

VSCODE\_PRINT\_SCRIPT\_TAGS

# Selected files

## 2 printable files

Assignments\Assignment3\stnc.c  
Assignments\Assignment3\makefile

### Assignments\Assignment3\stnc.c

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include <stdint.h>
4  #include<string.h>
5  #include <sys/un.h>
6  #include<sys/socket.h>
7  #include <sys/time.h>
8  #include <sys/types.h>
9  #include <sys/stat.h>
10 #include <sys/mman.h>
11 #include<netinet/tcp.h>
12 #include<netinet/in.h>
13 #include <arpa/inet.h>
14 #include <errno.h>
15 #include <poll.h>
16 #include <netdb.h>
17 #include <unistd.h>
18 #include <fcntl.h>
19 #include <time.h>
20 #include <unistd.h>
21
22 #define B_SIZE 6400
23 #define B_SIZE_UDP 64000
24 # define B_SIZE_UDP_IPV6 3000
25 #define FIFO_NAME "OS_EX3_pipe"
26 #define SOCK_PATH "/tmp/stnc_sock_path.sock"
27
28 int ipv6_to_ipv4(char *ipv6_str, char *ipv4_str) {
29     struct in6_addr ipv6_addr;
30     struct sockaddr_in ipv4_addr;
31     int ret;
32
33     // Convert the IPv6 address string to a binary representation
34     ret = inet_pton(AF_INET6, ipv6_str, &ipv6_addr);
35     if (ret != 1) {
36         return -1;
37     }
38
39     // Convert the IPv6 address to an IPv4-mapped IPv6 address
40     memset(&ipv4_addr, 0, sizeof(ipv4_addr));
41     ipv4_addr.sin_family = AF_INET;
42     ipv4_addr.sin_port = 0;
43     ipv4_addr.sin_addr.s_addr =
44         htonl(0xFFFFF000) | ((ipv6_addr.s6_addr[12] << 8) & 0xFF00) |
45         (ipv6_addr.s6_addr[13] & 0xFF);
```

```
46 // Convert the IPv4 address to a string
47 inet_ntop(AF_INET, &ipv4_addr.sin_addr, ipv4_str, INET_ADDRSTRLEN);
48
49 return 0;
50 }
51
52 void port_for_info(char *port, char *port_out) {
53     int new_port = atoi(port);
54     new_port++;
55     if (new_port == 65536) {
56         new_port -= 2;
57     }
58     sprintf(port_out, "%d", new_port);
59 }
60
61
62 long checksum(char *str, int limit) {
63     long sum = 0;
64     for (int i = 0; str[i] != '\0' && i < limit; i++) {
65         sum += (uint8_t) str[i];
66     }
67     return sum;
68 }
69
70 long checksum_file(FILE *file, long *bytes_counter) {
71     uint8_t byte;
72     long sum = 0;
73     long count = 0;
74     while (fread(&byte, sizeof(byte), 1, file) == 1) {
75         sum += byte;
76         count++;
77     }
78     *bytes_counter = count;
79     return sum;
80 }
81
82
83 int portHandler(int port) {
84     if ((port <= 1024) || (port >= 65535)) {
85         printf("PORT should be numerical between 1024 and 65535 included\n");
86         exit(EXIT_FAILURE);
87     }
88     return 0;
89 }
90
91 int IPtype(char *ip) {
92     for (int i = 0; ip[i] != '\0'; i++) {
93         if (ip[i] == ':') {
94             return 6;
95         }
96     }
97     return 4;
98 }
99
100
101 int IPv4Handler(char *ip) {
102     char *token = strtok(ip, ".");
103     int dot_count = 0;
104     while (token != NULL) {
105         int num = atoi(token);
```

```
106     if (num < 0 || num > 255) {
107         printf("Invalid IP address\n");
108         return 0;
109     }
110     token = strtok(NULL, ".");
111     dot_count++;
112 }
113
114 if (dot_count != 4) {
115     printf("Invalid IP address\n");
116     return 0;
117 }
118
119 return 1;
120 }
121
122 int IPv6Handler(char *ip) {
123     struct sockaddr_in6 sa6;
124     int result = inet_pton(AF_INET6, ip, &(sa6.sin6_addr));
125     return result == 1;
126 }
127
128
129 int tcp_client_conn(char *ip, char *port) {
130     int PORT = atoi(port);
131     char IP[100];
132     strcpy(IP, ip);
133     portHandler(PORT);
134
135     int ip_type = IPtype(ip);
136
137     if (ip_type == 4) {
138
139         if (IPv4Handler(IP)) {
140             //initializing a TCP socket.
141             int sock = socket(AF_INET, SOCK_STREAM, 0);
142             if (sock == -1) {
143                 printf("Could not create socket.\n");
144                 exit(EXIT_FAILURE);
145             }
146
147             struct sockaddr_in receiver_address;
148             //setting to zero the struct senderAddress
149             memset(&receiver_address, 0, sizeof(receiver_address));
150             receiver_address.sin_family = AF_INET;
151             receiver_address.sin_port = htons(PORT);
152             int checkP = inet_pton(AF_INET, (const char *) IP,
153 &receiver_address.sin_addr);
154
155             if (checkP < 0) {
156                 printf("inet_pton() FAILED.\n");
157                 exit(EXIT_FAILURE);
158             }
159
160             //connecting to the Receiver on the socket
161             sleep(1);
162             int connectCheck = connect(sock, (struct sockaddr *) &receiver_address,
163 sizeof(receiver_address));
164
165             if (connectCheck == -1) {
```

```

164         printf("connect() FAILED.\n");
165         exit(EXIT_FAILURE);
166     }
167     return sock;
168 }
169 } else if (ip_type == 6) {
170     if (IPv6Handler(ip)) { //need to create IPv6Handler method
171         int sock = socket(AF_INET6, SOCK_STREAM, 0);
172         if (sock == -1) {
173             perror("socket error\n");
174             exit(EXIT_FAILURE);
175         }
176         int on = 1;
177         if (setsockopt(sock, IPPROTO_IPV6, IPV6_V6ONLY, &on, sizeof(on)) == -1) {
178             perror("setsockopt error\n");
179             exit(EXIT_FAILURE);
180         }
181         // struct addrinfo hints, *res; //tell along there is an error in hints
182         // memset(&hints, 0, sizeof(hints)); //error on sizeof
183         // hints.ai_family = AF_INET6; //error on hints
184         // hints.ai_socktype = SOCK_STREAM; //error on hints
185         // if (getaddrinfo(ip, port, &hints, &res) != 0) {
186         //     perror("getaddrinfo error");
187         //     exit(EXIT_FAILURE);
188         // }
189         // if (connect(sock, res->ai_addr, res->ai_addrlen) == -1) {
190         //     perror("connect error");
191         //     exit(EXIT_FAILURE);
192         // }
193         return sock;
194     }
195 }
196 return -1;
197 }
198
199 int tcp_server_conn(char *port) {
200     int PORT = atoi(port);
201     char IP[] = "0.0.0.0";
202     portHandler(PORT);
203
204     //initializing a TCP socket.
205     int sock = socket(AF_INET, SOCK_STREAM, 0);
206     struct sockaddr_in senderAddress;
207     //setting to zero the struct senderAddress
208     memset(&senderAddress, 0, sizeof(senderAddress));
209     senderAddress.sin_family = AF_INET;
210     senderAddress.sin_port = htons(PORT);
211
212     //opening the socket.
213     int Bcheck = bind(sock, (struct sockaddr *) &senderAddress, sizeof(senderAddress));
214     if (Bcheck == -1) {
215         return 1;
216     }
217     //start listening on the socket (one client at the time)
218     int Lcheck = listen(sock, 4);
219     if (Lcheck == -1) {
220         printf("Error in listen().\n");
221         return 1;
222     }
223 }

```

