Concurrency: Intro and Threads
CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:
Dao Yi
Nathan Lipiarski
Yibo Cao

Farrell Fileas
Renshu Gu
Yifan Bai

Lukas Joswiak
Travis McGaha
Yifan Xu
About how long did Exercise 16 take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I prefer not to say
Administrivia

- HW4 due two Thursdays from now (12/05)
  - You can use at most ONE late day

- Short week next week:
  - Wed lecture cancelled (but OH available in AND 223 at 11:30)
  - 🦃 Fri holiday 🦃
Lecture Outline

❖ HTTP/2 Review
❖ From Query Processing to a Search Server
❖ Intro to Concurrency
❖ Threads
❖ Search Server with pthreads
HTTP/1.1 Feature: Persistent connections

❖ Establishing a TCP connection is costly
  ▪ Multiple network round trips to set up the TCP connection
  ▪ TCP has a feature called “slow start”; slowly grows the rate at which a TCP connection transmits to avoid overwhelming networks

❖ A web page consists of multiple objects and a client probably visits several pages on the same server
  ▪ Bad idea: separate TCP connection for each object
  ▪ Better idea: single TCP connection, multiple requests
HTTP/2 (2 of 3)

❖ Based on Google SPDY (2010) ; standardized in 2015

❖ Features:
  ▪ Same core request/response model (GET, POST, OK, ...)
  ▪ Binary protocol
    • Easier parsing by machines (harder for humans)
    • Sizes in headers, not discovered as requests are processed
    • Headers compressed and deduplicated by default!
  ▪ Multiple data steams multiplexed on single TCP connection
    • Fixes “head-of-line blocking”
    • With priorities on the streams!
  ▪ Server push and more...
Lecture Outline

❖ HTTP/2 Review
❖ From Query Processing to a Search Server
❖ Intro to Concurrency
❖ Threads
❖ Search Server with pthreads
Building a Web Search Engine

❖ We have:
  ▪ A web index
    • A map from *word* to *list of documents containing the word*
    • This is 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

❖ We need:
  ▪ Something that turns HTTP requests into well-formed queries
Search Engine Architecture

index file

index file

index file

query processor

HTTP Server

client
Search Engine (Pseudocode)

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

main() {
    SetupServerToReceiveConnections();
    while (1) {
        string query_words[] = GetNextQuery();
        results = Lookup(query_words[0]);
        foreach word in query[1..n] {
            results = results.intersect(Lookup(word));
        }
        Display(results);
    }
}
What About I/O-caused Latency?

- Jeff Dean’s “Numbers Everyone Should Know” (LADIS ‘09)
Execution Timeline: One multi-word query

- **main()**
  - **network I/O**
  - **GetNextQuery()**

- **Lookup()**
  - **file I/O**
  - **disk I/O**

- **Lookup()**
  - **file I/O**
  - **disk I/O**

- **Lookup()**
  - **file I/O**
  - **disk I/O**

- **results.intersect()**
  - **network I/O**
  - **Display()**

- **network I/O**
  - **GetNextQuery()**
Execution Timeline: To Scale

- CPU is busy in these tiny slivers
- CPU is idle in these colored blocks

network I/O

main()

disk I/O

disk I/O

disk I/O

network I/O
Multiple (Single-Word) Queries

Query 1

CPU 1.1
I/O 1.1
CPU 1.2
I/O 1.2
CPU 1.3
I/O 1.3
CPU 1.4
I/O 1.4
CPU 1.5
I/O 1.5

Query 2

CPU 2.1
I/O 2.1
CPU 2.2
I/O 2.2
CPU 2.3
I/O 2.3
CPU 2.4
I/O 2.4
CPU 2.5
I/O 2.5

Query 3

CPU 3.1
I/O 3.1
CPU 3.2
I/O 3.2
CPU 3.3
I/O 3.3
CPU 3.4
I/O 3.4
CPU 3.5
I/O 3.5

lookup() setup

getNextQuery() setup

getNextQuery() network

getNextQuery() disk

intersect() and display() execution

Time

14
Multiple Queries: To Scale

CPU is busy in these tiny slivers

query 1

query 2

network

I/O 1.b

I/O 1.d

I/O 1.b

I/O 1.d

I/O 1.b

I/O 1.d

time
Uh-Oh (1 of 2)

The CPU is idle most of the time! (picture not to scale)

Only one I/O request at a time is “in flight”

Queries don’t run until earlier queries finish

query 1

query 2

query 3

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

time

query 2

query 3

Queries don’t run until earlier queries finish

The CPU is idle most of the time!
Uh-Oh (2 of 2)
Sequential Can Be Inefficient

- Only one query is being processed at a time
  - All other queries queue up behind the first one
  - And clients queue up behind the queries ...

