Operating systems. Lecture 10

Michał Goliński

2018-12-04

Introduction

Recall

- `fork` – one more example
- allocating memory – system and library calls
- signals

Plan for today

- Threads
- `pthreads`
- Threads in the standard libraries
- OpenMP

Multithreading

Introduction

Modern CPU performance improved has put emphasis on the number of cores lately, not the performance of a single core. To utilise the power in these processors we either need to run many processes concurrently (e.g., compiling many files at the same time) or solve problems in parallel using threads.

Threads are an abstraction given by operating systems that allow for parallel programming: multiple CPUs executing code in a common address space.

Introduction cont.

In the past (when multiple CPUs in a single computer were very rare) threads have been used most often for operations that may block and should be run in the background such as I/O (serial multiprocessing). Today more and more often they are used to solve problems faster. Unfortunately not all computational problems can be solved in a parallel way and multithread programming brings a new set of challenges of its own.

Threads

In Linux there is no clean separation between a process and a thread – in fact for the kernel everything is just a **task**. Both threads and processes are created by the same system function `clone`.

One usually thinks that processes tend to be more separate (e.g., separate address spaces) while threads tend to share more resources. Still, processes can share memory (see `mmap`) and threads usually have some memory that is private (e.g., the stack), so there is a lot of grey area.

Threads under Windows

This is not the case under Windows: threads and processes are separate entities, created by different system calls. Threads are much more lightweight: creating a new thread can be hundreds of times faster than creating a new process.

`pthreads`

We will see how threads are created under Linux using the POSIX threads. A thread is created by running `pthread_create`:

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t
 ↪  *attr,
                   void *(*start_routine) (void *), void *arg);
```

Each thread needs its own `pthread_t` structure that should be treated as opaque. A thread runs a function (`start_routine`) that takes a pointer as argument and returns a pointer. It is the programmer's task to pass a meaningful argument to the `start_routine` and make sense of the result. The argument `attr` can set some specific attributes of the newly create thread. The return value informs of errors.

Finishing work

A thread finishes work by calling `pthread_exit` or by returning a value. The main thread should wait for all the threads by calling `pthread_join` with the `pthread_t` variable as argument. This function also gives access to the return value (a pointer).

Thread-safety

Different threads may want to call the same function (e.g., from the library) at the same time. If this can be done safely, we call such a function *thread-safe*. Most function in the POSIX standard are thread-safe, but there is a not so short list of exceptions. It can be found in `man 7 pthreads`. Thread-safety is of paramount importance in multithreaded programming.

Thread-safety is similar to reentrancy, but the are not directly equivalent: one can find examples of functions that are thread-safe and not reentrant and vice versa.

`pthread` example

```
#include <stdio.h>
#include <pthread.h>
#include <math.h>
#include <stdlib.h>
#include <stdint.h>
#include <inttypes.h>

uint64_t sum = 0;

typedef struct thread_params {
    uint32_t start;
    uint32_t end;
    pthread_mutex_t *mutex;
} thread_params;

void *parallelSum(void *arg) {
    thread_params *params = arg;
    uint64_t partialSum = 0;
    for (uint32_t i = params->start; i < params->end; ++i) {
        partialSum += i;
    }
//    pthread_mutex_lock(params->mutex);
    sum += partialSum;
//    pthread_mutex_unlock(params->mutex);
    return NULL;
}


int main(int argc, char *argv[]) {
        uint32_t n; sscanf(argv[1], "%" SCNu32, &n);
        uint32_t k; sscanf(argv[2], "%" SCNu32, &k);
    pthread_t *threads = malloc(k*sizeof(pthread_t));
    thread_params *params = malloc(k*sizeof(thread_params));
    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
    uint32_t start = 0;
    uint32_t step = ceil((n+1.0)/k);
    for(uint32_t i = 0; i < k; ++i) {
        params[i].mutex = &mutex;
        params[i].start = start;
        params[i].end = (start + step <= n+1) ? (start+step) :
↪   (n+1);
```