```

224 //accepting the client (the Sender)
225 unsigned int senderAddressLen = sizeof(senderAddress);
226 int senderSock = accept(sock, (struct sockaddr *) &senderAddress, &senderAddressLen);
227 if (senderSock == -1) {
228     printf("accept() failed.\n");
229     close(sock);
230     return 1;
231 }
232 close(sock);
233
234 return senderSock;
235
236 }
237
238
239 void usage() {
240     printf("Usage options:\n");
241     printf("client side usage: ./stnc -c IP PORT\n");
242     printf("server side usage: ./stnc -s PORT\n");
243 }
244
245
246 int client_A(char *port, char *ip) {
247
248     int sock = tcp_client_conn(ip, port);
249     int fd = -1;
250     struct pollfd fds[2];
251     fds[0].fd = 0; // stdin
252     fds[0].events = POLLIN; // tell me when I can read from it
253     fds[1].fd = sock;
254     fds[1].events = POLLIN;
255     while (1) {
256         int err = poll(fds, 2, -1);
257         if (err < 0) {
258             printf("poll failed\n");
259             return 1;
260         }
261         if (fds[0].revents && POLLIN) {
262             // read from keyboard and send to the server
263             char buffer[10000] = {'\0'};
264             if (fgets(buffer, sizeof(buffer), stdin) != NULL) {
265                 // remove newline character from the end of the line
266                 buffer[strcspn(buffer, "\n")] = '\0';
267             }
268             int size = strlen(buffer);
269             if (send(sock, buffer, size, 0) < 0) {
270                 perror("could not send the data\n");
271                 return 1;
272             }
273         }
274         if (fds[1].revents && POLLIN) {
275             // read from sock and print out
276             char buffer2[10000] = {'\0'};
277             if (recv(sock, buffer2, 10000, 0) < 0) {
278                 perror("could not receive data\n");
279                 return 1;
280             }
281             printf("%s\n", buffer2);
282         }
283     }

```

```
284     return 0;
285 }
286
287 int server_A(char *port) {
288     int PORT = atoi(port);
289     char IP[] = "0.0.0.0";
290     portHandler(PORT);
291
292     int sock = socket(AF_INET, SOCK_STREAM, 0);
293     struct sockaddr_in senderAddress;
294     //setting to zero the struct senderAddress
295     memset(&senderAddress, 0, sizeof(senderAddress));
296     senderAddress.sin_family = AF_INET;
297     senderAddress.sin_port = htons(PORT);
298
299     //opening the socket.
300     int Bcheck = bind(sock, (struct sockaddr *) &senderAddress, sizeof(senderAddress));
301     if (Bcheck == -1) {
302         return 1;
303     }
304     //start listening on the socket (one client at the time)
305     int Lcheck = listen(sock, 1);
306     if (Lcheck == -1) {
307         printf("Error in listen().\n");
308         return 1;
309     }
310
311     //accepting the client (the Sender)
312     unsigned int senderAddressLen = sizeof(senderAddress);
313     int senderSock = accept(sock, (struct sockaddr *) &senderAddress, &senderAddressLen);
314     if (senderSock == -1) {
315         printf("accept() failed.\n");
316         close(sock);
317         return 1;
318     }
319     int fd = -1;
320     struct pollfd fds[2];
321     fds[0].fd = 0; // stdin
322     fds[0].events = POLLIN; // tell me when I can read from it
323     fds[1].fd = senderSock;
324     fds[1].events = POLLIN;
325     while (1) {
326         int err = poll(fds, 2, -1);
327         if (err < 0) {
328             perror("poll failed\n");
329             return 1;
330         }
331         if (fds[0].revents & POLLIN) {
332             // read from keyboard and send to the server
333             char buffer[10000] = {'\0'};
334
335             if (fgets(buffer, sizeof(buffer), stdin) != NULL) {
336                 // remove newline character from the end of the line
337                 buffer[strcspn(buffer, "\n")] = '\0';
338             }
339             int size = strlen(buffer);
340             if (send(senderSock, buffer, size, 0) < 0) {
341                 perror("could not send the data\n");
342                 close(senderSock);
343                 return 1;
344             }
345         }
346     }
```

```

344     }
345 }
346 if (fds[1].revents & POLLIN) {
347     // read from sock and print out
348     char buffer2[10000] = {'\0'};
349     if (recv(senderSock, buffer2, 10000, 0) < 0) {
350         perror("could not receive data\n");
351         close(senderSock);
352         return 1;
353     }
354     printf("%s\n", buffer2);
355 }
356 }
357
358 return 0;
359 }
360
361 //sending on the info socket to the server <type> , <param>, checksum, bytes to send.
362 void type_param(int sock, char *type, char *param, long checks, long bytes) {
363     char checks_str[16] = {'\0'};
364     char bytes_str[16] = {'\0'};
365     sprintf(checks_str, "%ld", checks);
366     sprintf(bytes_str, "%ld", bytes);
367     char message[100] = {'\0'};
368     strcat(message, type);
369     strcat(message, ",");
370     strcat(message, param);
371     strcat(message, ",");
372     strcat(message, checks_str);
373     strcat(message, ",");
374     strcat(message, bytes_str);
375
376     send(sock, message, strlen(message), 0);
377 }
378
379 int client_TCP_B(char *ip, char *port, int info_sock, FILE *file) {
380     int data_sock = tcp_client_conn(ip, port);
381
382     char buffer[B_SIZE];
383     size_t bytes_read;
384     char *start = "start";
385     char *end = "end";
386     send(info_sock, start, strlen(start), 0);
387     fseek(file, 0L, SEEK_SET);
388     while (1) {
389         bytes_read = fread(buffer, 1, B_SIZE, file);
390
391         if (bytes_read == 0) {
392             // End of file
393             break;
394         }
395         //buffer[bytes_read] = '\0'; // add null terminator
396         if (send(data_sock, buffer, bytes_read, 0) == -1) {
397             perror("send");
398             exit(EXIT_FAILURE);
399         }
400     }
401
402     send(info_sock, end, strlen(end), 0);
403

```

```

404     close(data_sock);
405     close(info_sock);
406     return 0;
407 }
408
409 int server_TCP_B(char *port, int info_sock, long bytes_target, long checksum_target, int
q) {
410     struct timeval start, end, diff;
411     int data_sock = tcp_server_conn(port);
412
413     struct pollfd fds[2];
414     fds[0].fd = info_sock;
415     fds[0].events = POLLIN; // tell me when I can read from it
416     fds[1].fd = data_sock;
417     fds[1].events = POLLIN;
418     long bytes_recived = 0;
419     long checksum_sum = 0;
420     int done = 0;
421     int started = 0;
422     char buffer[B_SIZE];
423     char buffer_str[B_SIZE + 1];
424     while (1) {
425         memset(buffer, 0, sizeof(buffer));
426         int err = poll(fds, 2, -1);
427         if (err < 0) {
428             perror("poll");
429             return 1;
430         }
431         if (fds[0].revents && POLLIN) {
432             //read from the socket
433             if (!started)
434                 recv(info_sock, buffer, strlen("start") + 1, 0);
435             else
436                 recv(info_sock, buffer, strlen("end") + 1, 0);
437             if (strcmp(buffer, "start") == 0) {
438                 gettimeofday(&start, NULL);
439                 started = 1;
440             } else if (strcmp(buffer, "end") == 0) {
441                 gettimeofday(&end, NULL);
442                 done = 1;
443             }
444         }
445         else if (fds[1].revents && POLLIN) {
446             // recive the data and count byts
447             bytes_recived += recv(data_sock, buffer, sizeof(buffer), 0);
448             strcpy(buffer_str, buffer);
449             buffer_str[B_SIZE] = '\0';
450             checksum_sum += checksum(buffer_str, B_SIZE + 1);
451         }
452
453         if (done == 1) {
454             //set the socket to non-blocking code
455             int flags = fcntl(data_sock, F_GETFL, 0);
456             fcntl(data_sock, F_SETFL, flags | O_NONBLOCK);
457
458             // receive data for one second
459             time_t start_time = time(NULL);
460             while (time(NULL) - start_time <= 1) {
461                 long bytes_temp = 0;
462                 bytes_temp = recv(data_sock, buffer, sizeof(buffer), 0);

```



