Computer Operating Systems Homework 3

Hakan Duran 150200091

May 2024

Your report should convey the problem and the implementation details of your code.

1 Introduction & Problem

In this homework, a single-threaded web server is given and we were asked to make this server multi-threaded. By using thread APIs, i developed skills on multi-threading and observed how threads add speed to the process, especially for web servers.

2 Implementation details

2.1 server.c

In my code, i didn't add the code for extra part, which is scheduling policies (FIFO etc).

As an introduction, i need to introduce global variables and initialization for threads and semaphores:

```c
#include <pthread.h>
#include <semaphore.h>
#include "blg312e.h"
#include "request.h"

sem_t empty; // Semaphore to track empty slots in the buffer
sem_t full;  // Semaphore to track filled slots in the buffer
sem_t mutex; // Mutex for accessing shared buffer
int *buffer; // Shared buffer to hold connection descriptors
int num_buffers = 0; // Maximum number of buffers
int in = 0; // Index for adding items to the buffer
int out = 0; // Index for removing items from the buffer

int main(int argc, char *argv[])
{
  int listenfd, connfd, port, clientlen;
  struct sockaddr_in clientaddr;
  int num_threads;

  getargs(&port, &num_threads, &num_buffers, argc, argv);

  buffer = malloc(num_buffers * sizeof(int));
  if (buffer == NULL) {
    fprintf(stderr, "Failed to allocate memory for buffer\n");
    exit(1);
  }

  // Semaphores initialized here
  sem_init(&empty, 0, num_buffers); // It will make thread sleep when buffer is full
  sem_init(&full, 0, 0); // It will make thread sleep when buffer is empty
  sem_init(&mutex, 0, 1); // Locks for producer-consumer relationship

```

```
33   // Threads created here
34   pthread_t tid;
35   for (int i = 0; i < num_threads; i++) {
36       pthread_create(&tid, NULL, worker, NULL);
37   }
38   ...
39 }
```

There are 3 semaphores here called as empty, full and mutex. Empty and full semaphores are for producer-consumer relationship between worker and master thread. Mutex is used for critical sections.

Buffer pointer points to buffer array, which created in line 22. This buffer will consist of file descriptors which will be reached by worker threads.

Semaphore initialization has been made carefully as it is seen in line 29. Semaphore "empty" is initialized by the value of num_buffers. It is decremented when new request comes in. If buffer became full, it will make master thread sleep. Semaphore "full" is initialized by the value of 0. It will make worker threads sleep when buffer is empty. Since semaphore "mutex" is for lock, it is initialized by 1.

In line 35, it can be seen that threads are being created by using the function of pthread_create(). The number of threads is determined by the user.

```
1  void *worker(void *arg) {
2      int connfd;
3      while (1) {
4          sem_wait(&full); // Wait if buffer is empty
5          sem_wait(&mutex);
6          connfd = buffer[out]; // Get connection descriptor from buffer
7          out = (out + 1) % num_buffers; // Update index
8          sem_post(&mutex);
9          sem_post(&empty); // Signal that there's an empty slot in the buffer
10
11         // Handle the request
12         requestHandle(connfd);
13         Close(connfd);
14     }
15     return NULL;
16 }
17
18 int main(int argc, char *argv[])
19 {
20 ...
21 listenfd = Open_listenfd(port);
22   while (1) {
23       clientlen = sizeof(clientaddr);
24       connfd = Accept(listenfd, (SA *)&clientaddr, (socklen_t *)&clientlen);
25
26       sem_wait(&empty); // Wait if buffer is full
27       sem_wait(&mutex);
28       buffer[in] = connfd; // Put connection descriptor in buffer
29       in = (in + 1) % num_buffers; // Update index
30       sem_post(&mutex);
31       sem_post(&full); // Signal that there's a filled slot in the buffer
32   }
33 }
```

In the above code, worker and master threads can be seen. Connection is accepted and connection file descriptor (connfd) put in the buffer while request is handled by worker threads.

In the master thread, after getting connection fd (connfd), the first function we see is sem_wait(&empty), which means master will wait for buffer to empty. It only happens in line 9, sem_post(&empty), which signals there is empty slot in the buffer.

After first semaphore, we came to line 27, sem_wait(&mutex). Mutual exclusion prevents threads from interfering in each other's jobs. File descriptor is added to buffer and counter variable "in" updated.

Semaphore function sem_post(&mutex) shows that it is end of critical section. At line 31, sem_post(&full) function increases the semaphore "full" by one, signals to the worker threads which waits at line 4, sem_wait(&full).