```
        pthread_create(&threads[i], NULL, parallelSum,
↪   &params[i]);
        start += step;
    }
    for(uint64_t i = 0; i < k; ++i) {
        pthread_join(threads[i], NULL);
    }
    free(threads);
    free(params);
    printf("%" PRIu64 "\n", sum);
    return 0;
}
```

We need to compile with `-pthread`.

Threads in the standard library

For a long period of time (too long) the C and C++ standard libraries provided no way to use threads, forcing programmers to use `pthread` or Windows Threads or other platform specific API. This was remedied in C11 and C++11, which introduced the relevant functionalities.

The API was first created in C++ and afterwords incorporated into C. It is a bit similar to `pthreads`.

`threads.h`

To use threads in C one can now use the `threads.h` header together with other goodies of the library like atomics:

```
#include <threads.h>
#include <stdatomic.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <inttypes.h>

atomic_uint_least64_t sum;

typedef struct thread_params {
    uint32_t a;
    uint32_t b;
} thread_params;
```

```
int parallelSum(void *arg) {
    thread_params *params = arg;
        uint64_t partialSum = 0;
        for (uint32_t i = params->a; i < params->b; ++i) {
                partialSum += i;
    }
        atomic_fetch_add(&sum, partialSum);
    return 0;
}

int main(int argc, char *argv[]) {
        uint32_t n; sscanf(argv[1], "%" SCNu32, &n);
        uint32_t k; sscanf(argv[2], "%" SCNu32, &k);
    uint32_t start = 0;
    uint32_t step = ceil((n+1.0)/k);
        thrd_t *threads = malloc(k*sizeof(thrd_t));
        thread_params *params = malloc(k*sizeof(thread_params));
        for(int i = 0; i < k; ++i) {
        params[i].a = start;
        params[i].b = (start+step < n+1)?(start+step):(n+1);
                thrd_create(threads+i, &parallelSum, params+i);
                start += step;
        }
        for(int i = 0; i < k; ++i) {
                thrd_join(threads[i], NULL);
        }
        printf("sum = %" PRIuLEAST64 "\n", sum);
        return 0;
}
```

thread

The library threads in C++ are accessed through the `thread` header that relies heavily on templates. This may make it difficult in the beginning (and compile errors may be completely incomprehensible) but allows for cleaner code:

```
#include <iostream>
#include <vector>
#include <thread>
#include <atomic>
#include <cmath>
#include <cstdint>
#include <cinttypes>

using namespace std;
```

```
atomic<uint64_t> sum;

void parallelSum(uint32_t a, uint32_t b) {
        uint64_t partialSum = 0;
        for (uint32_t i = a; i < b; ++i)
                partialSum += i;
        sum.fetch_add(partialSum);
}

int main(int argc, char *argv[]) {
        uint32_t n; sscanf(argv[1], "%" SCNu32, &n);
        uint32_t k; sscanf(argv[2], "%" SCNu32, &k);
    uint32_t start = 0;
    uint32_t step = ceil((n+1.0)/k);
        vector<thread> threads;
        while (start < n) {
                threads.push_back(thread(&parallelSum, start,
 ↪  min(start+step,n+1)));
                start += step;
        }
        for (auto& th: threads) {
                th.join();
        }
        cout << "sum = " << sum << endl;
        return 0;
}
```

OpenMP

OpenMP is a set of compiler routines (in the form of `#pragma` directives) that can help to generate parallel code with little effort on the programmer's side. It can easily add parallelization to an otherwise non parallel program to better utilise multicore CPUs. Creating and joining threads is done by the compiler:

```
#include <stdio.h>
#include <sys/time.h>
#include <stdint.h>
#include <inttypes.h>

uint64_t sum(uint32_t n) {
    uint64_t sum = 0;
    for(uint32_t i = 1; i <= n; ++i) {
        sum += i;
    }
    return sum;
}
```

```
uint64_t parallelSum(uint32_t n) {
    uint64_t sum = 0;
    #pragma omp parallel for reduction(+:sum)
    for(uint32_t i = 1; i <= n; ++i) {
        sum += i;
    }
    return sum;
}