```

463         if (bytes_temp != 0) {
464             bytes_recived += bytes_temp;
465             strcpy(buffer_str, buffer);
466             buffer_str[B_SIZE] = '\0';
467             checksum_sum += checksum(buffer_str, B_SIZE + 1);
468             gettimeofday(&end, NULL);
469             start_time = time(NULL);
470         }
471     }
472     break;
473 }
474 }
475 close(data_sock);
476 timersub(&end, &start, &diff);
477 if (!q) {
478     printf("expected: %ld ,got: %ld\n", bytes_target, bytes_recived);
479 }
480 // compare checksum and bytes
481 if (bytes_recived != bytes_target) {
482     if (!q)
483         printf("error: did not received full data!\n");
484     else
485         printf("failure\n");
486     return 1;
487 }
488 if (checksum_target != checksum_sum) {
489     if (!q) {
490         printf("error: checksum failed\n");
491         printf("expected: %ld ,got: %ld\n", checksum_target, checksum_sum);
492     } else {
493         printf("failure\n");
494     }
495
496     return 1;
497 }
498
499 //print results
500 long microsec = diff.tv_usec;
501 long milisec = microsec / 1000;
502 milisec += diff.tv_sec * 1000;
503 printf("ipv4_tcp,%ld\n", milisec);
504 return 0;
505 }
506
507
508 int client_TCP_IPV6_B(char *ip, char *port, int info_sock, FILE *file){
509     int data_sock = socket(AF_INET6, SOCK_STREAM, 0);
510     if (data_sock < 0) {
511         perror("Error creating socket\n");
512         return 1;
513     }
514     int int_port = atoi(port);
515     struct sockaddr_in6 addr;
516     memset(&addr, 0, sizeof(addr));
517     addr.sin6_family = AF_INET6;
518     addr.sin6_port = htons(int_port);
519
520
521     // if (inet_pton(AF_INET6, ip, &addr.sin6_addr) <= 0) {
522     //     perror("inet_pton error");

```

```

523 //      return 1;
524 //  }
525
526
527 if (connect(data_sock, (struct sockaddr *)&addr, sizeof(addr)) == -1) {
528     perror("connect error");
529     return 1;
530 }
531
532
533
534 char buffer[B_SIZE];
535 size_t bytes_read;
536 char *start = "start";
537 char *end = "end";
538 send(info_sock, start, strlen(start), 0);
539 fseek(file, 0L, SEEK_SET);
540 while (1) {
541     bytes_read = fread(buffer, 1, B_SIZE, file);
542
543     if (bytes_read == 0) {
544         // End of file
545         break;
546     }
547     if(bytes_read < B_SIZE){
548         printf("%zu\n", bytes_read);
549     }
550     //buffer[bytes_read] = '\0'; // add null terminator
551     if (send(data_sock, buffer, bytes_read, 0) == -1) {
552         perror("send");
553         exit(EXIT_FAILURE);
554     }
555 }
556
557 send(info_sock, end, strlen(end), 0);
558
559 close(data_sock);
560 close(info_sock);
561 return 0;
562 }
563
564 int server_TCP_IPV6_B(char *port, int info_sock, long bytes_target, long checksum_target,
565 int q){
566     struct timeval start, end, diff;
567     int sockfd = socket(AF_INET6, SOCK_STREAM, 0);
568     if (sockfd == -1) {
569         perror("socket");
570         return 1;
571     }
572     int int_port = atoi(port);
573     struct sockaddr_in6 servaddr, cliaddr;
574     memset(&servaddr, 0, sizeof(servaddr));
575     servaddr.sin6_family = AF_INET6;
576     servaddr.sin6_addr = in6addr_any;
577     servaddr.sin6_port = htons(int_port);
578
579     if (bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1) {
580         perror("bind error");
581         return 1;
582     }

```

```

582
583     if (listen(sockfd, 2) == -1) {
584         perror("listen error");
585         return 1;
586     }
587
588     socklen_t clilen = sizeof(cliaddr);
589
590     // Accept the next incoming connection and create a new connected socket
591     int data_sock = accept(sockfd, (struct sockaddr *)&cliaddr, &clilen);
592     if (data_sock == -1) {
593         perror("accept error");
594         return 1;
595     }
596
597
598
599
600     struct pollfd fds[2];
601     fds[0].fd = info_sock;
602     fds[0].events = POLLIN; // tell me when I can read from it
603     fds[1].fd = data_sock;
604     fds[1].events = POLLIN;
605     long bytes_recived = 0;
606     long checksum_sum = 0;
607     int done = 0;
608     int started = 0;
609     char buffer[B_SIZE];
610     char buffer_str[B_SIZE + 1];
611     while (1) {
612         memset(buffer, 0, sizeof(buffer));
613         int err = poll(fds, 2, -1);
614         if (err < 0) {
615             perror("poll");
616             return 1;
617         }
618         if (fds[0].revents && POLLIN) {
619             //read from the socket
620             if (!started)
621                 recv(info_sock, buffer, strlen("start") + 1, 0);
622             else
623                 recv(info_sock, buffer, strlen("end") + 1, 0);
624             if (strcmp(buffer, "start") == 0) {
625                 gettimeofday(&start, NULL);
626                 started = 1;
627             } else if (strcmp(buffer, "end") == 0) {
628                 gettimeofday(&end, NULL);
629                 done = 1;
630             }
631         }
632         else if (fds[1].revents && POLLIN) {
633             // recive the data and count byts
634             bytes_recived += recv(data_sock, buffer, sizeof(buffer), 0);
635             strcpy(buffer_str, buffer);
636             buffer_str[B_SIZE] = '\0';
637             checksum_sum += checksum(buffer_str, B_SIZE + 1);
638         }
639
640         if (done == 1) {
641             //set the socket to non-blocking code

```

```

642     int flags = fcntl(data_sock, F_GETFL, 0);
643     fcntl(data_sock, F_SETFL, flags | O_NONBLOCK);
644
645     // receive data for one second
646     time_t start_time = time(NULL);
647     while (time(NULL) - start_time <= 1) {
648         long bytes_temp = 0;
649         bytes_temp = recv(data_sock, buffer, sizeof(buffer), 0);
650         if (bytes_temp != 0) {
651             bytes_recived += bytes_temp;
652             strcpy(buffer_str, buffer);
653             buffer_str[B_SIZE] = '\0';
654             checksum_sum += checksum(buffer_str, B_SIZE + 1);
655             gettimeofday(&end, NULL);
656             start_time = time(NULL);
657         }
658     }
659     break;
660 }
661 }
662 close(data_sock);
663 close(info_sock);
664 close(sockfd);
665
666 timersub(&end, &start, &diff);
667 if (!q) {
668     printf("expected: %ld ,got: %ld\n", bytes_target, bytes_recived);
669 }
670 // compare checksum and bytes
671 if (bytes_recived != bytes_target) {
672     if (!q)
673         printf("error: did not received full data!\n");
674     else
675         printf("failure\n");
676     return 1;
677 }
678 if (checksum_target != checksum_sum) {
679     if (!q) {
680         printf("error: checksum failed\n");
681         printf("expected: %ld ,got: %ld\n", checksum_target, checksum_sum);
682     } else {
683         printf("failure\n");
684     }
685
686     return 1;
687 }
688
689 //print results
690 long microsec = diff.tv_usec;
691 long milisec = microsec / 1000;
692 milisec += diff.tv_sec * 1000;
693 printf("ipv6_tcp,%ld\n", milisec);
694 return 0;
695 }
696
697 int client_UDP_B(char *ip, char *port, int info_sock, FILE *file) {
698
699     // open udp sock
700     struct sockaddr_in server_addr;
701     int data_sock = socket(AF_INET, SOCK_DGRAM, 0);

```