- Even while processing one query, the CPU is idle the vast majority of the time
  - It is *blocked* waiting for I/O to complete
    - Disk I/O can be very, very slow (10 million times slower ...)

- At most one I/O operation is in flight at a time
  - Missed opportunities to speed I/O up
    - Separate devices in parallel, better scheduling of a single device, etc.
Lecture Outline

❖ HTTP/2 Review
❖ From Query Processing to a Search Server
❖ **Intro to Concurrency**
❖ Threads
❖ Search Server with pthreads
Concurrenty

❖ Concurrency != parallelism

▪ Concurrency is doing multiple tasks at a time
▪ Parallelism is executing multiple CPU instructions simultaneously

❖ Our search engine could run concurrently:

▪ Example: Execute queries one at a time, but issue I/O requests against different files/disks simultaneously
  • Could read from several index files at once, processing the I/O results as they arrive

▪ Example: Our web server could execute multiple queries at the same time
  • While one is waiting for I/O, another can be executing on the CPU
A Concurrent Implementation

❖ Use multiple “workers”
  ▪ As a query arrives, create a new “worker” to handle it
    • The “worker” reads the query from the network, issues read requests against files, assembles results and writes to the network
    • The “worker” uses blocking I/O; the “worker” alternates between consuming CPU cycles and blocking on I/O
  ▪ The OS context switches between “workers”
    • While one is blocked on I/O, another can use the CPU
    • Multiple “workers”’ I/O requests can be issued at once

❖ So what should we use for our “workers”?
Lecture Outline

❖ From Query Processing to a Search Server
❖ Intro to Concurrency
❖ Threads
❖ Search Server with pthreads
Review: Processes

❖ To implement a “process”, the operating system gives us:
  - Resources such as file handles and sockets
  - Call stack + registers to support (eg, PC, SP)
  - Virtual memory (page tables, TLBs, etc...)

❖ If we want concurrency, what is the “minimal set” we need to execute a single line of code?

```
“Worker” 1
bucket = hash(word);
hitlist = file.read(bucket);

“Worker” 2
foreach hit in hitlist {
   doclist.append(file.read(hit));
}
```
Introducing Threads

❖ Separate the concept of a **process** from the “**thread of execution**”
  ▪ Usually called a **thread**, this is a sequential execution stream within a process

❖ In most modern OS’s:
  ▪ **Process**: address space, OS resources, security attributes
  ▪ **Thread**: stack, stack pointer, program counter, registers
  ▪ Threads are the *unit of scheduling* and processes are their *containers*; every process has at least one thread running in it
Threads

❖ Threads were formerly called “lightweight processes”
  ▪ They execute concurrently like processes
    • OS’s often treat them, not processes, as the unit of scheduling
    • Parallelism for free! If you have multiple CPUs/cores, can run them simultaneously
  ▪ Unlike processes, threads cohabitate the same address space
    • Threads within a process see the same heap and globals and can communicate with each other through variables and memory
      – But, they can interfere with each other – need synchronization for shared resources
    • Each thread has its own stack

❖ What does the OS do when you switch processes?
  ▪ How does that differ from switching threads?
Multi-threaded Search Engine (Pseudocode)

```plaintext
main() {
  while (1) {
    string query_words[] = GetNextQuery();
    CreateThread(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);
}
```
Multi-threaded Search Engine (Execution)

Still no parallelism!