Worker thread is not much different than master thread. It waits for a signal from master if buffer is empty. In critical section, connfd has acquired and counter variable "out" updated. In the last, request handled.

2.2 client.c

In this part, i am extracting requested files from a file. This file is consists of requested files and seperated by newline. The name is given from console.

Usage:

./client <host> <port> <file>

Example usage:

./client localhost 2000 files.txt

Inside of files.txt:

```
/output.cgi
/favicon.ico
/output.cgi
/
/output.cgi
```

Here is the part of main function:

```
char line[MAX_FILENAME_LENGTH];

// In here, each line of file is assigned to "line" variable
// Connection established with Open_clientfd
// Parameters are determined and thread created

while (fgets(line, sizeof(line), file) != NULL) {

line[strcspn(line, "\n")] = '\0';

clientfd = Open_clientfd(host, port);

ThreadParams *params = malloc(sizeof(ThreadParams));
params->arg1 = clientfd;
params->arg2 = strdup(line);

pthread_t tid;
pthread_create(&tid, NULL, clientSend, (void *)params);

}
```

Each line extracted from file and clientfd are parameters for created thread. Below is the clientSend function:

```
typedef struct { // Struct for clientSend function's parameters
    int arg1;
    char* arg2;
} ThreadParams;

void clientSend(void *args)
{
  // Parameters are assigned to values here
  ThreadParams *params = (ThreadParams *)args;
  int fd = params->arg1;
  char* filename = params->arg2;

  char buf1[MAXLINE];
  char hostname[MAXLINE];

  .
  .
  .

  while (n > 0) {
```

```
21      printf("%s", buf);
22      n = Rio_readlineb(&rio, buf, MAXBUF);
23    }
24
25    // Allocated params value is freed here
26    free(params);
27
28    return NULL;
29  }
```

In first 4 lines, a struct created for sending parameters to threads. Line 8-11 is for assigning parameter values to the inner variables. In line 26, params is freed.

3 Example

In order to test my code, i've requested files in files.txt. I also changed spinfor variable from 5 to 10 in order to have much more scalable measurement in output.c. Here are my server and clients:

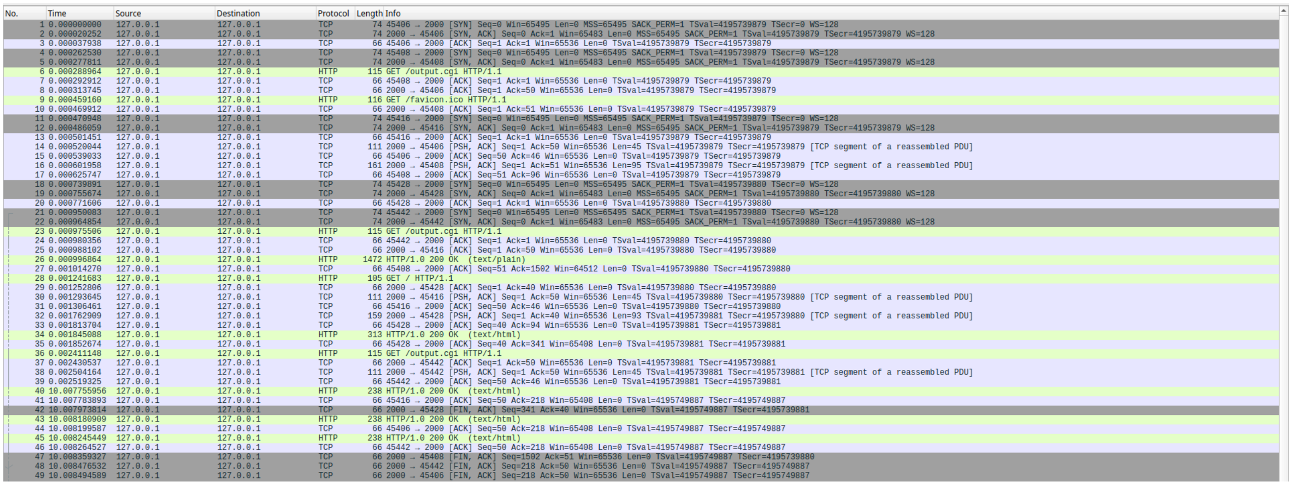
1. ./server 2000 5 10
2. ./client localhost 2000 files.txt

Inside of files.txt:

```
/output.cgi
/favicon.ico
/output.cgi
/
/output.cgi
```

Before implementing of multi-threading, it would take roughly 30 seconds to handle all requests. After implementing multi-threading, it started to take roughly 10 seconds. It is because different threads have a chance to deal with different requests anymore.

