OS and Parallelism
Review: Information Retrieval

• IR is the problem of finding **relevant documents** among an **index** that relate to some **query**

• We can rank documents by language features (e.g., tf-idf vectors) and **compare vectors of documents**

• We can rank documents based on their relative **prestige** using **PageRank**

• We can construct an **inverted index** by mapping **keywords** to **lists of documents**

• **Web Search** can be thought of as a **distributed task** in which the **index** can be split among multiple **workers**
Segment by document (divide cols)

<table>
<thead>
<tr>
<th>as</th>
<th>#docs</th>
<th>docid_0</th>
<th>docid_1</th>
<th>docid_2</th>
<th>...</th>
<th>docid_{#docs-1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>britney</td>
<td>#docs</td>
<td>docid_0</td>
<td>docid_1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cities</td>
<td>#docs</td>
<td>docid_0</td>
<td>docid_1</td>
<td>docid_2</td>
<td>docid_3</td>
<td></td>
</tr>
<tr>
<td>friendly</td>
<td>#docs</td>
<td>docid_0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>give</td>
<td>#docs</td>
<td>docid_0</td>
<td>docid_1</td>
<td>docid_2</td>
<td>...</td>
<td>docid_{#docs-1}</td>
</tr>
<tr>
<td>mayors</td>
<td>#docs</td>
<td>docid_0</td>
<td>docid_1</td>
<td>docid_2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nancy</td>
<td>#docs</td>
<td>docid_0</td>
<td>docid_1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>seattle</td>
<td>#docs</td>
<td>docid_0</td>
<td>docid_1</td>
<td>docid_2</td>
<td>...</td>
<td>docid_{#docs-1}</td>
</tr>
<tr>
<td>such</td>
<td>#docs</td>
<td>docid_0</td>
<td>docid_1</td>
<td>docid_2</td>
<td>...</td>
<td>docid_{#docs-1}</td>
</tr>
<tr>
<td>words</td>
<td>#docs</td>
<td>docid_0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Segment by document
### Segment by term (divide rows)

<table>
<thead>
<tr>
<th>Term</th>
<th>#docs</th>
<th>docid₀</th>
<th>docid₁</th>
<th>docid₂</th>
<th>...</th>
<th>docid #:docs-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>as</td>
<td>#docs</td>
<td>docid₀</td>
<td>docid₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>britney</td>
<td>#docs</td>
<td>docid₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cities</td>
<td>#docs</td>
<td>docid₀</td>
<td>docid₁</td>
<td>docid₂</td>
<td>docid₃</td>
<td></td>
</tr>
<tr>
<td>friendly</td>
<td>#docs</td>
<td>docid₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>give</td>
<td>#docs</td>
<td>docid₀</td>
<td>docid₁</td>
<td>docid₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mayors</td>
<td>#docs</td>
<td>docid₀</td>
<td>docid₁</td>
<td>docid₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nancy</td>
<td>#docs</td>
<td>docid₀</td>
<td>docid₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>seattle</td>
<td>#docs</td>
<td>docid₀</td>
<td>docid₁</td>
<td>docid₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>such</td>
<td>#docs</td>
<td>docid₀</td>
<td>docid₁</td>
<td>docid₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>words</td>
<td>#docs</td>
<td>docid₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Segment by term

- Docs 0-5M A-B
- Docs 0-5M C-D
- Docs 0-5M E-F
- Docs 0-5M G-H
- Docs 0-5M H-I

Ds 1, 29, ...

britney,
Review: Deduplication

• We deduplicate portions of the index to save re-computation and storage requirements

• The Jaccard similarity coefficient is a way of comparing sets
  • Each document can be broken up into shingles, which are $k$-long sequences of tokens in each document
  • Thus, each document can be represented by a set of shingles that can be used to compare documents for duplication

• The MinHash algorithm leverages statistical properties of sets to estimate the Jaccard coefficient
Jaccard similarity coefficient

• Jaccard similarity coefficient compares the similarity of the two sets of shingles (A and B)
• Size of the intersection / size of the union

\[ J(A, B) = \frac{|A \cap B|}{|A \cup B|}. \]

• 0 for disjoint sets, 1 for equal sets

What is the complexity of computing Jaccard?
• Assume A and B are size O(N)
One-Slide Summary: Processes, Threads, Sockets

• A **process** is a data structure used by the operating system to manage the execution of a program
  • Processes have their own address space, file handles, networking, etc.

