Administrivia (1)

HW4 due next Thursday night, 11 pm w/usual late days
  ▪ How is it going?

Section tomorrow: concurrency and pthreads
  - Followed by a pthreads exercise to do over the weekend

Reminder: watch your late days! (4 max per quarter)
  - We should have the “late days remaining” gradebook entry updated in the next day or two
  - Pop quiz: What happens if you turn in something late and have no late days left?
Final exam is Wed. 3/19, 2:30 pm

- Topic list on web will be updated shortly if needed, but don’t expect much to change
- Old exams on the web already
- High-level course review in sections next week
- New! Last-minute review Tue. 3/18, 4:30 pm. Location TBA, bring questions!
Goals

Understand concurrency
- why it is useful
- why it is hard

Exposure to concurrent programming styles
- using multiple threads or multiple processes
- using asynchronous or non-blocking I/O
  - “event-driven programming”
Let’s imagine you want to...

...build a web search engine.

- you need a Web index
  ‣ an inverted index (a map from “word” to “list of documents containing the word”)
  ‣ probably sharded over multiple files
- a query processor
  ‣ accepts a query composed of multiple words
  ‣ looks up each word in the index
  ‣ merges the result from each word into an overall result set
Architecturally

index file

index file

index file

query processor

client

client

client

client

client
A sequential implementation

doclist Lookup(string word) {
   bucket = hash(word);
   hitlist = file.read(bucket);
   foreach hit in hitlist {
      doclist.append(file.read(hit));
   }
   return doclist;
}

main() {
   while (1) {
      string query_words[] = GetNextQuery();
      results = Lookup(query_words[0]);
      foreach word in query[1..n] {
         results = results.intersect(Lookup(word));
      }
      Display(results);
   }
}
Visually

```
main() GetNextQuery()
network I/O
Lookup()
file::read()
disk I/O
Lookup()
file::read()
disk I/O
Lookup()
file::read()
disk I/O
Intersect Results()
Display()
network I/O
GetNextQuery()
```
Simplifying
Simplifying

the CPU is idle most of the time

only one I/O request at a time is in flight

queries don’t run until earlier queries finish

queries don’t run until earlier queries finish
Sequentiality can be inefficient

Only one query is being processed at a time
- all other queries queue up behind the first one

The CPU is idle most of the time
- it is “blocked” waiting for I/O to complete
  - disk I/O can be very, very slow

At most one I/O operation is in flight at a time
- misses opportunities to speed I/O up
  - separate devices in parallel, better scheduling of single device, ...
What we want... concurrency

A version of the program that executes multiple tasks simultaneously

- it could execute multiple queries at the same time
  ‣ while one is waiting for I/O, another can be executing on the CPU

- or, it could execute queries one at a time, but issue IO requests against different files/disks simultaneously
  ‣ it could read from several different index files at once, processing the I/O results as they arrive

Concurrency ≠ parallelism

- parallelism is when multiple CPUs work simultaneously
One way to do this

Use multiple **threads** or **processes**

- as a query arrives, **fork** a new thread (or process) to handle it
  - the thread reads the query from the console, issues read requests against files, assembles results and writes to the console
  - the thread uses blocking I/O; the thread alternates between consuming CPU cycles and blocking on I/O
- the OS context switches between threads / processes
  - while one is blocked on I/O, another can use the CPU
  - multiple threads’ I/O requests can be issued at once
Multithreaded pseudocode

```cpp
main() {
    while (1) {
        string query_words[] = GetNextQuery();
        ForkThread(ProcessQuery());
    }
}

doclist Lookup(string word) {
    bucket = hash(word);
    hitlist = file.read(bucket);
    foreach hit in hitlist
        doclist.append(file.read(hit));
    return doclist;
}

ProcessQuery() {
    results = Lookup(query_words[0]);
    foreach word in query[1..n] {
        results = results.intersect(Lookup(word));
    }
    Display(results);
}
```
Multithreaded, visually

