Parallel programming on multicore architecture

Julien 'Lta' BALLET

Ambiant-IT

June 14, 2016

- Graduated from EPITECH, class of 2009
- Teacher and head of the "OpenSource" lab
  - Linux "multimedia" specialty
- Freelance developer and trainer
  - A lot of different things
- Production Engineer @ Facebook
  - *Operating Systems: Core* team
  - Linux system programming trainer
- Linux Audio Developper

- High proficiency in C
- Knowledge of C++
- GNU/Linux environment (the shell, general usage)
- Linux development toolchain
  - g++/clang
  - gdb/lldb
  - make
  - man

- A low level approach
  - Hardware's architecture
  - Kernel's internals
  - Kernel's API and tools
- A practical approach
  - Examples
  - Exercices / Workshops
- Just enough theory

Summary

Parallel computing

What is parallel computing ?

Parallel computing is a type of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved at the same time. There are several different forms of parallel computing: bit-level, instruction-level, data, and task parallelism. Parallelism has been employed for many years [...] but interest in it has grown lately due to the physical constraints preventing frequency scaling.

Wikipedia

Instruction-level parallelism

- Branch prediction
- Instruction pipelining
- Hyper-threading
- ...

Task/Data-level parallelism

- Some overlap between the two
- Data-level
  - SIMD
  - Producer/consumer pattern
- Task Level
  - Pipeline pattern
- More on this later

Why parallelism matters ?

- Frequency scaling vs multi-core
- Operating systems are multitask
- Latency control
- Important even for non-parallel programs

Some history

- Context switching for batches *(Leo III, UK, 1961)*
- Cooperative multitasking Time-sharing *(CTSS, MIT, 1961)*
  - IBM, MacOS, Win 3.1/9.x
- Preemptive multitasking *(Multics, Cambridge, 1964)*
  - Multics, very influential OS.
  - Linux, OSX, Win NT+
  - Most OSes today
- Virtual memory *(Atlas/Burroughs, 1961/1962)*
- Threads *(OS/360, IBM, 1967)*

System concepts

What is a CPU ?

- Execute sequences of instruction
- Fetched from the memory through caches
- Operates on
  - Registers
  - Memory (through caches)
- Registers:
  - General purposes
  - PC
  - Stack pointer (BP/SP)
  - Various statuses

Memory

- Arrays of bytes for the CPU to work on
- Organized in Hierarchy:
  - register > caches > main memory > disk / networks
- A few numbers[1]:
  - Register: no latency (0-2 cycles)
  - Caches:
    - L1: 0.5 ns
    - L2: 7 ns
  - Memory: 100 ns
  - SSD: 150 000 ns

[1] https://gist.github.com/jboner/2841832

Virtual memory

- Real/Direct mode
  - Programs access memory directly
  - 0x00 $\rightarrow$ first byte of memory
  - Unsafe!
- Virtual/Protected mode
  - Programs addresses are mapped to different addresses
  - 0x00 $\rightarrow$ unknown
  - Safe !
  - Kernel maintains mapping (page directory)
  - MMU / TLB

Context switching

- Stopping the execution of a program to run another
- What is a context ?
  - Registers (pc, stack, ...)
  - Memory mappings (pagedir, TLB)
- Save the current context, load another
- Costly !

Multitasking

- Multiple program
- in their “own memory”
- running “at the same time”
- but actually sharing Memory and CPU Time
- Understanding the kernel is **vital**

The Unix/Linux model

POSIX

- *Many* UNIXes (Linux, BSDs, HP-UX, AIX, Solaris, ...)
- Common features and semantic
  - Virtual Memory
  - Preemptive multitasking
  - File system
  - Shell
  - Written in C
- Standardization effort
  - Portable Operating System Interface (IEEE, 1988-2008)
  - Single Unix Specification (Austin Group, 1997-2008)

Processes

- A running program
- Container of resources
  - Memory
  - File descriptors
  - Execution path
  - ...
- Coordinated by the kernel
  - Scheduler
  - Communication (pipes, shared memory)
  - Synchronization (locks, semaphores)
- Unix parallelism is process based

fork()

- The Unix process creation model:
  - Process duplication: *fork()*
  - Process image replacement: *exec()* family
  - Take care of lifecycle management !
- Creating processes is *cheap*
  - Many optimizations (COW, ...)

fork() example

```c
pid_t pid = fork();
if (pid) {
  // Parent (original process)
  int status;
  wait(&status); // Ask if you don't know
} else {
  // Child (new process)
  char *newenviron[] = { NULL };
  char *newargv[] = { NULL };

  // Replace the current process by "/bin/ls"
  execve("bin/ls", newargv, newenviron);
}
```

Threads

- a.k.a *Lightweight processes*
- *Relatively recent* in UNIX history
- Task with shared resources
  - Memory space
  - File descriptors
  - ...
- Multiple execution path within a process
  - 1 process: 1-n threads
- Separate stack and registers

clone()

- In Linux, *fork()* wraps *clone()*
- Duplicate tasks
  - Task: kernel name for execution context + resources
  - Something *schedulable*
- Choose what you share or copy/reset:
  - Memory
  - File descriptors
  - Network namespace, ...
- Share everything: You're a thread !
  - *and a hippie :)*

Summary

Threads API

- Lifecycle management
  - Creation
  - Destruction
- Synchronization
  - Lifecycle synchronization
  - Share resource
  - Share work
- Tuning
  - Scheduling
  - Stack
  - Signals
  - ...

Pthreads

Introducing the POSIX Threading API

- *pthreads* for short
  - *pthread_xxx* functions
  - Man pages: **man 7 pthreads**
- Low level
- Primitives
  - Lifecycle management
  - Synchronization
  - Tuning

C++ 11