/multithreaded-quicksort/
27 - http://www.nichtlustig.de/toondb/050528.html
<table>
<thead>
<tr>
<th>Sources</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://computing.llnl.gov/tutorials/pthreads/">https://computing.llnl.gov/tutorials/pthreads/</a></td>
</tr>
<tr>
<td><a href="https://en.wikipedia.org/wiki/Lock_%28computer_science%29">https://en.wikipedia.org/wiki/Lock_%28computer_science%29</a></td>
</tr>
<tr>
<td><a href="https://stackoverflow.com/questions/520837/what-are-common-concurrency-pitfalls">https://stackoverflow.com/questions/520837/what-are-common-concurrency-pitfalls</a></td>
</tr>
</tbody>
</table>