Designing for Performance: Concurrency and Parallelism

COS 518: Computer Systems
Fall 2015

Logan Stafman
Adapted from slides by Mike Freedman
Definitions

• Concurrency:
  – Execution of two or more tasks overlap in time.

• Parallelism:
  – Execution of two or more tasks occurs simultaneous.
Concurrency without parallelism?

- Parts of tasks interact with other subsystem
  - Network I/O, Disk I/O, GPU, ...

- Other task can be scheduled while first waits on subsystem’s response
Concurrency without parallelism?
Scheduling for fairness

• On time-sharing system also want to schedule between tasks, even if one not blocking
  – Otherwise, certain tasks can keep processing
  – Leads to starvation of other tasks

• Preemptive scheduling
  – Interrupt processing of tasks to process another task (why with tasks and not network packets?)

• Many scheduling disciplines
  – FIFO, Shortest Remaining Time, Strict Priority, Round-Robin
Preemptive Scheduling
Concurrency with parallelism

• Execute code concurrently across CPUs
  – Clusters
  – Cores

• CPU parallelism different from distributed systems as ready availability to shared memory
  – Yet to avoid difference between parallelism b/w local and remote cores, many apps just use message passing between both (like HPC’s use of MPI)
Symmetric Multiprocessors (SMPs)
Non-Uniform Memory Architectures (NUMA)
Pros/Cons of NUMA

• Pros
  Applications split between different processors can share memory close to hardware
  Reduced bus bandwidth usage

• Cons
  Must ensure applications sharing memory are run on processors sharing memory
Forms of task parallelism

• Processes
  – Isolated process address space
  – Higher overhead between switching processes

• Threads
  – Concurrency within process
  – Shared address space
  – Three forms
    • Kernel threads (1:1): Kernel support, can leverage hardware parallelism
    • User threads (N:1): Thread library in system runtime, fastest context switching, but cannot benefit from multi-threaded/proc hardware
    • Hybrid (M:N): Schedule M user threads on N kernel threads. Complex.
Programming with threads

• Multithreaded version:

webserverLoop() {
    newconn = accept();
    ThreadCreate(processReq(), newconn);
}

• Advantages of threaded version:
  – Can share file caches kept in memory, results of CGI scripts, ...
  – What if too many requests come in at once?
Dispatching packets to processes

• Network interrupts run at higher kernel priority than user-level tasks
  – Can lead to Receiver Livelock: All effort on receiving packets, no real work done

• Types of dispatch
  – Interrupts: One per network packet
  – Interrupt Coalescing: Wait for several pkts or timeout
  – poll: Make the user space / OS ask you
Livelock with threadCreate

• Cost of new threads < new proc, but still not free
• How much useful concurrency can you support?
  – If hardware support, # cores * # hyperthreads
  – Also, depends on I/O patterns of computation (e.g. how much threads pause on external I/O)
• So if you keep getting new connections
  – ...and keep creating new threads per connection
  – ...much faster than you can complete threads...
  – ...driven to livelock
Thread Pools

```java
master() {
    allocThreads(slave, queue);
    while (TRUE) {
        conn = accept();
        enqueue(queue, conn);
        wakeUp(queue);
    }
}

slave(queue) {
    while (TRUE) {
        conn = Dequeue(queue);
        if (conn == null) 
            sleepOn(queue);
        else 
            processReq(conn);
    }
}
```
Thread Pools

• How to choose how many threads are appropriate in your thread pool?
Parallelizing the NIC

• Where does master thread (recv) run?
• How is state transferred in multi-core machine?
• Higher performance if state doesn’t need to be xferred between cores/CPUs

• Multi-queue NIC
  – \( \sim O(100) \) queues in the NIC
  – Flow-hashing to map flows to NIC queues
  – One thread per (v)core to receive
  – Multiple threads per core to process
Writing Threads vs. Events

```python
conn = accept()
read(conn, inbuf, inlen)
webobj = parse_http(inbuf)
filefd = open(webobj.getLocalFilename())
if (filefd < 0)
    send(conn, 404Object())
else {
    send(conn, HttpHeaders())
    read(filefd, filebuf, len)
    while (len != EOF) {
        send(conn, filebuf, len)
        read(filefd, filebuf, len)
    }
}
close(filefd)
close(conn)
```
Writing Threads vs. Events

conn = accept()
read(conn, inbuf, inlen)
webobj = parse_http(inbuf)
filefd = open(webobj.getLocalFilename())
if (filefd < 0)
    send(conn, 404Object())
else {
    send(conn, HttpHeaders())
    read(filefd, filebuf, len)
    while (len != EOF) {
        send(conn, filebuf, len)
        read(fd, filebuf, len)
    }
}
close(filefd)
close(conn)
Async IO and event-based programming

• Events and async IO
  – select
  – Callbacks and stack ripping
  – State across callbacks (function currying)
  – libasync

• Async IO and multithreads
  – Boost’s asio strand model