CSCI-3753: Operating Systems
Fall 2019

Abigail Fernandes

Department of Computer Science

University of Colorado Boulder

Internships and Full Time

Uber

Uber Meet & Greet

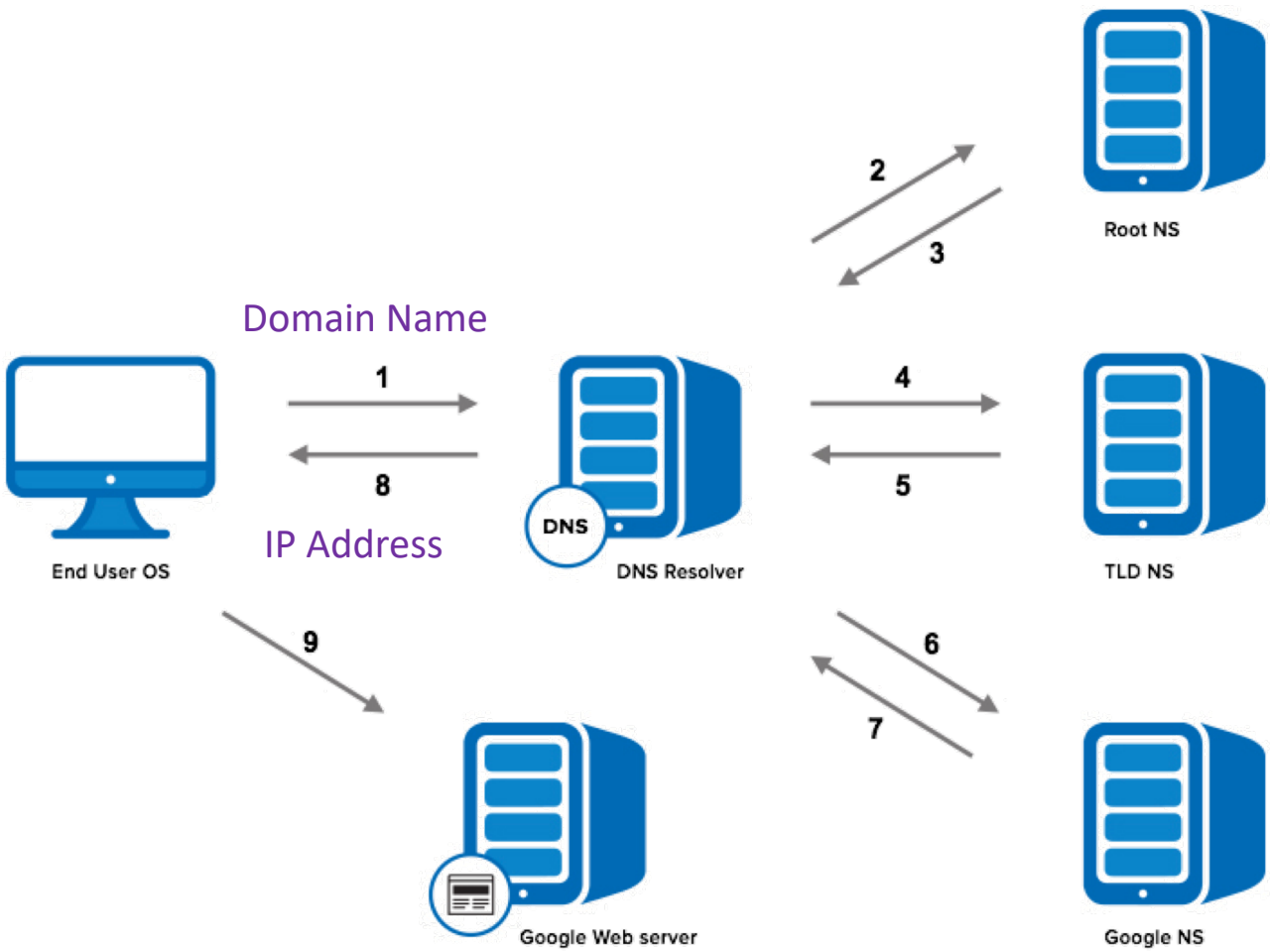
Tuesday, October 29 at 9:00am to 12:00pm