Here is a pcap file from wireshark to show how requesting 5 file which consists of 3 output.cgi is only takes 10 seconds:



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	45406 → 2000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=4195739879 TSecr=0 WS=128
2	0.000020252	127.0.0.1	127.0.0.1	TCP	74	2000 → 45406 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=4195739879 TSecr=4195739879 WS=128
3	0.000037938	127.0.0.1	127.0.0.1	TCP	66	45406 → 2000 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=4195739879 TSecr=4195739879
4	0.000262530	127.0.0.1	127.0.0.1	TCP	74	45408 → 2000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=4195739879 TSecr=0 WS=128
5	0.000277811	127.0.0.1	127.0.0.1	TCP	74	2000 → 45408 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=4195739879 TSecr=4195739879 WS=128
6	0.000288964	127.0.0.1	127.0.0.1	HTTP	115	GET /output.cgi HTTP/1.1
7	0.000292912	127.0.0.1	127.0.0.1	TCP	66	45408 → 2000 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=4195739879 TSecr=4195739879
8	0.000313745	127.0.0.1	127.0.0.1	TCP	66	2000 → 45406 [ACK] Seq=1 Ack=50 Win=65536 Len=0 TSval=4195739879 TSecr=4195739879
9	0.000459160	127.0.0.1	127.0.0.1	HTTP	116	GET /favicon.ico HTTP/1.1
10	0.000469912	127.0.0.1	127.0.0.1	TCP	66	2000 → 45408 [ACK] Seq=1 Ack=51 Win=65536 Len=0 TSval=4195739879 TSecr=4195739879
11	0.000470948	127.0.0.1	127.0.0.1	TCP	74	45416 → 2000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=4195739879 TSecr=0 WS=128
12	0.000486059	127.0.0.1	127.0.0.1	TCP	74	2000 → 45416 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=4195739879 TSecr=4195739879 WS=128
13	0.000501451	127.0.0.1	127.0.0.1	TCP	66	45416 → 2000 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=4195739879 TSecr=4195739879
14	0.000520044	127.0.0.1	127.0.0.1	TCP	111	2000 → 45406 [PSH, ACK] Seq=1 Ack=50 Win=65536 Len=45 TSval=4195739879 TSecr=4195739879 [TCP segment of a reassembled PDU]
15	0.000539033	127.0.0.1	127.0.0.1	TCP	66	45406 → 2000 [ACK] Seq=50 Ack=46 Win=65536 Len=0 TSval=4195739879 TSecr=4195739879
16	0.000601958	127.0.0.1	127.0.0.1	TCP	161	2000 → 45408 [PSH, ACK] Seq=1 Ack=51 Win=65536 Len=95 TSval=4195739879 TSecr=4195739879 [TCP segment of a reassembled PDU]
17	0.000625747	127.0.0.1	127.0.0.1	TCP	66	45408 → 2000 [ACK] Seq=51 Ack=96 Win=65536 Len=0 TSval=4195739879 TSecr=4195739879
18	0.000739891	127.0.0.1	127.0.0.1	TCP	74	45428 → 2000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=4195739880 TSecr=0 WS=128
19	0.000755674	127.0.0.1	127.0.0.1	TCP	74	2000 → 45428 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=4195739880 TSecr=4195739880 WS=128
20	0.000771606	127.0.0.1	127.0.0.1	TCP	66	45428 → 2000 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=4195739880 TSecr=4195739880
21	0.000950083	127.0.0.1	127.0.0.1	TCP	74	45442 → 2000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=4195739880 TSecr=0 WS=128
22	0.000964854	127.0.0.1	127.0.0.1	TCP	74	2000 → 45442 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=4195739880 TSecr=4195739880 WS=128
23	0.000975506	127.0.0.1	127.0.0.1	HTTP	115	GET /output.cgi HTTP/1.1
24	0.000980356	127.0.0.1	127.0.0.1	TCP	66	45442 → 2000 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=4195739880 TSecr=4195739880