• Processes can create one or more **threads** to execute tasks in parallel
  • Example: UI thread that runs separately from computation
  • Example: A Webserver with thousands of concurrent clients connected
  • Threads share the same address space as the host process

• Networked applications can use **sockets** to communicate with each other
  • Server binds, listens, accepts, recvs.  Client connects and sends
Project 4: MapReduce

• MapReduce: divide computation among many computers

• P4: simulate this on one computer
  • Multiple programs (processes): master and workers
  • Multiple threads: do computation at same time as waiting for network
 Processes

- Process: a running program

<table>
<thead>
<tr>
<th>Name</th>
<th>Status</th>
<th>CPU</th>
<th>Memory</th>
<th>Disk</th>
<th>Network</th>
<th>GPU</th>
<th>GPU engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NET Runtime Optimization Service (32 bit)</td>
<td></td>
<td>6.2%</td>
<td>11.3 MB</td>
<td>0.4 MB/s</td>
<td>0 Mbps</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>ipoint.exe</td>
<td></td>
<td>1.8%</td>
<td>1.7 MB</td>
<td>0 MB/s</td>
<td>0 Mbps</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Desktop Window Manager</td>
<td></td>
<td>1.7%</td>
<td>150.4 MB</td>
<td>0.1 MB/s</td>
<td>0 Mbps</td>
<td>1.2%</td>
<td>GPU 0 - 3D</td>
</tr>
<tr>
<td>Service Host: Bluetooth Support Service</td>
<td></td>
<td>1.5%</td>
<td>2.7 MB</td>
<td>0 MB/s</td>
<td>0 Mbps</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Snipping Tool</td>
<td></td>
<td>1.2%</td>
<td>3.5 MB</td>
<td>0.3 MB/s</td>
<td>0 Mbps</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Windows Driver Foundation - User-mode Driver Frame...</td>
<td></td>
<td>0.7%</td>
<td>1.9 MB</td>
<td>0 MB/s</td>
<td>0 Mbps</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Task Manager</td>
<td></td>
<td>0.7%</td>
<td>24.2 MB</td>
<td>0 MB/s</td>
<td>0 Mbps</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Client Server Runtime Process</td>
<td></td>
<td>0.5%</td>
<td>1.7 MB</td>
<td>0 MB/s</td>
<td>0 Mbps</td>
<td>0.5%</td>
<td>GPU 0 - 3D</td>
</tr>
<tr>
<td>Windows Explorer</td>
<td></td>
<td>0.5%</td>
<td>48.4 MB</td>
<td>0.1 MB/s</td>
<td>0 Mbps</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>System interrupts</td>
<td></td>
<td>0.4%</td>
<td>0 MB</td>
<td>0 MB/s</td>
<td>0 Mbps</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Cortana</td>
<td></td>
<td>0.3%</td>
<td>96.2 MB</td>
<td>0.9 MB/s</td>
<td>0.2 Mbps</td>
<td>0%</td>
<td>GPU 0 - 3D</td>
</tr>
<tr>
<td>Antimalware Service Executable</td>
<td></td>
<td>0.3%</td>
<td>117.1 MB</td>
<td>0.1 MB/s</td>
<td>0 Mbps</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Runtime Broker</td>
<td></td>
<td>0.3%</td>
<td>7.1 MB</td>
<td>0.1 MB/s</td>
<td>0 Mbps</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Windows Driver Foundation - User-mode Driver Frame...</td>
<td></td>
<td>0.2%</td>
<td>2.0 MB</td>
<td>0 MB/s</td>
<td>0 Mbps</td>
<td>0%</td>
<td>GPU 0 - 3D</td>
</tr>
<tr>
<td>System</td>
<td></td>
<td>0.2%</td>
<td>0.1 MB</td>
<td>0.6 MB/s</td>
<td>0 Mbps</td>
<td>0.1%</td>
<td>GPU 0 - 3D</td>
</tr>
<tr>
<td>Windows Security Health Service</td>
<td></td>
<td>0.1%</td>
<td>2.7 MB</td>
<td>0 MB/s</td>
<td>0 Mbps</td>
<td>0%</td>
<td></td>
</tr>
</tbody>
</table>
The process abstraction

• The process is the OS abstraction for execution
  • Also sometimes called a job or a task

• A process is a program in execution
  • Programs are static entities with potential for execution

• Process consists of:
  • A unique process ID (PID)
  • An address space (memory)
  • 1 or more threads (sequences of computation)
  • Some other resources (file handles, open sockets,...)
What does the OS do?