(b) is network I/O and doesn’t overlap

(d) is disk I/O and doesn’t overlap either

CPU 3.a  I/O 3.b
CPU 3.c  I/O 3.d
CPU 3.e

CPU 2.a  I/O 2.b
CPU 2.c  I/O 2.d
CPU 2.e

CPU 1.a  I/O 1.b
CPU 1.c  I/O 1.d
CPU 1.e

query 1

query 2

query 3

Time
Single-Threaded Address Spaces

- Before creating a thread
  - One thread of execution running in the address space
    - One PC, stack, SP
  - That main thread invokes a function to create a new thread
    - Typically `pthread_create()`
Multi-threaded Address Spaces

- After creating a thread
  - *Two* threads of execution running in the address space
    - Original thread (parent) and new thread (child)
    - New stack created for child thread
    - Child thread has its own *values* of the PC and SP
  - Both threads share the other segments (code, heap, globals)
    - They can cooperatively modify shared data
Lecture Outline

❖ From Query Processing to a Search Server
❖ Intro to Concurrency
❖ Threads
❖ Search Server with pthreads
POSIX Threads (pthreads)

- The POSIX APIs for dealing with threads
  - Declared in `pthread.h`
    - Not part of the C/C++ language (cf. Java)
  - To enable support for multithreading, must include `-pthread` flag when compiling and linking with `gcc` command
Creating and Terminating Threads

- `int pthread_create(
  pthread_t* thread,
  const pthread_attr_t* attr,
  void* (*start_routine)(void*),
  void* arg);`

  - Creates a new thread into `*thread`, with attributes `*attr` (NULL means default attributes)
  - Returns 0 on success and an error number on error (can check against error constants)
  - The new thread runs `start_routine(arg)`

- `void pthread_exit(void* retval);`

  - Equivalent of `exit(retval);` for a thread instead of a process
  - The thread will automatically exit once it returns from `start_routine()`
What To Do After Forking Threads?

- `int pthread_join(pthread_t thread, void** retval);`
  - Waits for the thread specified by `thread` to terminate
  - The thread equivalent of `waitpid()`
  - The exit status of the terminated thread is placed in `**retval`

- `int pthread_detach(pthread_t thread);`
  - Mark thread specified by `thread` as detached – it will clean up its resources as soon as it terminates
Multi-threaded Search Engine: Architecture

- A single *process* handles all of the connections, but a parent *thread* dispatches (creates) a new thread to handle each connection
  - The child thread handles the new connection and subsequent I/O, then exits when the connection terminates

- See `searchserver_threads/` for code if curious
Multi-threaded Search Engine: Request Flow

![Diagram showing a client connecting to a server via accept()]
Multi-threaded Search Engine: Request Flow

client

server

pthread_create()

pthread_detach()
Multi-threaded Search Engine: Request Flow

client

client

server

accept()
Multi-threaded Search Engine: Request Flow
Multi-threaded Search Engine: Request Flow
pthread Examples

- **pthread.c**: pthreads in C
- **pthread.cc**: Same, but in C++
- **searchserver_threads**: Non-trivial example

Things to keep in mind while reading:

- More instructions per thread = higher likelihood of interleaving
- How do you handle memory management?
  - Who allocates and deallocates memory?
  - Can two threads call `new` at the same time?
- When calling **pthread_create()**, `start_routine` points to a function that takes only one argument (a `void*`)
  - To pass complex arguments into the thread, create a struct to bundle the necessary data
Why Threads? (1 of 2)

❖ Advantages:
   ▪ Almost as simple to code as sequential
     • In fact, most of the code is identical! (but a bit more complicated to dispatch a thread)
   ▪ Threads can run in parallel if you have multiple CPUs/cores
   ▪ Concurrent execution with good CPU and network utilization
     • Some overhead, but less than processes
   ▪ Shared-memory communication is possible
Why Threads? (2 of 2)

❖ Disadvantages:

▪ Need language and OS support for threads
▪ If threads share data, you need locks or other synchronization
  • See next lecture: Very bug-prone and difficult to debug
▪ Threads can introduce overhead
  • See next lecture: Lock contention, context switch overhead, CPU thrashing, and other issues
  • Also cognitive overhead for future programmers!
▪ Threads within the same process have a “shared fate”
  • Eg, shared file-handle limits, no crash isolation, etc.
Why Sequential?

❖ Advantages:
   ▪ Simple to write, maintain, debug
   ▪ The default, supported everywhere

❖ Disadvantages:
   ▪ Depending on application, poor performance
     • One slow client will cause all others to block
     • Poor utilization of resources (CPU, network, disk)