- **CPU 1.a**
- **CPU 1.b**
- **CPU 1.c**
- **CPU 1.d**
- **CPU 1.e**

- **CPU 2.a**
- **CPU 2.b**
- **CPU 2.c**
- **CPU 2.d**
- **CPU 2.e**

- **CPU 3.a**
- **CPU 3.b**
- **CPU 3.c**
- **CPU 3.d**
- **CPU 3.e**

- **I/O 1.a**
- **I/O 1.b**
- **I/O 1.c**
- **I/O 1.d**
- **I/O 1.e**

- **I/O 2.a**
- **I/O 2.b**
- **I/O 2.c**
- **I/O 2.d**
- **I/O 2.e**

- **I/O 3.a**
- **I/O 3.b**
- **I/O 3.c**
- **I/O 3.d**
- **I/O 3.e**

- **Query 1**
- **Query 2**
- **Query 3**

**Time**
Whither threads?

Advantages

- you (mostly) write sequential-looking code
- if you have multiple CPUs / cores, threads can run in parallel

Disadvantages

- if your threads share data, need locks or other synchronization
  ‣ this is very bug-prone and difficult to debug
- threads can introduce overhead
  ‣ lock contention, context switch overhead, and other issues
- need language support for threads
One alternative

Fork **processes** instead of threads

- advantages:
  - no shared memory between processes, so no need to worry about concurrent accesses to shared variables / data structures
  - no need for language support; OS provides “fork”

- disadvantages:
  - more overhead than threads to create, context switch
  - cannot easily share memory between processes, so typically share through the file system
Another alternative

Use **asynchronous** or **non-blocking** I/O

- your program begins processing a query
  - when your program needs to read data to make further progress, it registers interest in the data with the OS, then switches to a different query
  - the OS handles the details of issuing the read on the disk, or waiting for data from the console (or other devices, like the network)
  - when data becomes available, the OS lets your program know
- your program (almost never) blocks on I/O
Event-driven programming

Your program is structured as an event-loop

```c
void dispatch(task, event) {
    switch(task.state) {
        case READING_FROM_CONSOLE:
            query_words = event.data;
            async_read(index, query_words[0]);
            task.state = READING_FROM_INDEX;
            return;
        case READING_FROM_INDEX:
            ...etc.
    }
}

while(1) {
    event = OS.GetNextEvent();
    task = lookup(event);
    dispatch(task, event);
}
```
Asynchronous, event-driven
Non-blocking vs. asynchronous

Non-blocking I/O (network, console)
- your program enables non-blocking I/O on its fd’s
- your program issues read( ), write( ) system calls
  ▸ if the read/write would block, the system call returns immediately
- program can ask the OS which fd’s are readable/writeable
  ▸ program can choose to block while no fds are ready

Asynchronous I/O (disk)
- program tells the OS to begin reading / writing
  ▸ the “begin_read” or “begin_write” returns immediately
  ▸ when the I/O completes, OS delivers an event to the program
Why the difference?

Non-blocking I/O is for networks
- according to Linux, the disk never **blocks** your program
  ‣ it just delays it
- but, reading from the network can truly block your program
  ‣ a remote computer may wait arbitrarily long before sending data

Asynchronous I/O is for files
- primarily used to hide disk latency
  ‣ asynchronous I/O system calls are messy and complicated :(
  ‣ instead, typically use a threadpool to emulate asynchronous I/O
Whither events?

Advantages

- don’t have to worry about locks and “race conditions”
- for some kinds of programs, especially GUIs, leads to a very simple and intuitive program structure
  ‣ one event handler for each UI event

Disadvantages

- can lead to very complex structure for programs that do lots of disk and network I/O
  ‣ sequential code gets broken up into a jumble of small event handlers
  ‣ you have to package up all task state between handlers
One way to think about it

Threaded code:
- each thread executes its task sequentially, and per-task state is naturally stored in the thread’s stack
- OS and thread scheduler switch between threads for you

Event-driven code:
- *you* are the scheduler
- you have to bundle up task state into continuations; tasks do not have their own stacks
See you on Friday!