```

702     if (data_sock < 0) {
703         perror("Error creating socket\n");
704         return 1;
705     }
706     int int_port = atoi(port);
707     memset(&server_addr, 0, sizeof(server_addr));
708     server_addr.sin_family = AF_INET;
709     server_addr.sin_port = htons(int_port);
710     inet_pton(AF_INET, ip, &server_addr.sin_addr);
711
712     char buffer[B_SIZE_UDP];
713     size_t bytes_read;
714     char *start = "start";
715     char *end = "end";
716     send(info_sock, start, strlen(start), 0);
717     fseek(file, 0L, SEEK_SET);
718     while (1) {
719         bytes_read = fread(buffer, 1, B_SIZE_UDP, file);
720
721         if (bytes_read == 0) {
722             // End of file
723             break;
724         }
725         // buffer[bytes_read] = '\0'; // add null terminator
726         if (sendto(data_sock, buffer, bytes_read, 0, (struct sockaddr *) &server_addr,
727             sizeof(server_addr)) < 0) {
728             perror("Send failed\n");
729             return 1;
730         }
731         printf("sending end\n");
732         send(info_sock, end, strlen(end), 0);
733
734         close(data_sock);
735         close(info_sock);
736         return 0;
737     }
738
739     int server_UDP_B(char *port, int info_sock, long bytes_target, long checksum_target, int
740 q) {
741         struct timeval start, end, diff;
742         //open udp sock
743         struct sockaddr_in servaddr, cliaddr;
744
745         // Creating socket file descriptor
746         int data_sock = socket(AF_INET, SOCK_DGRAM, 0);
747         if (data_sock < 0) {
748             perror("socket creation failed\n");
749             return 1;
750         }
751
752         memset(&servaddr, 0, sizeof(servaddr));
753         memset(&cliaddr, 0, sizeof(cliaddr));
754         unsigned int len;
755         len = sizeof(cliaddr);
756
757         // Filling server information
758         int port_int = atoi(port);
759         servaddr.sin_family = AF_INET;

```

```

760     servaddr.sin_addr.s_addr = INADDR_ANY;
761     servaddr.sin_port = htons(port_int);
762
763     // Bind the socket with the server address
764     if (bind(data_sock, (const struct sockaddr *) &servaddr, sizeof(servaddr)) < 0) {
765         perror("bind failed\n");
766         return 1;
767     }
768
769
770     struct pollfd fds[2];
771     fds[0].fd = info_sock;
772     fds[0].events = POLLIN; // tell me when I can read from it
773     fds[1].fd = data_sock;
774     fds[1].events = POLLIN;
775     long bytes_recived = 0;
776     long checksum_sum = 0;
777     int done = 0;
778     int started = 0;
779     char buffer[B_SIZE_UDP];
780     char buffer_str[B_SIZE_UDP + 1];
781     while (1) {
782         memset(buffer, 0, sizeof(buffer));
783         int err = poll(fds, 2, -1);
784         if (err < 0) {
785             perror("poll failed\n");
786             return 1;
787         }
788         if (fds[0].revents && POLLIN) {
789             //read from the socket
790             if (!started)
791                 recv(info_sock, buffer, strlen("start") + 1, 0);
792             else
793                 recv(info_sock, buffer, strlen("end") + 1, 0);
794             if (strcmp(buffer, "start") == 0) {
795                 gettimeofday(&start, NULL);
796                 started = 1;
797             } else if (strcmp(buffer, "end") == 0) {
798                 gettimeofday(&end, NULL);
799                 done = 1;
800             }
801         }
802         else if (fds[1].revents && POLLIN) {
803
804             bytes_recived = recvfrom(data_sock, (char *) buffer, B_SIZE_UDP, 0, (struct
sockaddr *) &cliaddr, &len);
805             strcpy(buffer_str, buffer);
806             buffer_str[B_SIZE_UDP] = '\0';
807             checksum_sum += checksum(buffer_str, B_SIZE_UDP + 1);
808         }
809         if (done == 1) {
810
811             //set the socket to non-blocking code
812             int flags = fcntl(data_sock, F_GETFL, 0);
813             fcntl(data_sock, F_SETFL, flags | O_NONBLOCK);
814
815             // receive data for one second
816             time_t start_time = time(NULL);
817             while (time(NULL) - start_time <= 5) {
818                 long bytes_temp = 0;

```

```

819         bytes_temp = recvfrom(data_sock, buffer, B_SIZE_UDP, 0, (struct sockaddr
*) &cliaddr, &len);
820         if (bytes_temp > 0) {
821             bytes_recived += bytes_temp;
822             strcpy(buffer_str, buffer);
823             buffer_str[B_SIZE_UDP] = '\0';
824             checksum_sum += checksum(buffer_str, B_SIZE_UDP + 1);
825             gettimeofday(&end, NULL);
826             start_time = time(NULL);
827         }
828     }
829     break;
830 }
831 }
832 close(data_sock);
833 timersub(&end, &start, &diff);
834 if (!q) {
835     printf("expected: %ld ,got: %ld\n", bytes_target, bytes_recived);
836 }
837 // compare checksum and bytes
838 if (bytes_recived != bytes_target) {
839     if (!q)
840         printf("error: did not received full data!\n");
841     else
842         printf("failure\n");
843     return 1;
844 }
845 if (checksum_target != checksum_sum) {
846     if (!q) {
847         printf("error: checksum failed\n");
848         printf("expected: %ld ,got: %ld\n", checksum_target, checksum_sum);
849     } else {
850         printf("failure\n");
851     }
852     return 1;
853 }
854 //print results
855 long microsec = diff.tv_usec;
856 long milisec = microsec / 1000;
857 milisec += diff.tv_sec * 1000;
858 printf("ipv4_udp,%ld\n", milisec);
859 return 0;
860 }
861
862
863 int client_UDP_IPV6_B(char *ip, char *port, int info_sock, FILE *file) {
864     // open udp sock
865     int data_sock = socket(AF_INET6, SOCK_DGRAM, 0);
866     if (data_sock < 0) {
867         perror("Error creating socket\n");
868         return 1;
869     }
870     int int_port = atoi(port);
871     struct sockaddr_in6 servaddr;
872     memset(&servaddr, 0, sizeof(servaddr));
873     servaddr.sin6_family = AF_INET6;
874     servaddr.sin6_port = htons(int_port);
875     if (inet_pton(AF_INET6, ip, &servaddr.sin6_addr) <= 0) {
876         perror("inet_pton failed");
877         return 1;

```

```

878     }
879     char buffer[B_SIZE_UDP_IPV6];
880     size_t bytes_read;
881     char *start = "start";
882     char *end = "end";
883     send(info_sock, start, strlen(start), 0);
884     fseek(file, 0L, SEEK_SET);
885     while (1) {
886         bytes_read = fread(buffer, 1, B_SIZE_UDP_IPV6, file);
887
888         if (bytes_read == 0) {
889             // End of file
890             break;
891         }
892         // buffer[bytes_read] = '\0'; // add null terminator
893         if (sendto(data_sock, buffer, bytes_read, 0, (struct sockaddr *) &servaddr,
sizeof(servaddr)) < 0) {
894             perror("Send failed");
895             return 1;
896         }
897     }
898     send(info_sock, end, strlen(end), 0);
899
900     close(data_sock);
901     close(info_sock);
902     return 0;
903 }
904
905 int server_UDP_IPV6_B(char *port, int info_sock, long bytes_target, long checksum_target,
int q) {
906     struct timeval start, end, diff;
907     //open udp sock
908     struct sockaddr_in6 servaddr, cliaddr;
909
910     // Creating socket file descriptor
911     int data_sock = socket(AF_INET6, SOCK_DGRAM, 0);
912     if (data_sock < 0) {
913         perror("socket creation failed\n");
914         return 1;
915     }
916
917     memset(&servaddr, 0, sizeof(servaddr));
918     memset(&cliaddr, 0, sizeof(cliaddr));
919     unsigned int len;
920     len = sizeof(cliaddr);
921
922     // Filling server information
923     // Bind the socket to a port
924     int int_port = atoi(port);
925     memset(&servaddr, 0, sizeof(servaddr));
926     servaddr.sin6_family = AF_INET6;
927     servaddr.sin6_port = htons(int_port);
928     servaddr.sin6_addr = in6addr_any;
929     if (bind(data_sock, (const struct sockaddr *) &servaddr, sizeof(servaddr)) < 0) {
930         perror("bind failed\n");
931         return 1;
932     }
933
934     struct pollfd fds[2];
935     fds[0].fd = info_sock;

```