25	0.000988102	127.0.0.1	127.0.0.1	TCP	66	2000 → 45416 [ACK] Seq=1 Ack=50 Win=65536 Len=0 TSval=4195739880 TSecr=4195739880
26	0.000996864	127.0.0.1	127.0.0.1	HTTP	1472	HTTP/1.0 200 OK (text/plain)
27	0.001014270	127.0.0.1	127.0.0.1	TCP	66	45408 → 2000 [ACK] Seq=51 Ack=1502 Win=64512 Len=0 TSval=4195739880 TSecr=4195739880
28	0.001241683	127.0.0.1	127.0.0.1	HTTP	105	GET / HTTP/1.1
29	0.001252806	127.0.0.1	127.0.0.1	TCP	66	2000 → 45428 [ACK] Seq=1 Ack=40 Win=65536 Len=0 TSval=4195739880 TSecr=4195739880
30	0.001293645	127.0.0.1	127.0.0.1	TCP	111	2000 → 45416 [PSH, ACK] Seq=1 Ack=50 Win=65536 Len=45 TSval=4195739880 TSecr=4195739880 [TCP segment of a reassembled PDU]
31	0.001306461	127.0.0.1	127.0.0.1	TCP	66	45416 → 2000 [ACK] Seq=50 Ack=46 Win=65536 Len=0 TSval=4195739880 TSecr=4195739880
32	0.001762909	127.0.0.1	127.0.0.1	TCP	159	2000 → 45428 [PSH, ACK] Seq=1 Ack=40 Win=65536 Len=93 TSval=4195739881 TSecr=4195739880 [TCP segment of a reassembled PDU]
33	0.001813704	127.0.0.1	127.0.0.1	TCP	66	45428 → 2000 [ACK] Seq=40 Ack=94 Win=65536 Len=0 TSval=4195739881 TSecr=4195739881
34	0.001845088	127.0.0.1	127.0.0.1	HTTP	313	HTTP/1.0 200 OK (text/html)
35	0.001852674	127.0.0.1	127.0.0.1	TCP	66	45428 → 2000 [ACK] Seq=40 Ack=341 Win=65408 Len=0 TSval=4195739881 TSecr=4195739881
36	0.002411148	127.0.0.1	127.0.0.1	HTTP	115	GET /output.cgi HTTP/1.1
37	0.002430537	127.0.0.1	127.0.0.1	TCP	66	2000 → 45442 [ACK] Seq=1 Ack=50 Win=65536 Len=0 TSval=4195739881 TSecr=4195739881
38	0.002504164	127.0.0.1	127.0.0.1	TCP	111	2000 → 45442 [PSH, ACK] Seq=1 Ack=50 Win=65536 Len=45 TSval=4195739881 TSecr=4195739881 [TCP segment of a reassembled PDU]
39	0.002519325	127.0.0.1	127.0.0.1	TCP	66	45442 → 2000 [ACK] Seq=50 Ack=46 Win=65536 Len=0 TSval=4195739881 TSecr=4195739881
40	10.007755956	127.0.0.1	127.0.0.1	HTTP	238	HTTP/1.0 200 OK (text/html)
41	10.007783893	127.0.0.1	127.0.0.1	TCP	66	45416 → 2000 [ACK] Seq=50 Ack=218 Win=65408 Len=0 TSval=4195749887 TSecr=4195749887
42	10.007973814	127.0.0.1	127.0.0.1	TCP	66	2000 → 45428 [FIN, ACK] Seq=341 Ack=40 Win=65536 Len=0 TSval=4195749887 TSecr=4195739881
43	10.008180909	127.0.0.1	127.0.0.1	HTTP	238	HTTP/1.0 200 OK (text/html)
44	10.008199587	127.0.0.1	127.0.0.1	TCP	66	45406 → 2000 [ACK] Seq=50 Ack=218 Win=65408 Len=0 TSval=4195749887 TSecr=4195749887
45	10.008245449	127.0.0.1	127.0.0.1	HTTP	238	HTTP/1.0 200 OK (text/html)
46	10.008264527	127.0.0.1	127.0.0.1	TCP	66	45442 → 2000 [ACK] Seq=50 Ack=218 Win=65408 Len=0 TSval=4195749887 TSecr=4195749887
47	10.008359327	127.0.0.1	127.0.0.1	TCP	66	2000 → 45408 [FIN, ACK] Seq=1502 Ack=51 Win=65536 Len=0 TSval=4195749887 TSecr=4195739880
48	10.008476532	127.0.0.1	127.0.0.1	TCP	66	2000 → 45442 [FIN, ACK] Seq=218 Ack=50 Win=65536 Len=0 TSval=4195749887 TSecr=4195749887
49	10.008494589	127.0.0.1	127.0.0.1	TCP	66	2000 → 45406 [FIN, ACK] Seq=218 Ack=50 Win=65536 Len=0 TSval=4195749887 TSecr=4195749887

Figure 1: PCAP file of whole request-response event

After 40. packet, it can be seen that requested output.cgi files has arrived. The arrival of packets containing the FIN flag after the 47th packet proves that it only takes 10 seconds to request 5 files. Since they are threads, small files are coming primarily even though they are not the one which requested firstly.