Engineering Center, ECAD Lobby
1111 Engineering Drive, Boulder, CO 80309

LeetCode

Week 7
> Programming Assignment 3

Overview


Root NS
2
3
Domain Name
1
8
IP Address
End User OS
DNS
DNS Resolver
4
5
TLD NS
9
6
7
Google Web server
Google NS

Overview

GOAL: Use **PThreads** to code a DNS Name Resolution Engine

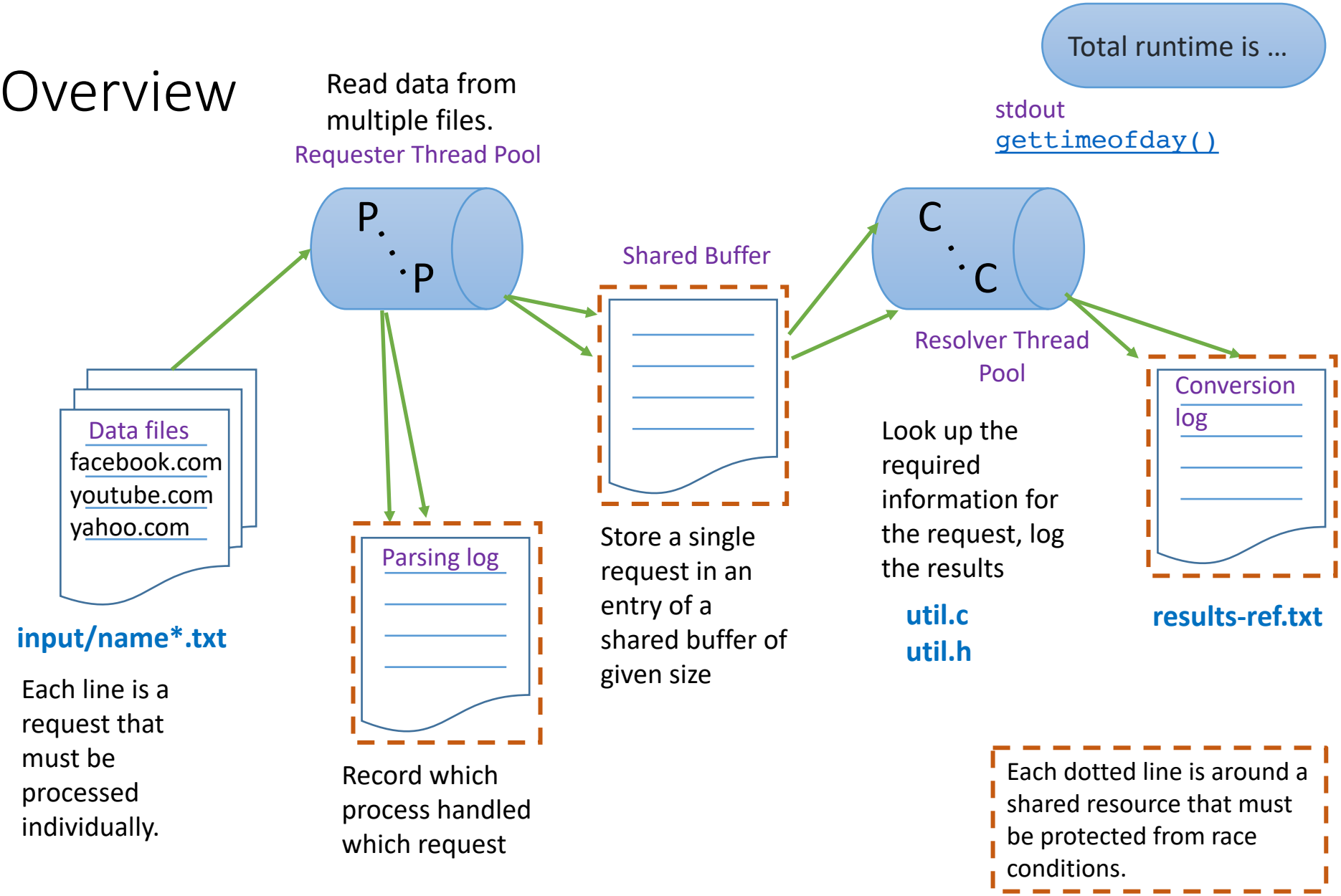
REQUIREMENTS

- Process multiple files
- Requesters
  - Get the next line from a file
  - Parse the line to extract the domain name
  - Place the domain name in a shared buffer
  - Record the processing in a file
- Resolvers
  - Get the next request (domain name) from the shared buffer
  - Do a look up of the IP address based on the domain name
  - Log the information into the output file