```

936     fds[0].events = POLLIN; // tell me when I can read from it
937     fds[1].fd = data_sock;
938     fds[1].events = POLLIN;
939     long bytes_recived = 0;
940     long checksum_sum = 0;
941     int done = 0;
942     int started = 0;
943     char buffer[B_SIZE_UDP_IPV6];
944     char buffer_str[B_SIZE_UDP_IPV6 + 1];
945     while (1) {
946         memset(buffer, 0, sizeof(buffer));
947         int err = poll(fds, 2, -1);
948         if (err < 0) {
949             perror("poll failed\n");
950             return 1;
951         }
952         if (fds[0].revents && POLLIN) {
953             //read from the socket
954             if (!started)
955                 recv(info_sock, buffer, strlen("start") + 1, 0);
956             else
957                 recv(info_sock, buffer, strlen("end") + 1, 0);
958             if (strcmp(buffer, "start") == 0) {
959                 gettimeofday(&start, NULL);
960                 started = 1;
961             } else if (strcmp(buffer, "end") == 0) {
962                 gettimeofday(&end, NULL);
963                 done = 1;
964             }
965         }
966         if (fds[1].revents && POLLIN ) {
967
968             bytes_recived += recvfrom(data_sock, (char *) buffer, B_SIZE_UDP_IPV6, 0,
(struct sockaddr *) &cliaddr,
969                                     &len);
970             strcpy(buffer_str, buffer);
971             buffer_str[B_SIZE_UDP_IPV6] = '\0';
972             checksum_sum += checksum(buffer_str, B_SIZE_UDP_IPV6 + 1);
973
974         }
975         if (done == 1) {
976             //set the socket to non-blocking code
977             int flags = fcntl(data_sock, F_GETFL, 0);
978             fcntl(data_sock, F_SETFL, flags | O_NONBLOCK);
979
980             // receive data for one second
981             time_t start_time = time(NULL);
982             while (time(NULL) - start_time <= 1) {
983
984                 long bytes_temp = 0;
985                 bytes_temp = recvfrom(data_sock, buffer, B_SIZE_UDP_IPV6, 0, (struct
sockaddr *) &cliaddr, &len);
986                 if (bytes_temp >= 0) {
987                     bytes_recived += bytes_temp;
988                     strcpy(buffer_str, buffer);
989                     buffer_str[B_SIZE_UDP_IPV6] = '\0';
990                     checksum_sum += checksum(buffer_str, B_SIZE_UDP_IPV6 + 1);
991                     gettimeofday(&end, NULL);
992                     start_time = time(NULL);
993                 }

```

```

994         }
995         break;
996     }
997 }
998 close(data_sock);
999 timersub(&end, &start, &diff);
1000 if (!q) {
1001     printf("expected: %ld ,got: %ld\n", bytes_target, bytes_recived);
1002 }
1003 // compare checksum and bytes
1004 if (bytes_recived != bytes_target) {
1005     printf("failure\n");
1006     return 1;
1007 }
1008 if (checksum_target != checksum_sum) {
1009     if(!q) {
1010         printf("error: checksum failed\n");
1011         printf("expected: %ld ,got: %ld\n", checksum_target, checksum_sum);
1012     } else{
1013         printf("failure\n");
1014     }
1015 }
1016 return 1;
1017 }
1018 //print results
1019 long microsec = diff.tv_usec;
1020 long milisec = microsec / 1000;
1021 milisec += diff.tv_sec * 1000;
1022 printf("ipv6_udp,%ld\n", milisec);
1023 return 0;
1024 }
1025
1026
1027 int client_UDS_DGRAM(int info_sock, FILE *file, long bytes_target){
1028     int data_sock;
1029     socklen_t len;
1030     char buffer[B_SIZE];
1031     char *start = "start";
1032     char *end = "end";
1033     size_t bytes_read;
1034     long total_bytes = 0;
1035
1036     struct sockaddr_un remote = {
1037         .sun_family = AF_UNIX,
1038         // .sin_path = SOCK_PATH //cant assign to an array
1039     };
1040
1041     if((data_sock = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1){
1042         perror("socket");
1043         exit(EXIT_FAILURE);
1044     }
1045
1046     strcpy(remote.sun_path, SOCK_PATH);
1047     len = strlen(remote.sun_path) + sizeof(remote.sun_family);
1048
1049     if(send(info_sock, start, strlen(start), 0) <= 0){ //start timer
1050         perror("send start");
1051         close(data_sock);
1052         close(info_sock);
1053         exit(EXIT_FAILURE);

```

```

1054     }
1055     fseek(file, 0L, SEEK_SET);
1056     while (total_bytes < bytes_target) {
1057         bytes_read = fread(buffer, 1, B_SIZE, file);
1058         total_bytes += bytes_read;
1059
1060         //buffer[bytes_read] = '\0'; // add null terminator
1061         if (sendto(data_sock, buffer, bytes_read, 0, (struct sockaddr *) &remote, len) ==
-1) {
1062             perror("sendto");
1063             close(data_sock);
1064             close(info_sock);
1065             exit(EXIT_FAILURE);
1066         }
1067     }
1068     if(send(info_sock, end, strlen(end), 0) == -1){ //end timer
1069         perror("send end");
1070         close(data_sock);
1071         close(info_sock);
1072         exit(EXIT_FAILURE);
1073     }
1074     close(data_sock);
1075     close(info_sock);
1076     return 0;
1077 }
1078 }
1079
1080 int server_UDS_DGRAM(int info_sock, long bytes_target, long checksum_target, int q){
1081     int sock, bytes = 0, done = 0, started = 0, i = 0;
1082     socklen_t local_len, remote_len;
1083     long bytes_recived = 0, checksum_sum = 0;
1084     char buffer[B_SIZE], buffer_str[B_SIZE + 1] = "";
1085
1086     struct timeval start, end, diff;
1087     struct sockaddr_un remote, local = {
1088         .sun_family = AF_UNIX,
1089         // .sin_path = SOCK_PATH //cant assign to an array
1090     };
1091
1092     strcpy(local.sun_path, SOCK_PATH);
1093     unlink(local.sun_path);
1094     local_len = strlen(local.sun_path) + sizeof(local.sun_family);
1095     remote_len = sizeof(remote);
1096
1097     if((sock = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1){
1098         perror("socket");
1099         exit(EXIT_FAILURE);
1100     }
1101
1102     if(bind(sock, (struct sockaddr *) &local, local_len) == -1){
1103         perror("bind");
1104         close(sock);
1105         exit(EXIT_FAILURE);
1106     }
1107
1108     struct pollfd fds[2];
1109     fds[0].fd = info_sock;
1110     fds[0].events = POLLIN; // tell me when I can read from it
1111     fds[1].fd = sock;
1112     fds[1].events = POLLIN;

```

```

1113
1114     while(1){
1115         memset(buffer, 0, sizeof(buffer));
1116
1117         if(poll(fds, 2, -1) < 0){
1118             perror("poll");
1119             close(sock);
1120             exit(EXIT_FAILURE);
1121         }
1122
1123         if (fds[0].revents && POLLIN) {
1124             //read from the socket
1125             if (!started){
1126                 recv(info_sock, buffer, strlen("start") + 1, 0);
1127             }else{
1128                 recv(info_sock, buffer, strlen("end") + 1, 0);
1129             }if (strcmp(buffer, "start") == 0) {
1130                 gettimeofday(&start, NULL);
1131                 started = 1;
1132             } else if (strcmp(buffer, "end") == 0) {
1133                 gettimeofday(&end, NULL);
1134                 done = 1;
1135             }
1136         }
1137
1138         if (fds[1].revents && POLLIN) {
1139             // receive the data and count byts
1140             bytes = recvfrom(sock, buffer, sizeof(buffer), 0, (struct sockaddr *)
&remote, &remote_len);
1141             if(bytes < 0){
1142                 perror("recvfrom");
1143                 close(sock);
1144                 exit(EXIT_FAILURE);
1145             }
1146
1147             bytes_recived += bytes;
1148             strcat(buffer_str, buffer);
1149             buffer_str[B_SIZE] = '\0';
1150             checksum_sum += checksum(buffer_str, B_SIZE + 1);
1151         }
1152
1153         if(done){
1154             break;
1155         }
1156     }
1157     close(sock);
1158     timersub(&end, &start, &diff);
1159     if (!q) {
1160         printf("expected: %ld ,got: %ld\n", bytes_target, bytes_recived);
1161     }
1162     // compare checksum and bytes
1163     if (bytes_recived != bytes_target) {
1164         if (!q)
1165             printf("error: did not received full data!\n");
1166         else
1167             printf("failure\n");
1168         return 1;
1169     }
1170     if (checksum_target != checksum_sum) {
1171         if (!q) {

```

