Concurrency: Intro and Threads
CSE 333 Winter 2020

Guest Instructor: Travis McGaha

Teaching Assistants:
Andrew Hu
Cheng Ni
Guramrit Singh
Rehaan Bhimani
Zachary Keyes
Austin Chan
Cosmo Wang
Mengqi Cen
Renshu Gu
Brennan Stein
Diya Joy
Pat Kosakanchit
Travis McGaha
Administrivia

❖ HW4 due two Thursdays from now (03/12)
  ▪ You can use two late days on HW4.

❖ Exercise 17 to be released Friday.
  ▪ Due Monday 3/09 @ 11 am
  ▪ 🎉 The Last Exercise 🎉
Some Common HW4 Bugs

❖ Your server works, but is really, really slow
  ▪ Check the 2\textsuperscript{nd} argument to the \texttt{QueryProcessor} \texttt{constructor}

❖ Funny things happen after the first request
  ▪ Make sure you’re not destroying the \texttt{HTTPConnection} \texttt{object}
    too early (\textit{e.g.} falling out of scope in a while loop)

❖ Server crashes on a blank request
  ▪ Make sure that you handle the case that \texttt{read()} (or \texttt{WrappedRead()}) returns 0
Lecture Outline

❖ From Query Processing to a Search Server
❖ Intro to Concurrency
❖ Threads and other concurrency methods
❖ 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
Search Engine Architecture

index file

index file

index file

query processor

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);
    }
}
Execution Timeline: a Multi-Word Query

- GetNextQuery
- network I/O
- Lookup
- disk I/O
- Lookup
- disk I/O
- results.intersect
- CPU
- Lookup
- disk I/O
- results.intersect
- CPU
- Display
- network I/O
- GetNextQuery

query

time
What About I/O-caused Latency?

- Jeff Dean’s “Numbers Everyone Should Know” (LADIS ‘09)

```
Numbers Everyone Should Know

L1 cache reference       0.5 ns
Branch mispredict         5 ns
L2 cache reference        7 ns
Mutex lock/unlock         100 ns
Main memory reference     100 ns
Compress 1K bytes with Zippy 10,000 ns
Send 2K bytes over 1 Gbps network 20,000 ns
Read 1 MB sequentially from memory 250,000 ns
Round trip within same datacenter 500,000 ns
Disk seek                      10,000,000 ns
Read 1 MB sequentially from network 10,000,000 ns
Read 1 MB sequentially from disk 30,000,000 ns
Send packet CA->Netherlands->CA 150,000,000 ns
```
Execution Timeline: To Scale
Multiple (Single-Word) Queries

# is the Query Number
#.a -> GetNextQuery()
#.b -> network I/O
#.c -> Lookup() & file.read()
#.d -> Disk I/O
#.e -> Intersect() & Display()
Multiple Queries: To Scale

query 1

query 2

query 3

time
Uh-Oh (1 of 2)

index file

index file

index file

query processor

client

client

client

client

client
Uh-Oh (2 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
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

❖ From Query Processing to a Search Server
❖ Intro to Concurrency
❖ Concurrent Programming Styles
❖ Threads
❖ Search Server with pthreads
Concurrency

❖ 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

❖ Concurrency != parallelism
  ▪ Concurrency is doing multiple tasks at a time
  ▪ Parallelism is executing multiple CPU instructions *simultaneously*
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 and other concurrency methods**
❖ Search Server with pthreads
Review: Processes

❖ The components of a “process” are:
  ▪ Resources such as file descriptors and sockets
  ▪ An address space (page tables, etc.)

❖ Different Processes have independent components:
  ▪ Most importantly: Isolated address spaces.

