About how long did Exercise 10 take you?

A. [0, 2) hours
B. [2, 4) hours
C. [4, 6) hours
D. [6, 8) hours
E. 8+ Hours
F. I didn’t submit / I prefer not to say
Introduction to Concurrency
CSE 333 Fall 2023

Instructor: Chris Thachuk

Teaching Assistants:

Ann Baturytski   Humza Lala
                Alan Li
Noa Ferman       Leanna Mi Nguyen
James Froelich   Chanh Truong
Hannah Jiang     Deeksha Vatwani
Yegor Kuznetsov  Jennifer Xu
Relevant Course Information

- Homework 3 due tomorrow (11/23) by 10pm
  - *Help during holidays is unlikely so seek help by today*

- Homework 4 out, due 12/6
  - Open private Ed post if missing hw4 starter files

- Exercise 11 due Monday by 10pm
  - Can use ex10 posted solutions

- No Thursday section or Friday lecture this week
  - Make time to rest and pursue something fun over the holiday!
Homework 4 Summary

- **Build a Multithreaded Web Server (333gle)**
  - You will host the querying service that you built in your previous homework on a web server

- **Running your server**
  - `.http333d <port> <static files> <unit indices>`
  - Static files are the files on disk corresponding to our index files
  - You (and others) can access it on any browser now!

- **Implementation**
  - Using network protocols to communicate between client/server
  - Handling some additional security flaws
  - Note: Multithreading is already implemented for you
Some Common HW4 Bugs

- Your server works, but is really, really slow
  - Check the 2nd argument to the `QueryProcessor` constructor

- Funny things happen after the first request
  - Make sure you’re not destroying the `HTTPConnection` object too early (e.g., falling out of scope in a while loop)

- Server crashes on a blank request
  - Make sure that you handle the case that `read()` (or `WrappedRead()`) returns 0
Lecture Outline

- From Query Processing to a Search Server
- Concurrency and Concurrency Methods
Building a Web Search Engine

- We have:
  - Some indexes
    - 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
Sequential 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);
    }
}

See searchserver_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)
Execution Timeline: a Multi-Word Query

- main()
- 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()
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: (Loosely) To Scale

network I/O  
disk I/O  
disk I/O  
disk I/O  
network I/O

main()  
disk I/O  
CPU  
disk I/O  
CPU  

...
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: (Loosely) To Scale
Sequential Issues

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
- Concurrency and Concurrency Methods
Concurrency

- Concurrency ≠ parallelism
  - Concurrency is working on multiple tasks with overlapping execution times
  - Parallelism is executing multiple CPU instructions *simultaneously*

- Our search engine could run concurrently in multiple different ways:
  - **Example:** 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:** 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 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”?
Worker Option 1: Processes (Review)

- Processes can fork “cloned” processes that have a parent-child relationship
  - Work almost entirely independent of each other

- The major components of a process are:
  - An address space to hold data and instructions
  - Open resources such as file descriptors
  - Current state of execution
    - Includes values of registers (including program counter and stack pointer) and parts of memory (the Stack, in particular)
Why Processes?

❖ Advantages:

  ▪ Processes are isolated from one another
    • No shared memory between processes
    • If one crashes, the other processes keep going
  ▪ No need for language support (OS provides fork)

❖ Disadvantages:

  ▪ A lot of overhead during creation and context switching
  ▪ Cannot easily share memory between processes – typically must communicate through the file system
Worker Option 2: Threads

- From within a process, we can separate out the concept of a “thread of execution” (thread for short)
  - Processes are the containers that hold shared resources and attributes
    - e.g., address space, file descriptors, security attributes
  - Threads are independent, sequential execution streams (units of scheduling) within a process
    - e.g., stack, stack pointer, program counter, registers
## Threads vs. Processes

<table>
<thead>
<tr>
<th>OS kernel [protected]</th>
<th>Stack&lt;sub&gt;parent&lt;/sub&gt;</th>
<th>Stack&lt;sub&gt;child&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stack</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Shared Libraries</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Heap (malloc/free)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Read/Write Segments</strong></td>
<td><code>.data, .bss</code></td>
<td><code>.data, .bss</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Read-Only Segments</strong></td>
<td><code>.text, .rodata</code></td>
<td><code>.text, .rodata</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `fork()`: Creates a new process (child) from an existing process (parent).

- `SP_{parent}`, `PC_{parent}`, `SP_{child}`, `PC_{child}`: Stack Pointer and Program Counter for parent and child processes.
Threads vs. Processes

OS kernel [protected]

Stack\textsubscript{parent}

\hspace{1cm}

\text{Stack}\textsubscript{parent}

\hspace{1cm}

\text{Stack}\textsubscript{child}

\hspace{1cm}

\text{Shared Libraries}

\hspace{1cm}

\text{Shared Libraries}

\hspace{1cm}

\text{Heap (malloc/free)}

\hspace{1cm}

\text{Heap (malloc/free)}

\hspace{1cm}

\text{Read/Write Segments}

\text{\texttt{.data, .bss}}

\hspace{1cm}

\text{Read/Write Segments}

\text{\texttt{.data, .bss}}

\hspace{1cm}

\text{Read-Only Segments}

\text{\texttt{.text, .rodata}}

\hspace{1cm}

\text{Read-Only Segments}

\text{\texttt{.text, .rodata}}

\hspace{1cm}

\text{pthread\_create()}

\hspace{1cm}

\text{pthread\_create()}

\hspace{1cm}

SP\textsubscript{parent}

\hspace{1cm}

SP\textsubscript{parent}

\hspace{1cm}

SP\textsubscript{child}

\hspace{1cm}

SP\textsubscript{child}

\hspace{1cm}

PC\textsubscript{parent}

\hspace{1cm}

PC\textsubscript{child}

\hspace{1cm}

PC\textsubscript{parent}

\hspace{1cm}

PC\textsubscript{parent}
Multi-threaded Search Engine (Pseudocode)

```c
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);
}
```

All we did was put the code into a function, and create a thread that invokes it!
Multi-threaded Search Engine (Execution)

Note how only one thread uses any specific resource at a time.

The OS schedules all of this for us! 😊

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

query 1
query 2
query 3

---

time
Why Threads?

- **Advantages:**
  - You (mostly) write sequential-looking code
  - Less overhead than processes during creation and context switching
  - 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
Alternate: 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
Alternate: Asynchronous I/O

- Using **asynchronous** I/O, your program (almost never) \textit{blocks} on 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 by delivering an \textit{event}
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:
            ...
    }
}

while (1) {
    event = OS.GetNextEvent();
    task = lookup(event);
    dispatch(task, event);
}
```
Asynchronous, Event-Driven

# is the Query Number
#.a -> GetNextQuery()
#.b -> network I/O
#.c -> Lookup() & file.read()
#.d -> Disk I/O
#.e -> Intersect() & Display()
Why 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
Outline (next two lectures)

- We’ll look at different searchserver implementations
  - Concurrent via dispatching threads – `pthread_create()`
  - Concurrent via forking processes – `fork()`