```

1172         printf("error: checksum failed\n");
1173         printf("expected: %ld ,got: %ld\n", checksum_target, checksum_sum);
1174     } else {
1175         printf("failure\n");
1176     }
1177
1178     return 1;
1179 }
1180
1181 //print results
1182 long microsec = diff.tv_usec;
1183 long milisec = microsec / 1000;
1184 milisec += diff.tv_sec * 1000;
1185 printf("uds_dgram,%ld\n", milisec);
1186 return 0;
1187
1188 }
1189
1190
1191 int client_UDS_STREAM(int info_sock, FILE *file, long bytes_target){
1192     int data_sock, len;
1193     char buffer[B_SIZE];
1194     char *start = "start";
1195     char *end = "end";
1196     size_t bytes_read;
1197     long total_bytes = 0;
1198
1199     struct sockaddr_un remote = {
1200         .sun_family = AF_UNIX,
1201         // .sin_path = SOCK_PATH //cant assign to an array
1202     };
1203
1204     if((data_sock = socket(AF_UNIX, SOCK_STREAM, 0)) == -1){
1205         perror("socket");
1206         exit(EXIT_FAILURE);
1207     }
1208
1209     strcpy(remote.sun_path, SOCK_PATH);
1210     len = strlen(remote.sun_path) + sizeof(remote.sun_family);
1211     if(connect(data_sock, (struct sockaddr *) &remote, len) == -1){
1212         perror("connect");
1213         exit(EXIT_FAILURE);
1214     }
1215
1216     send(info_sock, start, strlen(start), 0); //start timer
1217     fseek(file, 0L, SEEK_SET);
1218     while (total_bytes < bytes_target) {
1219         bytes_read = fread(buffer, 1, B_SIZE, file);
1220         total_bytes += bytes_read;
1221
1222         //buffer[bytes_read] = '\0'; // add null terminator
1223         if (send(data_sock, buffer, bytes_read, 0) == -1) {
1224             perror("send");
1225             exit(EXIT_FAILURE);
1226         }
1227     }
1228     send(info_sock, end, strlen(end), 0); //end timer
1229
1230     close(data_sock);
1231     close(info_sock);

```

```
1232     return 0;
1233
1234 }
1235
1236 int server_UDS_STREAM(int info_sock, long bytes_target, long checksum_target, int q){
1237     int sock1, sock2, len, done = 0, started = 0;
1238     long bytes_recived = 0, checksum_sum = 0;
1239     char buffer[B_SIZE], buffer_str[B_SIZE + 1] = "";
1240
1241     struct timeval start, end, diff;
1242     struct sockaddr_un remote, local = {
1243         .sun_family = AF_UNIX,
1244         // .sin_path = SOCK_PATH //cant assign to an array
1245     };
1246
1247     if((sock1 = socket(AF_UNIX, SOCK_STREAM, 0)) == -1){
1248         perror("socket");
1249         exit(EXIT_FAILURE);
1250     }
1251
1252     strcpy(local.sun_path, SOCK_PATH);
1253     unlink(local.sun_path);
1254     len = strlen(local.sun_path) + sizeof(local.sun_family);
1255
1256     if(bind(sock1, (struct sockaddr *) &local, len) == -1){
1257         perror("bind");
1258         exit(EXIT_FAILURE);
1259     }
1260
1261     if(listen(sock1, 1) == -1){
1262         perror("listen");
1263         exit(EXIT_FAILURE);
1264     }
1265
1266     if((sock2 = accept(sock1, (struct sockaddr *) &remote, &len)) == -1){
1267         perror("accept");
1268         exit(EXIT_FAILURE);
1269     }
1270
1271     close(sock1);
1272
1273     struct pollfd fds[2];
1274     fds[0].fd = info_sock;
1275     fds[0].events = POLLIN; // tell me when I can read from it
1276     fds[1].fd = sock2;
1277     fds[1].events = POLLIN;
1278
1279     while(1){
1280         memset(buffer, 0, sizeof(buffer));
1281
1282         if(poll(fds, 2, -1) == -1){
1283             perror("poll");
1284             exit(EXIT_FAILURE);
1285         }
1286
1287         if (fds[0].revents && POLLIN) {
1288             //read from the socket
1289             if (!started)
1290                 recv(info_sock, buffer, strlen("start") + 1, 0);
1291             else
```

```

1292         recv(info_sock, buffer, strlen("end") + 1, 0);
1293     if (strcmp(buffer, "start") == 0) {
1294         gettimeofday(&start, NULL);
1295         started = 1;
1296     } else if (strcmp(buffer, "end") == 0) {
1297         gettimeofday(&end, NULL);
1298         done = 1;
1299     }
1300 }
1301
1302 else if (fds[1].revents && POLLIN) {
1303     // receive the data and count byts
1304     bytes_recived += recv(sock2, buffer, sizeof(buffer), 0);
1305     strcat(buffer_str, buffer);
1306     buffer_str[B_SIZE] = '\0';
1307     checksum_sum += checksum(buffer_str, B_SIZE + 1);
1308 }
1309
1310 if(done){
1311     break;
1312 }
1313 }
1314 close(sock2);
1315 timersub(&end, &start, &diff);
1316 if (!q) {
1317     printf("expected: %ld ,got: %ld\n", bytes_target, bytes_recived);
1318 }
1319 // compare checksum and bytes
1320 if (bytes_recived != bytes_target) {
1321     if (!q)
1322         printf("error: did not received full data!\n");
1323     else
1324         printf("failure\n");
1325     return 1;
1326 }
1327 if (checksum_target != checksum_sum) {
1328     if (!q) {
1329         printf("error: checksum failed\n");
1330         printf("expected: %ld ,got: %ld\n", checksum_target, checksum_sum);
1331     } else {
1332         printf("failure\n");
1333     }
1334
1335     return 1;
1336 }
1337
1338 //print results
1339 long microsec = diff.tv_usec;
1340 long milisec = microsec / 1000;
1341 milisec += diff.tv_sec * 1000;
1342 printf("uds_stream,%ld\n", milisec);
1343 return 0;
1344 }
1345
1346
1347
1348 int client_mmap(int info_sock , FILE *file, long bytes_target){
1349     int fd;
1350     char *data;
1351     long total_bytes = 0;

```

```

1352
1353     fd = fileno(file);
1354     ftruncate(fd, B_SIZE);
1355
1356     char buffer[B_SIZE];
1357     size_t bytes_read;
1358     char *start = "start";
1359     char *end = "end";
1360     if(send(info_sock, start, strlen(start), 0) == -1){
1361         perror("failed sending start");
1362         exit(EXIT_FAILURE);
1363     }
1364
1365     data = mmap(NULL, B_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
1366     if(data == MAP_FAILED){
1367         perror("failed mapping data");
1368         exit(EXIT_FAILURE);
1369     }
1370
1371     if(send(info_sock, end, strlen(end), 0) == -1){
1372         printf("failed!!!\n");
1373         perror("failed sending end");
1374         exit(EXIT_FAILURE);
1375     }
1376
1377     close(fd);
1378     close(info_sock);
1379 }
1380
1381 int server_mmap(int info_sock, long bytes_target, long checksum_target, int q){
1382     int fd, started = 0, done = 0, i = 0;
1383     long bytes_recived = 0, checksum_sum = 0;
1384     char buffer[B_SIZE], buffer_str[B_SIZE + 1], *data;
1385
1386     struct timeval start, end, diff;
1387     struct pollfd fds[2];
1388
1389     fds[0].fd = info_sock;
1390     fds[0].events = POLLIN;
1391     fds[1].fd = fd;
1392     fds[1].events = POLLIN;
1393
1394     while(1){
1395         memset(buffer, 0, sizeof(buffer));
1396         if(poll(fds, 2, -1) == -1){
1397             perror("poll failed");
1398             exit(EXIT_FAILURE);
1399         }
1400
1401         if (fds[0].revents && POLLIN) {
1402             //read from the socket
1403             if (!started){
1404                 if(recv(info_sock, buffer, strlen("start") + 1, 0) == -1){
1405                     perror("failed receiving start");
1406                     exit(EXIT_FAILURE);
1407                 }
1408             }else{
1409                 if(recv(info_sock, buffer, strlen("end") + 1, 0) == -1){
1410                     perror("failed receiving end");
1411                     exit(EXIT_FAILURE);

```