Overview

```
For every file
    For every line in the file
        Parse line to get domain names
        For every domain name
            Find the corresponding IP address
            Write information to the output file
```

Overview


Total runtime is …
stdout
gettimeofday()
Read data from multiple files.
Requester Thread Pool
P
P
Shared Buffer
C
C
Resolver Thread Pool
Data files
facebook.com
youtube.com
yahoo.com
input/name*.txt
Each line is a request that must be processed individually.
Parsing log
Record which process handled which request
Store a single request in an entry of a shared buffer of given size
Look up the required information for the request, log the results
util.c
util.h
Conversion log
results-ref.txt
Each dotted line is around a shared resource that must be protected from race conditions.

Program Inputs

```
./multi-lookup 5 10 requester_log.txt results.txt name1.txt name2.txt
```

- **Number** of requester threads to place into the thread pool
- **Number** of resolver threads to place into the thread pool
- **Parsing Logs:** The **file** into which all the parser status information is written
  - Per line format: Thread <Thread ID> serviced ## files
  - `pthread_self()`
- **Conversion Logs:** The file into which all the converter status information is written
  - Per line format: www.google.com, 74.125.224.81
- **Data files:** List of filenames that are to be processed. Each file contains a list of domain names, one per line, that need to be resolved.

Program Limits

- MAX_INPUT_FILES = 10
- MAX_RESOLVER_THREADS = 10
- MAX_REQUESTOR_THREADS = 5
- MAX_DOMAIN_NAME_LENGTH = 1025 characters
- MAX_IP_LENGTH = INET6_ADDRSTRLEN or 46

Error Handling

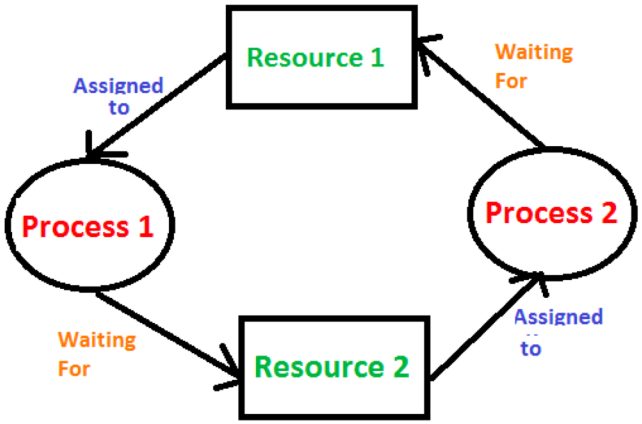
"I didn't run into any bugs in testing, so there are no bugs…right?"

- If the domain name cannot be resolved:
  - Output a blank string in the format ","
  - Print a message to stderr alerting the user
- If output file path is bad:
  - Print an appropriate message to stderr
  - Exit the program (gracefully!)
- If input file path is bad:
  - Print an appropriate error to stderr
  - Move on to the next file