int main() {
    struct timeval start, end;
    for(;;) {
        uint32_t n;
        scanf("%" SCNu32, &n);
        gettimeofday(&start, NULL);
        uint64_t s = sum(n);
        gettimeofday(&end, NULL);
        printf("The calculations took %d miliseconds, the result
 ↪  is %" PRIu64 ".\n",
               (end.tv_sec-start.tv_sec)*1000 +
 ↪  (end.tv_usec-start.tv_usec)/1000, s);
        gettimeofday(&start, NULL);
        s = parallelSum(n);
        gettimeofday(&end, NULL);
        printf("The calculations took %d miliseconds, the result
 ↪  is %" PRIu64 ".\n",
               (end.tv_sec-start.tv_sec)*1000 +
 ↪  (end.tv_usec-start.tv_usec)/1000, s);
    }
    return 0;
}
```

Theory

Easy parallelization

Some problems are particularly well suited for parallelization, they are called *embarrassingly parallel*, e.g.:

- brute-force search in cryptography (cracking passwords, extending blockchains)
- ray-tracing a graphics scene
- generating frames of computer animated movie
- calculating FFT
- compiling a large C project

Parallelization impossible

Not all problems are easily translated to parallel algorithms. Some tasks are difficult to parallelize, they are inherently sequential:

- calculating an iterate of a function (e.g. hashing in blockchain)

Parallelization difficult

Some problems can be solved in parallel but require a lot of communication between threads/processes, which makes hardware very expensive (supercomputers, not server farms), e.g., mathematical modelling by solving differential equations (e.g., weather forecasts, virtual nuclear armaments testing).

Amdahl's law

Amdahl's law calculates how much speedup is possible by parallelization. Assume that the execution time of a task can be divided in the following proportions:

- p – part that is perfectly parallelizable
- $1 - p$ – part that is sequential

Then the speedup S enjoyed by the task by speeding the parallel part s times can be calculated as follows:

$S(s) = \frac{1}{1-p+\frac{p}{s}}$.

Corollary

We cannot expect that growing the number of cores gives us linear speedup: this mostly depends on the task in question and almost always gives diminishing returns.

BACI

Ben-Ari Concurrent Interpreter

BACI is a toy programming language that helps with understanding the problems inherent to multithreaded programming. Multithreaded programs in C can have subtle bugs that will not manifest themselves very often. BACI runs programs in such a way that problems will be more evident (by interleaving threads as much as possible). It was created by Prof. Mordechai Ben-Ari from the Weizmann Institutu, hence the name.

C– compiler

BACI uses a separate compiler and an interpreter. The source code is first compiled to an intermediate representation (PCode) and then the file containing PCode instructions is executed by the interpreter. This allows for one high level instruction to map into multiple "machine" level instructions, like in general programming languages.

There are BACI compilers for a Pascal analogue and a C++ analogue. We will use the C++ analogue called C–.

Entry point

The program starts execution in the `main()` function. Note the lack of any arguments, the return type can be `int` or `void` or none – it doesn't matter much.

Basic types

C– supports just two primitive types: `int` and `char`. The variables must be declared at the beginning of the block they appear in.

Strings

BACI supports strings, but string variables must be given maximum length when declared. Functions can accept strings of any length. No bounds are checked.

String functions

- `void stringCopy(string dest, string src)` – copies strings
- `void stringConcat(string dest, string src)` – adds `src` to `dest`
- `void stringCompare(string x, string y)` – compares lexicographically
- `void stringLength(string x)`
- `int sscanf(string x, rawstring fmt, ...)` – `scanf` analogue
- `int sprintf(string x, rawstring fmt, ...)` – `printf` analogue

Arrays

One can create multidimensional arrays, as in C.

Functions

One can create functions the same way as in C. Parameters are passed by value or by reference (as in C++).

Example: arguments by reference

```
void f(int x, int &y) {
  x = 2;
  y = 2;
}

main() {
  int a = 1;
  int b = 1;
```

```
  f(a, b);
  cout << a << " " << b << endl;
}
```

Flow control

Normal compound statements (`if`, `while`, `for`, `break`, `continue`) are supported.

I/O

Our toy language uses the `cin`, `cout` from C++ together with « and ».

`cobegin`

A thread can run any function returning `void`. Functions that should be run concurrently must be put in one `cobegin` block:

```
cobegin {
  run1(...);
  run2(...);
  ...
  runN(...);
}
```

The PCode instruction will be randomly interleaved to show how the execution may differ from run to run.

Example: cobegin

```
int n = 0;

void A() {
  n = n+1;
}

main() {
  cobegin {
    A(); A();
  }
  cout << n << endl;
}
```