```

1412         }
1413     }
1414     if (strcmp(buffer, "start") == 0) {
1415         gettimeofday(&start, NULL);
1416         started = 1;
1417     } else if (strcmp(buffer, "end") == 0) {
1418         gettimeofday(&end, NULL);
1419         done = 1;
1420     }
1421 }
1422
1423 else if(fds[1].revents && POLLIN){
1424     data = mmap(NULL, B_SIZE, PROT_READ, MAP_SHARED, fd, 0);
1425     if(data == MAP_FAILED){
1426         perror("failed sending start");
1427         close(fd);
1428         exit(EXIT_FAILURE);
1429     }
1430     strcpy(buffer_str, data);
1431     buffer_str[B_SIZE] = '\0';
1432     checksum_sum += checksum(buffer_str, B_SIZE + 1);
1433 }
1434 if(done){
1435     break;
1436 }
1437 }
1438 close(fd);
1439 timersub(&end, &start, &diff);
1440 if (!q) {
1441     printf("expected: %ld ,got: %ld\n", bytes_target, bytes_recived);
1442 }
1443 // compare checksum and bytes
1444 if (bytes_recived != bytes_target) {
1445     if (!q)
1446         printf("error: did not received full data!\n");
1447     else
1448         printf("failure\n");
1449     return 1;
1450 }
1451 if (checksum_target != checksum_sum) {
1452     if (!q) {
1453         printf("error: checksum failed\n");
1454         printf("expected: %ld ,got: %ld\n", checksum_target, checksum_sum);
1455     } else {
1456         printf("failure\n");
1457     }
1458
1459     return 1;
1460 }
1461
1462 //print results
1463 long microsec = diff.tv_usec;
1464 long milisec = microsec / 1000;
1465 milisec += diff.tv_sec * 1000;
1466 printf("mmap,%ld\n", milisec);
1467 return 0;
1468 }
1469
1470
1471 int client_named_pipe(int info_sock ,char *fifo_name, FILE *file, long bytes_target){

```

```
1472     int fd;
1473     long total_bytes = 0;
1474
1475     mkfifo(fifo_name, 0666);
1476
1477     fd = open(fifo_name, O_WRONLY);
1478     if(fd == -1){
1479         perror("failed to open the pipe\n");
1480         exit(EXIT_FAILURE);
1481     }
1482
1483     char buffer[B_SIZE];
1484     size_t bytes_read;
1485     char *start = "start";
1486     char *end = "end";
1487     if(send(info_sock, start, strlen(start), 0) == -1){
1488         perror("failed sending start\n");
1489         exit(EXIT_FAILURE);
1490     }
1491
1492     fseek(file, 0L, SEEK_SET);
1493     while (total_bytes < bytes_target) {
1494         bytes_read = fread(buffer, 1, B_SIZE, file);
1495         total_bytes += bytes_read;
1496         write(fd, buffer, bytes_read);
1497     }
1498
1499     if(send(info_sock, end, strlen(end), 0) == -1){
1500         perror("failed sending end\n");
1501         exit(EXIT_FAILURE);
1502     }
1503
1504     close(fd);
1505     close(info_sock);
1506 }
1507
1508 int server_named_pipe(int info_sock, char *fifo_name, long bytes_target, long
checksum_target, int q){
1509     int fd, started = 0, done = 0, i = 0;
1510     long bytes_recived = 0, checksum_sum = 0;
1511     char buffer[B_SIZE], buffer_str[B_SIZE + 1];
1512
1513     mkfifo(fifo_name, 0666);
1514
1515     fd = open(fifo_name, O_RDONLY);
1516     if(fd == -1){
1517         perror("failed to open the pipe\n");
1518         exit(EXIT_FAILURE);
1519     }
1520
1521     struct timeval start, end, diff;
1522     struct pollfd fds[2];
1523
1524     fds[0].fd = info_sock;
1525     fds[0].events = POLLIN;
1526     fds[1].fd = fd;
1527     fds[1].events = POLLIN;
1528
1529     while(1){
1530         memset(buffer, 0, sizeof(buffer));
```

```

1531     if(poll(fds, 2, -1) == -1){
1532         perror("poll failed\n");
1533         exit(EXIT_FAILURE);
1534     }
1535
1536     if (fds[0].revents && POLLIN) {
1537         //read from the socket
1538         if (!started){
1539             if(recv(info_sock, buffer, strlen("start") + 1, 0) == -1){
1540                 perror("failed receiving start\n");
1541                 exit(EXIT_FAILURE);
1542             }
1543         }else{
1544             if(recv(info_sock, buffer, strlen("end") + 1, 0) == -1){
1545                 perror("failed receiving end\n");
1546                 exit(EXIT_FAILURE);
1547             }
1548         }
1549         if (strcmp(buffer, "start") == 0) {
1550             gettimeofday(&start, NULL);
1551             started = 1;
1552         } else if (strcmp(buffer, "end") == 0) {
1553             gettimeofday(&end, NULL);
1554             done = 1;
1555         }
1556     }
1557
1558     else if(fds[1].revents && POLLIN){
1559         while(1){
1560             int bytes = read(fd, buffer, sizeof(buffer));
1561
1562             if(bytes > 0){
1563                 bytes_recived += bytes;
1564             }
1565             else if(bytes == 0){
1566                 break;
1567             }
1568             else{
1569                 perror("error while readinf the file\n");
1570                 exit(EXIT_FAILURE);
1571             }
1572         }
1573         strcpy(buffer_str, buffer);
1574         buffer_str[B_SIZE] = '\0';
1575         checksum_sum += checksum(buffer_str, B_SIZE + 1);
1576     }
1577     if(done){
1578         break;
1579     }
1580 }
1581 close(fd);
1582 timersub(&end, &start, &diff);
1583 if (!q) {
1584     printf("expected: %ld ,got: %ld\n", bytes_target, bytes_recived);
1585 }
1586 // compare checksum and bytes
1587 if (bytes_recived != bytes_target) {
1588     if (!q)
1589         printf("error: did not received full data!\n");
1590     else

```

```

1591         printf("failure\n");
1592     return 1;
1593 }
1594 if (checksum_target != checksum_sum) {
1595     if (!q) {
1596         printf("error: checksum failed\n");
1597         printf("expected: %ld ,got: %ld\n", checksum_target, checksum_sum);
1598     } else {
1599         printf("failure\n");
1600     }
1601
1602     return 1;
1603 }
1604
1605 //print results
1606 long microsec = diff.tv_usec;
1607 long milisec = microsec / 1000;
1608 milisec += diff.tv_sec * 1000;
1609 printf("pipe,%ld\n", milisec);
1610 return 0;
1611 }
1612
1613
1614 int server_B(char *port, int q) {
1615     char info_port[6];
1616     port_for_info(port, info_port);
1617     int info_sock = tcp_server_conn(info_port);
1618
1619     char message[100] = {'\0'};
1620
1621     recv(info_sock, message, sizeof(message), 0);
1622
1623     char *type = strtok(message, ",");
1624     char *param = strtok(NULL, ",");
1625     char *checksum_target = strtok(NULL, ",");
1626     char *bytes_target = strtok(NULL, ",");
1627
1628     long bytes_target_long = atol(bytes_target);
1629     long checksum_target_long = atol(checksum_target);
1630
1631     int ret = 0;
1632     if (!q) {
1633         printf("type: %s\n", type);
1634         printf("param: %s\n", param);
1635     }
1636     if (strcmp(type, "ipv4") == 0) {
1637         if (strcmp(param, "tcp") == 0) {
1638             ret = server_TCP_B(port, info_sock, bytes_target_long, checksum_target_long,
1639 q);
1640         } else if (strcmp(param, "udp") == 0) {
1641             ret = server_UDP_B(port, info_sock, bytes_target_long, checksum_target_long,
1642 q);
1643         }
1644     } else if (strcmp(type, "ipv6") == 0) {
1645         if (strcmp(param, "tcp") == 0) {
1646             server_TCP_IPV6_B(port, info_sock, bytes_target_long, checksum_target_long, q);
1647         } else if (strcmp(param, "udp") == 0) {
1648             ret = server_UDP_IPV6_B(port, info_sock, bytes_target_long,
checksum_target_long, q);

```