Program Focus

- Synchronization (of access to shared resources)
- Avoid Deadlocks / Busy wait

- Suggested Solutions
  - Mutex
  - Semaphores
  - Conditional Variables

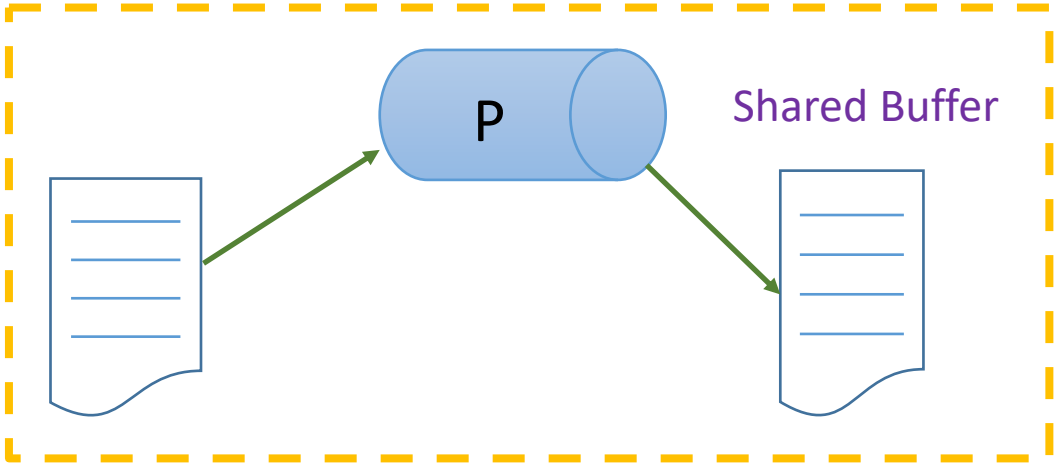


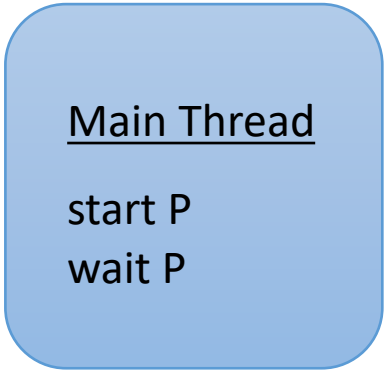
"Great things are done by a series of small things that are brought together."

-Vincent Van Gogh

Implementation – Step 1

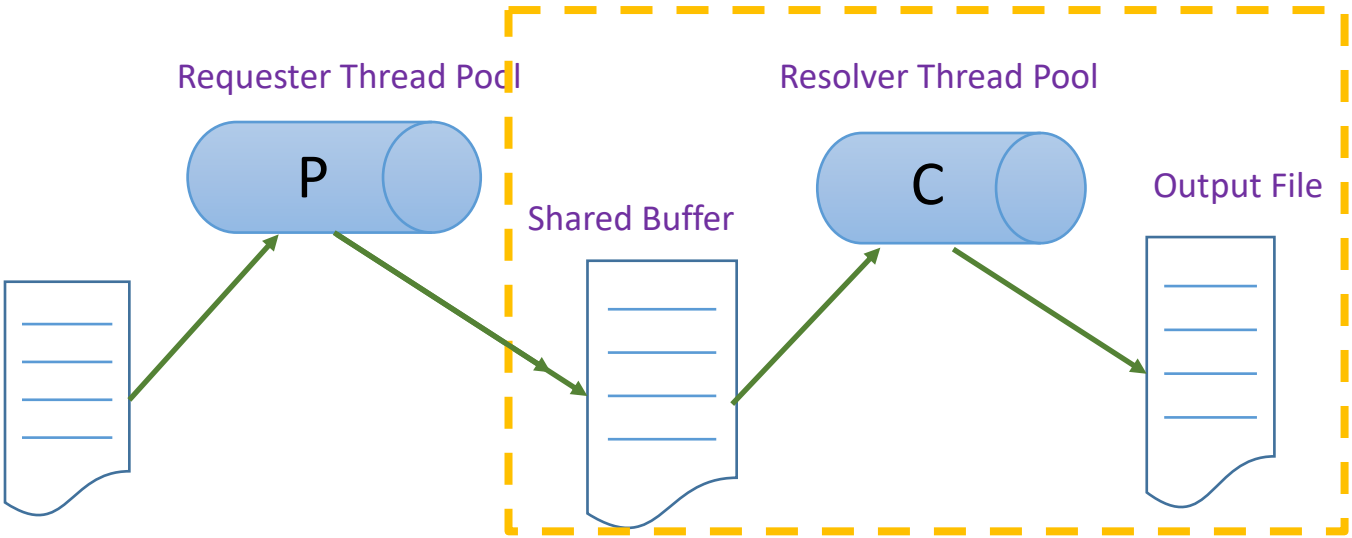
- **TO DO**
  - Write a simple program that creates a thread for **parsing**. This thread will repeatedly read a line from a given file and add an entry into the shared buffer
- **VALIDATE**
  - Does the buffer have the correct number of entries?





