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
Relevant Course Information

- Homework 4 due 1 week from tomorrow (3/9)
  - Partner form due end of tomorrow
  - You can still use **two** late days (until Sunday, 3/12)

- Exercise 11 due Friday
- Exercise 12 (the last exercise™) to be released Friday
  - Consumer-producer concurrency
  - Due Wednesday 3/8 @ 11 am

- Final Exam (Monday, 3/13 – Wednesday, 3/15)
  - Same policies as the midterm
  - ex8-ex12, hw3-hw4, overall course questions
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 \texttt{read()} (or \texttt{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

- Query processor
- Client
- Index file
- Index file
- Index file
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);
    }
}
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

3-word query: ocean whale ravenous

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

main() GetNextQuery() Lookup() Lookup() results.intersect() Lookup() Display() GetNextQuery()
What About I/O-caused Latency?

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

<table>
<thead>
<tr>
<th>Task</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5 ns</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5 ns</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7 ns</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>100 ns</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 ns</td>
</tr>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>10,000 ns</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>20,000 ns</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000 ns</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000 ns</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000 ns</td>
</tr>
<tr>
<td>Read 1 MB sequentially from network</td>
<td>10,000,000 ns</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>30,000,000 ns</td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000 ns</td>
</tr>
</tbody>
</table>
Execution Timeline: (Loosely) To Scale

network I/O

main()

disk I/O

disk I/O

CPU

disk I/O

CPU

network I/O

...
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”

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

Queries don’t run until earlier queries finish

query 1

query 2

query 3

time

The CPU is idle most of the time!
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
Concurrenty

- 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

- **Stack**
  - Parent
  - Child

- **Shared Libraries**

- **Heap** (malloc/free)

- **Read/Write Segments**
  - .data, .bss

- **Read-Only Segments**
  - .text, .rodata

- **OS kernel** [protected]
  - Parent
  - Child

- **fork()**

- **PC**
  - Parent
  - Child
Threads vs. Processes

OS kernel [protected]

Stack\textsubscript{parent}

Shared Libraries

Heap (malloc/free)

Read/Write Segments \texttt{.data, .bss}

Read-Only Segments \texttt{.text, .rodata}

--

OS kernel [protected]

Stack\textsubscript{parent}

Shared Libraries

Heap (malloc/free)

Read/Write Segments \texttt{.data, .bss}

Read-Only Segments \texttt{.text, .rodata}

\texttt{pthread_create()}

\textbf{parent}

\textbf{child}

SP\textsubscript{parent}

PC\textsubscript{parent}

SP\textsubscript{child}

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

```java
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

no overlap! CPU has to switch between threads.

query 1 (thread 1)
query 2 (thread 2)
query 3 (thread 3)
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 with `select()` or `poll()`
    - Program can choose to block while no file descriptors are ready
Alternate: Asynchronous I/O

- Using asynchronous I/O, your program (almost never) 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 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);
}
```

- what we do depends on where we are in program (`state`) and what event came in.
- asynchronous notice to OS
- OS sends events back to process as they occur/finish
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 search server implementations
  - Concurrent via dispatching threads – `pthread_create()`
  - Concurrent via forking processes – `fork()`