```

1649     }
1650
1651     } else if (strcmp(type, "uds") == 0) {
1652         if (strcmp(param, "dgram") == 0) {
1653             ret = server_UDS_DGRAM(info_sock, bytes_target_long, checksum_target_long,
1654 q);
1655         } else if (strcmp(param, "stream") == 0) {
1656             ret = server_UDS_STREAM(info_sock, bytes_target_long, checksum_target_long,
1657 q);
1658         }
1659     } else if (strcmp(type, "mmap") == 0) {
1660         ret = server_mmap(info_sock, bytes_target_long, checksum_target_long, q);
1661     } else if (strcmp(type, "pipe") == 0) {
1662         ret = server_named_pipe(info_sock, FIFO_NAME, bytes_target_long,
1663 checksum_target_long, q);
1664     } else {
1665         perror("wrong parameters\n");
1666         return 1;
1667     }
1668     close(info_sock);
1669     return ret;
1670 }
1671
1672 int main(int argc, char *argv[]) {
1673     int is_tcp = 0;
1674     int is_udp = 0;
1675     int is_ip = 0;
1676     int is_port = 0;
1677     int is_ipv4 = 0;
1678     int is_ipv6 = 0;
1679     int is_uds = 0;
1680     int is_dgram = 0;
1681     int is_stream = 0;
1682     int is_mmap = 0;
1683     int is_pipe = 0;
1684     int is_c = 0;
1685     int is_p = 0;
1686     int is_q = 0;
1687     int is_server = 0;
1688     char file_name[50] = {'\0'};
1689     char ip[40] = {'\0'};
1690     char port[10] = {'\0'};
1691     if (argc < 3) {
1692         usage();
1693         exit(EXIT_FAILURE);
1694     }
1695     for (int i = 1; i < argc; ++i) {
1696         if (strcmp(argv[i], "-c") == 0) {
1697             is_c = 1;
1698         } else if (strcmp(argv[i], "-s") == 0) {
1699             is_server = 1;
1700         } else if (strcmp(argv[i], "-p") == 0) {
1701             is_p = 1;
1702         } else if (strcmp(argv[i], "-q") == 0) {
1703             is_q = 1;
1704         } else if (strcmp(argv[i], "pipe") == 0) {
1705             is_pipe = 1;
1706             if (i + 1 >= argc) {

```

```

1707         usage();
1708         exit(EXIT_FAILURE);
1709     }
1710     strcpy(file_name, argv[i + 1]);
1711     i++;
1712 } else if (strcmp(argv[i], "mmap") == 0) {
1713     is_mmap = 1;
1714     if (i + 1 >= argc) {
1715         usage();
1716         exit(EXIT_FAILURE);
1717     }
1718     strcpy(file_name, argv[i + 1]);
1719     i++;
1720 } else if (strcmp(argv[i], "uds") == 0) {
1721     is_uds = 1;
1722 } else if (strcmp(argv[i], "ipv6") == 0) {
1723     is_ipv6 = 1;
1724 } else if (strcmp(argv[i], "ipv4") == 0) {
1725     is_ipv4 = 1;
1726 } else if (strcmp(argv[i], "tcp") == 0) {
1727     is_tcp = 1;
1728 } else if (strcmp(argv[i], "udp") == 0) {
1729     is_udp = 1;
1730 } else if (strstr(argv[i], ".") != NULL) {
1731     is_ip = 1;
1732     strcpy(ip, argv[i]);
1733 } else if (strstr(argv[i], ":") != NULL) {
1734     is_ip = 1;
1735     strcpy(ip, argv[i]);
1736 } else if (strcmp(argv[i], "dgram") == 0) {
1737     is_dgram = 1;
1738 } else if (strcmp(argv[i], "stream") == 0) {
1739     is_stream = 1;
1740 } else {
1741     is_port = 1;
1742     strcpy(port, argv[i]);
1743 }
1744 }
1745 if (!is_p) { //part A
1746     if (is_c) { //client A
1747         if (argc != 4) {
1748             usage();
1749             exit(EXIT_FAILURE);
1750         }
1751         if (!is_port || !is_ip) {
1752             usage();
1753             exit(EXIT_FAILURE);
1754         }
1755         if (client_A(port, ip) != 0)
1756             exit(EXIT_FAILURE);
1757     }
1758     else if (is_server) { // server A
1759         if (argc != 3) {
1760             usage();
1761             exit(EXIT_FAILURE);
1762         }
1763         if (!is_port) {
1764             usage();
1765             exit(EXIT_FAILURE);
1766         }

```

```

1767         if (server_A(port) != 0)
1768             exit(EXIT_FAILURE);
1769     } else {
1770         usage();
1771         exit(EXIT_FAILURE);
1772     }
1773 } else { // part B
1774     if (is_c) { // client side
1775         FILE *file;
1776         file = fopen("100MB.bin", "ab+");
1777         if (file == NULL) {
1778             perror("File open failed\n");
1779             exit(EXIT_FAILURE);
1780         }
1781         long bytes_count = 0;
1782         long checksum = checksum_file(file, &bytes_count);
1783
1784         char new_port[6];
1785         port_for_info(port, new_port);
1786         char info_ip[40] = {'\0'};
1787         if (is_ipv6)
1788             ipv6_to_ipv4(ip, info_ip);
1789         else
1790             strcpy(info_ip, ip);
1791         int info_sock = tcp_client_conn(info_ip, new_port);
1792
1793
1794         if (is_ipv4 && is_tcp) { // ipv4 - tcp
1795             //socket to notify the receiver to start timing
1796             type_param(info_sock, "ipv4", "tcp", checksum, bytes_count);
1797             client_TCP_B(ip, port, info_sock, file);
1798
1799         } else if (is_ipv4 && is_udp) { // ipv4 - udp
1800             //socket to notify the receiver to start timing
1801             type_param(info_sock, "ipv4", "udp", checksum, bytes_count);
1802             client_UDP_B(ip, port, info_sock, file);
1803         } else if (is_ipv6 && is_tcp) { // ipv6- tcp
1804             //socket to notify the receiver to start timing
1805             type_param(info_sock, "ipv6", "tcp", checksum, bytes_count);
1806             client_TCP_IPV6_B(ip, port, info_sock, file);
1807         } else if (is_ipv6 && is_udp) { // ipv6 - udp
1808             //socket to notify the receiver to start timing
1809             type_param(info_sock, "ipv6", "udp", checksum, bytes_count);
1810             client_UDP_IPV6_B(ip, port, info_sock, file);
1811
1812         } else if (is_uds && is_dgram) { // uds dgram
1813             //socket to notify the receiver to start timing
1814             type_param(info_sock, "uds", "dgram", checksum, bytes_count);
1815             client_UDS_DGRAM(info_sock, file, bytes_count);
1816
1817         } else if (is_uds && is_stream) { // uds stream
1818             //socket to notify the receiver to start timing
1819             type_param(info_sock, "uds", "stream", checksum, bytes_count);
1820             client_UDS_STREAM(info_sock, file, bytes_count);
1821
1822         } else if (is_mmap) { // mmap
1823             //socket to notify the receiver to start timing
1824             type_param(info_sock, "mmap", "filename", checksum, bytes_count);
1825             client_mmap(info_sock, file, bytes_count);
1826

```

```
1827     } else if (is_pipe) { // pipe
1828         //socket to notify the receiver to start timing
1829         type_param(info_sock, "pipe", "filename", checksum, bytes_count);
1830         client_named_pipe(info_sock, FIFO_NAME, file, bytes_count);
1831
1832     } else {
1833         fclose(file);
1834         usage();
1835         exit(EXIT_FAILURE);
1836     }
1837     fclose(file);
1838 } else if (is_server) {
1839     server_B(port, is_q); //continue as long as server_B returns 0;
1840     return 0;
1841 } else {
1842     usage();
1843     exit(EXIT_FAILURE);
1844 }
1845 }
1846 return 0;
1847 }
```

## Assignments\Assignment3\makefile

```
1 all: stnc
2
3 clean:
4     rm -f *.o OS_EX3_pipe stnc
5     .PHONY: all clean
6
7 stnc:stnc.c
8     gcc stnc.c -o stnc
9
10
11
```