Implementation - Step 2

- **TO DO**
  - Now you will use your result from Step 1. Once you have completed that step, start a **conversion** thread to take items out of the shared buffer. Then write the results to an output file.
- **VALIDATE**
  - Does the output file contain the entries stored in the buffer?



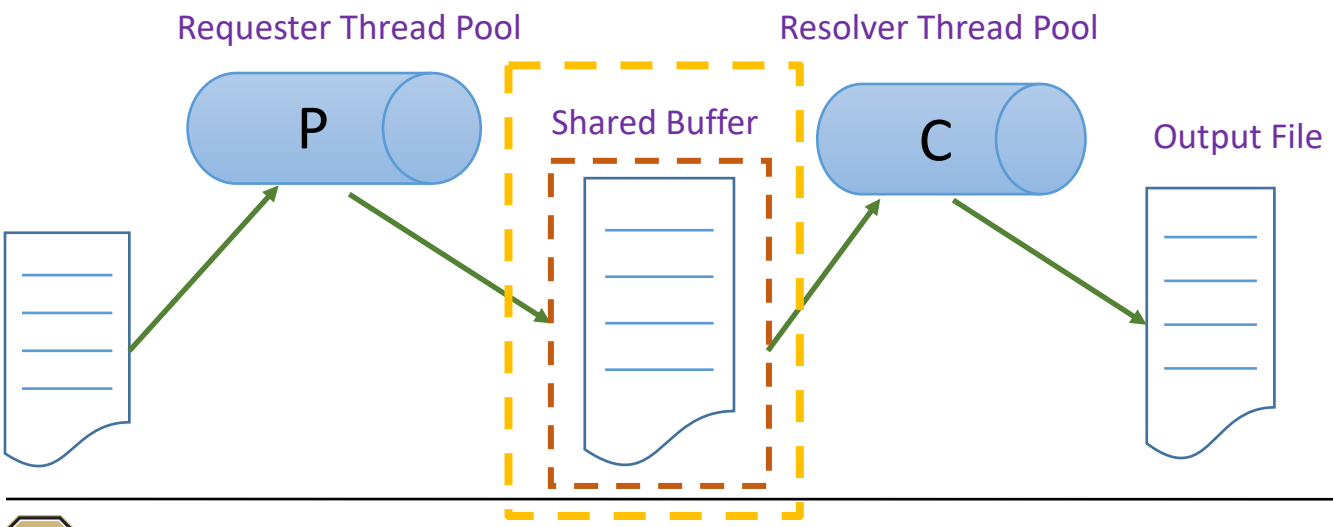
Main Thread

start P
wait P

start C
wait C

Implementation - Step 3

- **TO DO**
  - Once you're sure that your application can read from and write to the the buffer correctly (albeit serially), now try to make it concurrent.
  - Multiple processes can now access and modify the same data. This can result in race conditions.
  - You will need to protect the critical sections of each thread with a mutex.