❖ An address space of a process can hold stack(s) that distinguish different “threads” of execution
Introducing Threads

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

❖ In most modern OS’s:
  ▪ Threads are the unit of scheduling.
Multi-threaded Search Engine (Pseudocode)

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

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

ProcessQuery(string query_words[]) {
    results = Lookup(query_words[0]);
    foreach word in query[1..n]
        results = results.intersect(Lookup(word));
    Display(results);
}
```
Multi-threaded Search Engine (Execution)

CPU

CPU 1: a
CPU 1: c
CPU 1: e
CPU 2: a
CPU 2: c
CPU 3: a
CPU 3: c
CPU 3: e
I/O 1: b
I/O 2: b
I/O 3: b
I/O 2: d
I/O 1: d
I/O 3: d

query 1
query 2
query 3

time
Why Threads?

❖ Advantages:
  ▪ You (mostly) write sequential-looking code
  ▪ Threads can run in parallel if you have multiple CPUs/cores

❖ Disadvantages:
  ▪ If threads share data, you need locks or other synchronization
    • Very bug-prone and difficult to debug
  ▪ Threads can introduce overhead
    • Lock contention, context switch overhead, and other issues
  ▪ Need language support for threads
Threads vs. Processes

❖ In most modern OS’s:
   ▪ A Process has a unique: address space, OS resources, & security attributes
   ▪ A Thread has a unique: 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 vs. Processes

OS kernel [protected]

Stack_{parent}

Shared Libraries

Heap (malloc/free)

Read/Write Segments .data, .bss

Read-Only Segments .text, .rodata

pthread_create()

OS kernel [protected]

Stack_{parent}

Stack_{child}

Shared Libraries

Heap (malloc/free)

Read/Write Segments .data, .bss

Read-Only Segments .text, .rodata
Threads vs. Processes

OS kernel [protected]
- Stack\textsubscript{parent}
- Shared Libraries
- Heap (malloc/free)
- Read/Write Segments \textit{.data, .bss}
- Read-Only Segments \textit{.text, .rodata}

OS kernel [protected]
- Stack\textsubscript{parent}
- Shared Libraries
- Heap (malloc/free)
- Read/Write Segments \textit{.data, .bss}
- Read-Only Segments \textit{.text, .rodata}

OS kernel [protected]
- Stack\textsubscript{child}
- Shared Libraries
- Heap (malloc/free)
- Read/Write Segments \textit{.data, .bss}
- Read-Only Segments \textit{.text, .rodata}

\textbf{fork()}

Alternative: Processes

❖ What if we forked processes instead of threads?

❖ Advantages:
  ▪ No shared memory between processes
  ▪ No need for language support; OS provides “fork”
  ▪ Processes are isolated. If one crashes, other processes keep going

❖ Disadvantages:
  ▪ More overhead than threads during creation and context switching
  ▪ Cannot easily share memory between processes – typically communicate through the file system
Alternate: Different I/O Handling

- 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 and 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
Non-blocking I/O

❖ Reading from the network can truly *block* your program
  ▪ Remote computer may wait arbitrarily long before sending data

❖ Non-blocking I/O (network, console)
  ▪ Your program enables non-blocking I/O on its file descriptors
  ▪ Your program issues `read()` and `write()` system calls
    • If the read/write would block, the system call returns immediately
  ▪ Program can ask the OS which file descriptors are readable/writeable
    • Program can choose to block while no file descriptors are ready
Outline (next two lectures)

- We’ll look at different `searchserver` implementations
  - Sequential
  - Concurrent via dispatching threads – `pthread_create()`
  - Concurrent via forking processes – `fork()`
    - 🎉 Lecture With Andrew Hu! 🎉

Sequential

- Pseudocode:

```c
listen_fd = Listen(port);

while (1) {
    client_fd = accept(listen_fd);
    buf = read(client_fd);
    resp = ProcessQuery(buf);
    write(client_fd, resp);
    close(client_fd);
}
```

- See `searchserver_server_sequential/`
Why Sequential?

❖ Advantages:
  ▪ Super(?) simple to build/write

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

❖ Threads are like lightweight processes
  ▪ They execute concurrently like processes
    • Multiple threads can run simultaneously on multiple CPUs/cores
  ▪ 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
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
    - `gcc -g -Wall -std=c11 -pthread -o main main.c`
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()`