“C - Grundlagen und Konzepte”

Threads

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

Universität Hamburg

5. Juli 2013
SOSE 13

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

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

time in ms		
Ite	Fork	Thread
1	5830	750
2	5680	730
3	5840	750
4	5620	740
5	5660	770
6	5580	750
7	5780	780
8	5890	750
9	5810	760
10	5670	770
Average	5736	750

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”

PThread API

```
#include <pthread.h>
```

pthread_	main functions
pthread_attr_	thread attribute objects
pthread_mutex_	mutexes
pthread_mutexattr_	mutex attributes objects
pthread_cond_	condition variables
pthread_condattr_	condition attributes objects
pthread_rwlock_	read/write locks

PThread API

```
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

[1] please don’t mix up with “Multiply and Surrender”

quicksort

$50 \cdot 10^6$ Elements		time in ms	
Ite	std::sort()	ST	Async
1	7309	6555	2512
2	6977	6320	2337
3	7180	6516	2450
4	6933	6388	2372
5	7189	7074	2387
6	7040	7399	2339
7	7040	6875	2434
8	7187	7060	2562
9	7145	7050	2470
10	6898	6846	2422
Average	7089.8	6808.3	2428.5

simple example

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

pitfalls

```c
int i;

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

```c
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 happend 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? $\Rightarrow$ lost update $\wedge$ race-condition

race-conditions

Balance: 100€		
time	Thread 1	Thread 2
1	reads 100€	
2		reads 100€
3		withdraw 75€
4		write new balance: 25€
5	deposits 50€	
6	write new balance: 150€	

new balance: 150€ $\Rightarrow$ Wrong!

race-conditions

Balance: 100€		
time	Thread 1	Thread 2
1	reads 100€	
2		reads 100€
3		withdraw 75€
4		write new balance: 25€
5	deposits 50€	
6	write new balance: 150€	

new balance: 150€ $\Rightarrow$ 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

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

```
void Lock (Semaphor s) {
  s.counter = s.couter - 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

```
mutex a;
int i;

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

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

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

Recursive Locking

```
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


HÖREN SIE SOFORT AUF, MIR NACHZULAUFEN.
ABER... ICH LIEBE SIE.
WWW.NICHTLUSTIG.DE

deadlock

A deadlock is a state where is no back and no forth.

```
mutex a, b;

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

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

deadlock

A deadlock is a state where is no back and no forth.

```
mutex a, b;

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

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

Solution

```
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

Read-/Writelock

```
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?

C11 API

```
#include <threads.h>

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 API

C11 supports also supports

- mutex
- condition variables
- thread local storage

C++11

- RAII idom (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

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

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

    t1.join();
    return 0;
}
```

C++11 exception

```
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++

- rember RAII?
- c'tor creates object and locks mutex
- d'tor unlocks mutex and destroy object
- failsafe and easy to use!
- works because C++ does not have GC, but call d'tor instant
- this way also unique_ptr works

C++11

```
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);
}
```

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

Sources

https://computing.llnl.gov/tutorials/pthreads/
http://en.cppreference.com/w/c/thread
http://en.cppreference.com/w/cpp/thread
https://en.wikipedia.org/wiki/Lock_%28computer_science%29
https://stackoverflow.com/questions/520837/what-are-common-concurrency-pitfalls