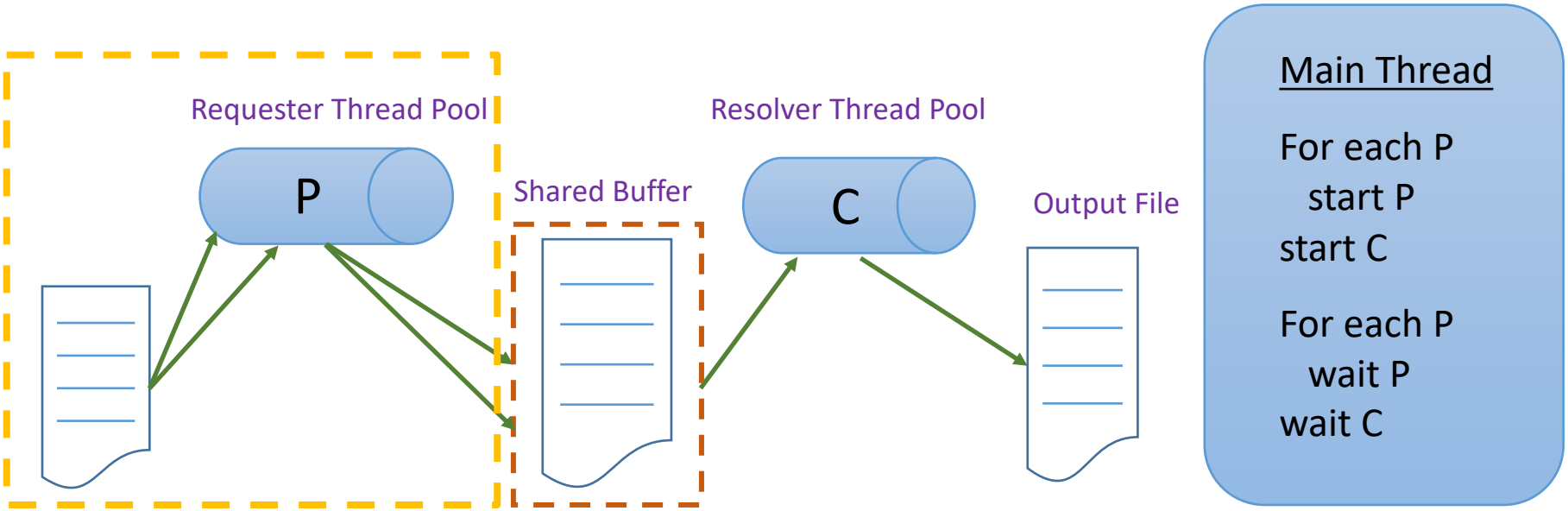
Main Thread

start P
start C

wait P
wait C – exits when no data and no active P

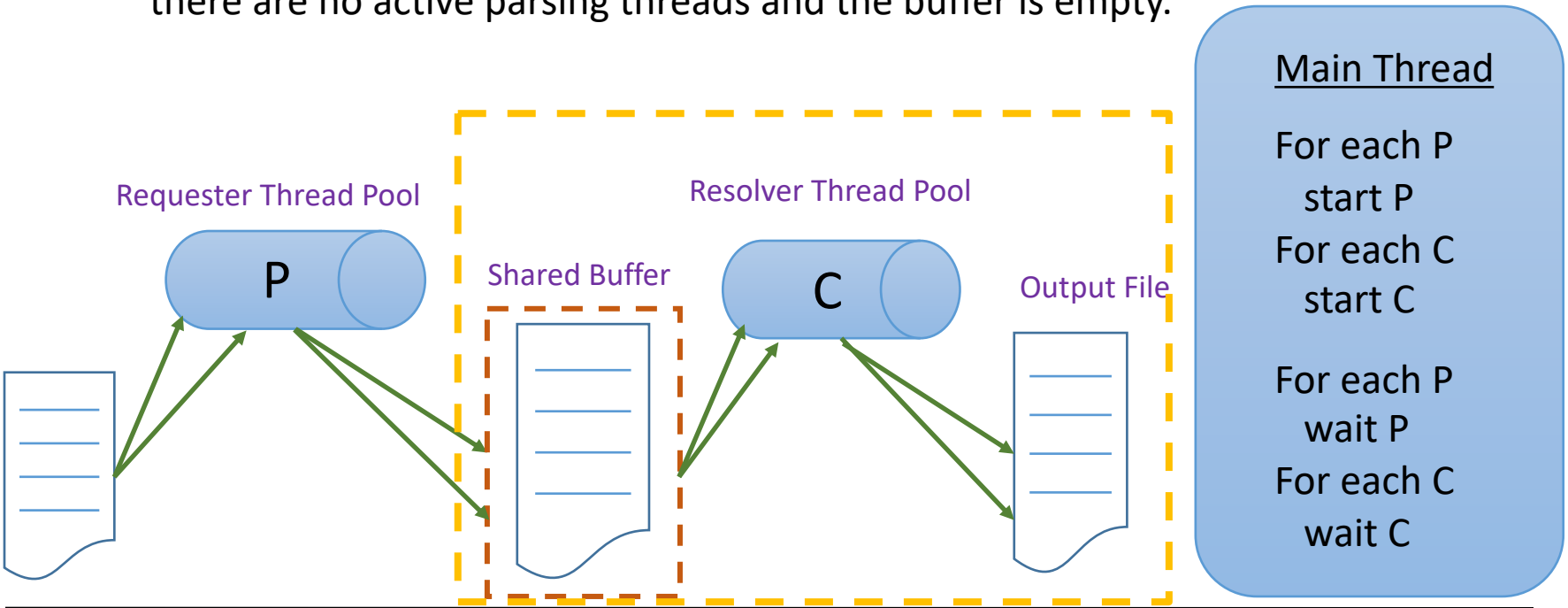
Implementation - Step 4

- **TO DO**
  - Create multiple parsing threads to read from multiple different files. Each parsor can read single lines from a different file. The parsing thread will terminate only when all lines from the file have been processed



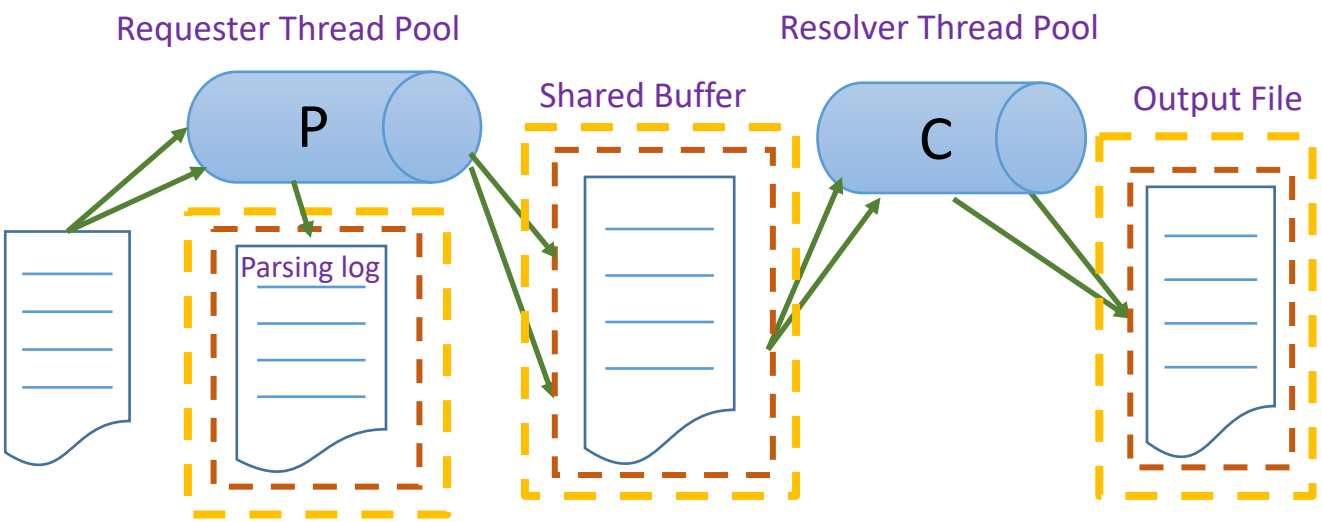
Implementation - Step 5

- **TO DO**
  - Create multiple converter threads to read from multiple parsing threads via the single shared buffer.
  - The converter will wait for data (spin wait is acceptable) but will terminate if there are no active parsing threads and the buffer is empty.



Implementation - Step 6

- **TO DO**
  - Move back to the parsing threads. Each thread must record the data it has processed.
  - Files are a shared resource and therefore must be protected from multiple processes that are accessing it.



Main Thread

For each P
  start P
For each C
  start C

For each P
  wait P
For each C
  wait C
Check P log file

Check Memory Leaks (in your code)

- To verify that your code does not result in any memory leaks, use `valgrind()` to test your code.

- To install valgrind(), run the following command:

```
sudo apt-get install valgrind
```

- To use valgrind() to monitor your program, use this command:

```
valgrind ./pa3main …… text1.text textN.txt
```

Week 8 Checklist

- ❑ PA3 Guidelines
- ❑ PA3 Lab Session in recitation next week
- ❑ **Start PA3** (I'm really looking out for you here!!)

Half way there??

PA 1

PA 2

PA 3

SOON PA 4

Mid Term FCQs

Your feedback is important!!!

Please and Thank you!