• Create abstractions to make hardware easier to use
  • Abstractions “nicer” than raw hardware interfaces
• Manage shared hardware resources
Operating system abstractions

Applications

Process  File system  Virtual mem  Sockets

Operating System

CPU  Disk  RAM  Network
The process abstraction

• What interface does hardware provide?
  • Hardware interface: single computer (CPU & memory), executing instructions from a mix of many different applications

• What interface does OS provide?
  • OS interface: several dedicated “computers” (one per application process)

• Process interface differs from hardware interface
from multiprocessing import Process
from time import sleep

def worker(worker_id):
    print("Hello from process {}".format(worker_id))
    sleep(10)

if __name__ == "__main__":
    for i in range(3):
        p = Process(target=worker, args=(i,))
        p.start()

$ python3 test_multiprocessing.py
Hello from process 0
Hello from process 1
Hello from process 2
Example

$ python3 test_multiprocessing.py
Hello from process 0
Hello from process 1
Hello from process 2

$ pgrep -lf test_multiprocessing  # macOS
$ pgrep -af test_multiprocessing  # Linux/WSL
16571 python test_multiprocessing.py
16572 python test_multiprocessing.py
16573 python test_multiprocessing.py
16574 python test_multiprocessing.py
Example

$ python3 test_multiprocessing.py
Hello from process 0
Hello from process 1
Hello from process 2

$ pgrep -lf test_multiprocessing  # macOS
$ pgrep -af test_multiprocessing  # Linux/WSL
16571 python test_multiprocessing.py
16572 python test_multiprocessing.py
16573 python test_multiprocessing.py
16574 python test_multiprocessing.py

4 separate processes
Agenda

• Processes
• **Threads**
• Synchronization
  • Atomic operations
• Sockets
Threads

• A thread is a sequential set of executing instructions
• Active object: describes the execution itself!
• Unit of concurrency
  • How many computations can happen at once?
  • Maximum: # of threads in the program
• Each thread has
  • The program counter (PC) indicating the next instruction
  • A set of general-purpose registers with current values
Thread example

```python
# test_threading.py
from threading import Thread
from time import sleep

def worker(worker_id):
    print("Hello from thread {}".format(worker_id))
    sleep(10)

if __name__ == "__main__":
    for i in range(3):
        t = Thread(target=worker, args=(i,))
        t.start()

$ python3 test_threading.py
Hello from thread 0
Hello from thread 1
Hello from thread 2
```
Thread example

```
$ python3 test_threading.py
Hello from thread 0
Hello from thread 1
Hello from thread 2

$ pgrep -lf test_threading  # macOS
$ pgrep -af test_threading  # Linux/WSL
  16976 python3 test_threading.py

$ ps -M 16976   # macOS
$ ps -m 16976  # Linux/WSL
awdeorio  17006 s001  0.0 S   31T  0:00.01  0:00.02
           17006  56.5 R   31T  0:18.65  0:10.97
           17006  56.2 R   31T  0:18.53  0:10.94
           17006  56.9 R   31T  0:18.71  0:11.17
```
Thread example

$ python3 test_threading.py
Hello from thread 0
Hello from thread 1
Hello from thread 2

$ pgrep -lf test_threading # macOS
$ pgrep -af test_threading # Linux/WSL
16976 python3 test_threading.py

$ ps -M 16976 # macOS
$ ps -m 16976 # Linux/WSL
awdeorio 17006 s001 0.0 S 31T 0:00.01 0:00.02
17006 56.5 R 31T 0:18.65 0:10.97
17006 56.2 R 31T 0:18.53 0:10.94
17006 56.9 R 31T 0:18.71 0:11.17

Stack (thread 1)
Stack (thread 2)
Stack (thread 3)
Stack (thread 4)
Heap
Static Data
Code
Option 1: One request at a time

