Parallel programming on multicore architecture

Julien 'Lta' BALLET

Ambiant-IT

June 15, 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

```
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 resources
  - Share work
- Tuning
  - Scheduling
  - Stack
  - Signals
  - ...

POSIX Threads

Introducing the POSIX Threading API

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

Why the low-level API ?

- The more you know
  - Used in a lot of code
  - Precise documentation
  - System semantic and behavior
- More control
- Not everything is wrapped
- Access to non-portable features

Lifecycle management

- Creation
  - Process creation
  - *pthread_create()* with a function pointer
- Destruction
  - The thread function returns
  - *pthread_exit()* exits the current thread
  - *pthread_cancel()* exits another thread

Synchronization

- MUTual EXclusions (aka mutex or lock)
  - Thread can acquire/release it
  - Only 1 thread can acquire it at the time
  - Helps protect shared resources
- RW Locks
  - Acquired for Read OR Write
  - Multiple reader OR one writer at the time
- Conditions / Signals
  - Wait for a condition to happen
  - Signal (awake threads) when a condition happen
- Barriers
  - Wait for N other threads to reach a certain point

Mutex

- Helps protect a shared resource from concurrent access.
- Lifecycle
  - *pthread_mutex_init()*
  - *pthread_mutex_destroy()*
  - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- Lock
  - *pthread_mutex_lock()*
  - *pthread_mutex_timedlock()*
  - *pthread_mutex_trylock()*
- Unlock: *pthread_mutex_unlock()*

RW Locks

- Helps protect a shared resource from concurrent access.
- For *read-mostly* data
- Lifecycle
  - *pthread_rwlock_init()*
  - *pthread_rwlock_destroy()*
  - `pthread_rwlock_t rwl = PTHREAD_RWLOCK_INITIALIZER;`
- Lock as a writer:
  - *pthread_rwlock_wrlock()*
  - *_trywrlock()*, *_timedwrlock()*
- Lock as a reader:
  - *pthread_rwlock_rdlock()*
  - ...
- Unlock: *pthread_rwlock_unlock()*

Conditions

- 1-n thread(s) wait(s) on a condition
- Another thread wakes one or all waiting thread(s)
- Requires a mutex
- API calls:
  - *pthread_cond_init()* and *pthread_cond_destroy()*
  - Wait: *_cond_wait()* and *_cond_timedwait()*
  - Wake one: *_cond_signal()*
  - Wake all: *_cond_broadcast()*

Barriers

- A barrier is initialized with an int "N".
- Threads may *wait* on the barrier.
- When "N" threads are waiting, they're all resumed.
- One thread get a special value returned, the other get 0.
- API calls:
  - *pthread_barrier_init()* and *_destroy()*
  - *pthread_barrier_wait()*

Spinlocks

- Busy-wait based mutexes
- Avoid context switches
- Good for short lived locks
- API Calls:
  - *pthread_spin_init* and *_destroy*
  - *_lock()* and *_unlock()*

Utils

- Wait for a thread to terminate: *pthread_join()*
- Get called before/after fork: *pthread_atfork()*
- Get the current thread: *pthread_self()*
- Compare threads: *pthread_equals()*
- Call a function once per process: *pthread_once()*
- Relinquish the CPU: *pthread_yield()* or *sched_yield()*

Thread Local Storage

- Thread specific pointers
- Non thread-safe library (codec, interpreter, ...)
- API Calls:
  - Create a key: *pthread_key_create()*
  - *pthread_key_destroy()*
  - Store a pointer for key/thread: *pthread_setspecific()*
  - Get the pointer: *pthread_getspecific()*

TLS Example

```
static pthread_key_t key;
static pthread_once_t key_once = PTHREAD_ONCE_INIT;

static void make_key() {
  (void) pthread_key_create(&key, NULL);
}

func() {
  void *ptr;

  (void) pthread_once(&key_once, make_key);
  if ((ptr = pthread_getspecific(key)) == NULL) {
    ptr = malloc(OBJECT_SIZE);
    // ...
    (void) pthread_setspecific(key, ptr);
  }
  // ...
}
```

C++ 11

threading in std::

- C++11 adds threading support in the stdlib
- Clean C++
- Support portable behaviors and primitives
- Supported by most compilers/stdlib:
  - gcc/libstdc++
  - clang/libc++
  - MSVC 2012+
  - icc: meh !
- boost::thread is close-enoguh

A quick tour of process based parallelism

Why care about processes ?

- Original UNIX-way
- Very good support
- Safer and more robust:
  - Isolate segfaults
  - Less coupling
- Isolate incompatible licenses
- Complementary with threads

Two APIS

- System V
  - Older
  - Akward API
  - Widely supported
- POSIX
  - Not fully supported everywhere
  - More traditional semantic (open, mmap, ...)
- We'll talk about SysV APIs here

Shared memory

- Pages of memory shared between 2 or more processes
- API Calls:
  - Open/Create a segment: *shmget()*
  - Map the segment in memory: *shmat()*
  - Unmap the segment: *shmdt()*
  - Delete the segment: *shmctl()* with the *IPC_RMID* flag
- spinlocks and barriers can be shared via shared memory

Semaphores

- Interprocess synchronization primitive
- Positive or null integer with 3 operations:
  - Add another integer
  - Wait until value is null
  - Wait until value is bigger than N, then substract N

Semaphores API

- Manipulated as sets
- Create/Open a semaphore set: *semopen()*
- Operates on the semaphore set values: *semop()*
- Deletes a semaphore set: *semctl()* with *IPC_RMID* operation

Message Queues

- Exchange arbitrary messages between processes
- API Calls:
  - Create/Open a queue: *msgget()*
  - Post a message on the queue: *msgsnd()*
  - Receive a message from a queue: *msgrcv()*
    - The message is removed from the queue
  - Delete a queue: *msgctl()* with *IPC_RMID* operation

Summary

- Define a repeatable process
- Lay out some terminology
- Identify classical patterns
- Observe some real world examples

Know the problem

Serial problem

Premature optimization is the root of all evil -Knuth

- Solve the business domain problem first
- Best done as a serial problem
  - Existing code ?
  - Naive implementation ?
- Easier to grasp
- Easier to get data

Get data !

Use the data

- or do an algorithmic complexity evaluation
- Identify the time consuming parts
- Where are the hotspots ?
  - 80% of the time in 20% of the code
- Where are the bottlenecks ?
  - IO
  - I'm looking at you hard drive

Evaluate other options

- Parallel algorithms are hard
- Are there other algorithms ?
- A good-enough heuristic ?
- Already parallel libraries available ?
- Fear the threads and avoid them if possible

Is it parallelizable ?

- Real-time constraints
  - Scheduling isn't really deterministic
- Control plane latency
- Communication-intensive [2]
- **Data dependencies**

[2] https://en.wikipedia.org/wiki/Amdahl's_law

Is it parallelizable ?

- 3D rendering ?
- Game of Life ?
- Sorting algorithms ?
- Search algorithms ?
- Data compression ?
- Google Page rank :) ?

Partitioning

Drawing the dependency graph

- Input $\rightarrow$ Program $\rightarrow$ Output
- Moving data through a graph of algorithms
- Let's draw this graph
  - All the inputs
  - All the intermediary data
  - All outputs

Examples

- XXX
- Video transcoder

Data partitioning

- Which data can be split ?
- How do we split them ?
  - http requests ?
  - images ?
  - financial data analysis ?

Functional partitioning

- What are the functional blocks within the graphs ?
- Within the algorithms ?

Communication and Synchronization

Load balancing and Tuning

Common patterns