• Example execution schedule:
  • Request 1 arrives
  • Server receives request 1
  • Server starts disk I/O 1
  • Request 2 arrives
  • Server waits for I/O 1 to finish

• Easy to program, but slow
  • Why slow?
    • Can’t overlap disk requests with computation, or with network receives
Option 2: event-driven web server

• Asynchronous I/O: issue I/O, but don’t wait for them to complete
  • Request 1 arrives
  • Server receives request 1
  • Server starts disk I/O 1 to satisfy request 1
  • Request 2 arrives
  • Server receives request 2
  • Server starts disk I/O 2 to satisfy request 2
  • Disk I/O 1 finishes

• Fast, but complicated to program
  • Keep track of multiple requests, what stage they’re in, outstanding disk I/O, outstanding network I/O
Option 3: multi-threaded web server

• One thread per request
  • Thread issues disk (or n/w) I/O, then waits for it to finish
  • Though thread is blocked on I/O, other threads can run
Option 3: multi-threaded web server

• One thread per request
  • Thread issues disk (or n/w) I/O, then waits for it to finish
  • Though thread is **blocked** on I/O, other threads can run
  • Where is the state of each request stored?

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request 1 arrives</td>
<td>Request 2 arrives</td>
<td>Request 3 arrives</td>
</tr>
<tr>
<td>Receive request 1</td>
<td>Receive request 2</td>
<td>Receive request 3</td>
</tr>
<tr>
<td>Start disk I/O 1</td>
<td>Start disk I/O 2</td>
<td></td>
</tr>
</tbody>
</table>

Disk I/O 1 finishes
Continue handling request 1
Benefits of threads

• Simpler programming model
  • The illusion of a dedicated CPU per thread

• Thread manager takes care of CPU sharing
  • Other threads can progress when one thread issues blocking I/O
  • Private state for each thread, but memory shared with process
Cooperating threads

• How can multiple threads cooperate on a single task?
  • Example: Ticketmaster’s webserver
  • Assume each thread has a dedicated processor

• Problem:
  • Ordering of events across threads is non-deterministic
  • Speed of each processor is unpredictable

Thread A ------------------------------------------->
Thread B - - - - - - - - - - - - - - - - - - - - - - - ->
Thread C - - - - - - - - - - - - - - - - - - - - - - - ->

• Consequences:
  • Many possible global ordering of events
  • Some may produce incorrect results
Non-deterministic ordering → Non-deterministic results

• Printing example
  • Possible outputs?
    • ABC123, AB1C23, AB12C3, AB123C, A1BC23, A12BC3, A123BC, 1ABC23, 1A2BC3, ...
  • Impossible outputs?
    • ABC321,...

• Ordering within thread is sequential, but many ways to merge per-thread order into a global order

• What’s being shared between these threads?
  • The output stream
Another example

• x is initially 0
  • x could take many values after execution

• We say that multi-threaded applications have **interleavings**
  • The order of execution of each line in each thread can differ from run to run

• Some **interleavings** may lead to **race conditions**
  • Multiple threads may **race** to execute a certain piece of code

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>x = -1</td>
</tr>
<tr>
<td>x = 0</td>
<td>x = 0</td>
</tr>
<tr>
<td>x = x + 1</td>
<td>x = x - 1</td>
</tr>
</tbody>
</table>
Agenda

- Processes
- Threads
- **Synchronization**
  - Atomic operations
- Sockets
Atomic operations

• Before we can reason at all about cooperating threads, we must know what operations are **atomic**
  - Indivisible, i.e., happens in its entirety or not at all
  - No events from other threads can occur in between

• Print example:
  - What if each print statement were atomic?
  - What if printing a single character were not atomic?

• Most computers
  - Memory load and store are atomic on a single core
  - Other instructions are not atomic (double-prec. Floating pt.)
  - Modern compilers, multicore architectures not seq. consistent
  - Need an atomic operation to build a bigger atomic operation
Atomic operations in Python

• The following operations are atomic in Python:
  • Update, insert, read, etc. of dictionaries, lists
  • Reads/writes of built-in types
  • Read and assignment of instance variables

• Safe to perform these operations simultaneously from different threads

• But combinations of above are not atomic:
  • Safe to execute $x = 1$ and $x = 0$ in different threads
  • Not safe to execute $x = x + 1$ in different threads!
Locks for mutual exclusion

• How can we make $x = x + 1$ safe?

```python
from threading import Thread, Lock

def worker (myLock, i):
    global x
    myLock.acquire()
    x = x + 1
    myLock.release()

if __name__ == "main":
    myLock = Lock()
    for i in range(3):
        t = Thread(target=worker, args=(myLock, i))
        t.start()
```
Agenda

• Processes
• Threads
• Synchronization
  • Atomic operations
• Sockets
The OS network abstraction

- Applications
  - Process
  - File system
  - Virtual mem
  - Sockets

- Operating System
  - CPU
  - Disk
  - RAM
  - Network
Sockets

• How do we share the network?
  • One (or a few) network interface cards/IP addresses
  • But many applications sending/receiving data

• A network **socket** provides virtual NIC for a process
  • Specify sender or receiver by both IP address and port #
  • TCP establishes connection between two sockets
  • UDP sends packets from one socket to another

• **Socket API** is provided by the operating system, and lets application programs use network sockets
Sockets

• A process sends messages to its **socket**

• A process receives messages from its **socket**
Sockets at the command line

Server

# macOS
$ nc -l localhost 8000
hello world

# Linux
$ nc -l -p 8000

Client

# macOS
$ echo "hello world" | nc localhost 8000
Socket example

At the client

• Translate localhost into IP address 127.0.0.1
• Decide to use the TCP protocol
• Create a socket
• connect to 127.0.0.1 at port 8000
• send the data
• Close the connection

$ echo "hello world" | nc localhost 8000
Socket example

At the receiver (server)

• Note: receiver was started first
• Create socket
• Decide to use TCP
• bind the socket to port 8000
• listen on the socket
• accept the connection request
• recv the packets (until connection closed)
• Close the socket

$ nc -l localhost 8000
hello world
import socket
if __name__ == "__main__":
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('', 8000))
    # ...

    Create socket
IPv4 (AF_INET6 for IPv6)
TCP (SOCK_DGRAM for UDP)

    Avoid "[ERROR] Address already in use"

    bind the socket
to port 8000
Socket server in Python

```python
#...
sock.listen(5)

clientsocket, address = sock.accept()
print("Connection from", address)

message_chunks = []
while True:
    data = clientsocket.recv(4096)
    if not data:
        break
    message_chunks.append(data)

# ...
```

**listen** on the socket, queue up as many as 5 connect requests before refusing outside connections

**accept** a connection request

**recv**() returns up to 4096 bytes. Loop to receive all data. Assumes sender cleanly closes connection.
Socket server in Python

```python
#...
clientsocket.close()

message_bytes = b''.join(message_chunks)
message_str = message_bytes.decode("utf-8")
print(message_str)
```

Close the socket

Join together chunks of bytes, then interpret as a UTF-8 string
Socket server in Python

Server

$ python3 test_server.py
Server: connection from ('127.0.0.1', 60980)
hello world

Client

$ echo "hello world" | nc localhost 8000
Socket server in Python

**Server**

```bash
$ python3 test_server.py
Server: connection from ('127.0.0.1', 60980)
hello world
```

**Client**

- When a TCP server accepts a new client, it returns a new socket to communicate with the client
- Allows the server to communicate with multiple clients
import socket

if __name__ == "__main__":
    # create an INET, STREAMing socket
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # connect to the server
    s.connect(("localhost", 8000))

    # send a message
    message = "hello world!"
    sock.sendall(message.encode('utf-8'))
    s.close()
Socket client in Python

**Server**

```
$ nc -l localhost 8000
hello world
```

**Client**

```
$ python3 test_client.py
```
Socket client/server in Python

Server

$ python3 test_server.py
Server: connection from ('127.0.0.1', 60980)
hello world

Client

$ python3 test_client.py
Sockets and project 4

• Project 4 uses processes, threads and sockets to implement a Map Reduce server

• We need to do multiple things in parallel
  • Example: a master and N workers
    • N+1 processes
  • Example: a worker's task (a map function) and a worker's heartbeat (“I’m alive” message)
    • 2 threads

• Communicate between several machines
  • Master communicates with workers using sockets