```c
/* A simple echo server using TCP */
#include <arpa/inet.h>
#include <dirent.h>
#include <errno.h>
#include <fcntl.h>
#include <netdb.h>
#include <netinet/in.h>
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <poll.h>
#include <unistd.h>
#include <time.h>

#include "parser.h"
#include "media_transfer.h"
#include "queue.h"
#include "linked_list.h"

#define SERVER_TCP_PORT     3000    /* well-known port */
#define HEADERLEN           256     /* header packet length */
#define CONFIG_BUFFER       256     /* length of buffer for config line */

typedef enum sched_type {
    FIFO,
    RANDOM,
    SJF
} sched_type_e;

typedef struct hanlder_arg {
    int port;                       /* port number server is running on */
    int client_socket;              /* port number of client to deal with */
} handler_arg_t;

typedef struct {
    int sd;                         /* server socket descriptor */
    int port;                       /* port number server will listen on */
    int num_threads;                /* no.of threads - can be modified using CL Arg #3 */
    int max_requests;               /* max no.of client requests server can have at any
    time - can be modified using CL Arg number #4 */
    char * directory;               /* place to look for media files */
    pthread_t *handlers;            /* array of threads server can handle client
    requests */
    pthread_mutex_t lock;           /* mutex lock to do some thread safe
    functionanlities */
    pthread_cond_t cond;            /* condition variable */
    Queue *job_queue;               /* process client request queue */
    list *job_list;
    sched_type_e scheduling_type;   /* FIFO or RANDOM processing */
} server_config_t;

server_config_t config; // lets make config global.


/*
 *  @param config: struct to store config value from rc script
 *  @param confgirc: file to read config information from
 *  @returns: success - 1 or failure - 0
 *  reads config file in to config struct
 */
int parse_configuration(server_config_t *config, char *configrc);

/*
 *  @param config: server configurtion struct
```

```
 *  @returns number of chars printed
 *  prints server configuration summary
 */
int print_configuration(server_config_t *config);

/*
 * @param filepath - name of the file for which extension is needed
 * @returns
 *      point to first char in extension
 */
const char *get_file_ext(const char *filename);

/*
 * @param config: servert config struct
 * initializes threads, and locks for config struct
 */
int initialize_thread_pool(server_config_t *config);

/*
 * @param arg - handler args passing
 * @returns
 *      1 if client wants to disconnect
 *      0 if client wants to continue
 * Fulfils client requests
 *
 */
void handle_request(void*arg);

void get_sjf (void *arg);
void handle_sjf (void *client_request);

/*
 * @param arg - server config arg will be passed
 * takes a job from job queue.
 */
void *watch_requests(void *arg);

int main(int argc, char * argv[]) {
    char * config_file = NULL;

    switch(argc) {
    case 1:
        config_file = "mserver.config";
        break;
    case 3:
        if (strcmp(argv[1], "-c") == 0) config_file = argv[2];
        break;
    }


    /* seed for random generator */
    srand(time(0));

    /* init server configuration */
    char pwd[BUFLEN];
    getcwd(pwd, BUFLEN);

    /* fill in default config */
    config.port = SERVER_TCP_PORT;
    config.directory = pwd;
    config.num_threads = 4;
    config.max_requests = 10;
    config.scheduling_type = RANDOM;

    /* override default config if file provided */
    if (argc == 3) {
        switch(parse_configuration(&config, config_file)) {
            case -1:
                fprintf(stderr, "Unable to open configuration file!\n");
```

```
                break;
            case -2:
                fprintf(stderr, "Configuration error!\n");
                break;
        }
    } else {
        parse_configuration(&config, config_file);
    }
    config.handlers  = (pthread_t*)malloc(sizeof(pthread_t)*(config.num_threads));

    if (config.scheduling_type == SJF) {
        config.job_list = create_list();
        config.job_queue = NULL;
    } else {
        config.job_queue = createQueue(config.max_requests);
        config.job_list = NULL;
    }

    /* switch current working dir to media dir */
    int ret = chdir(config.directory);
    if(ret != 0) {
        printf("cannot change to dir %s.\n", config.directory);
        exit(1);
    }

    struct sockaddr_in server;

    /* Create a stream socket */
    if ((config.sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        fprintf(stderr, "Can't create a socket\n");
        exit(1);
    }

    /* Bind an address to the socket */
    bzero((char *)&server, sizeof(struct sockaddr_in));
    server.sin_family = AF_INET;
    server.sin_port = htons(config.port);
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(config.sd, (struct sockaddr *)&server, sizeof(server)) == -1) {
        fprintf(stderr, "Can't bind name to socket\n");
        exit(1);
    }
    config.scheduling_type = SJF;
    /* initialize threads and mutex locks */
    initialize_thread_pool(&config);

    /* print sever configuration */
    print_configuration(&config);

    /* queue up to config.max_requests connect requests */
    listen(config.sd, config.max_requests);

    /* loop forever and add requests to queue */
    while(1) {

        /* get a new connection req on server socket*/
        int new_client_sd = accept(config.sd, NULL, NULL);

        /* if found a request, enqueue it for processing */
        if(new_client_sd > 0) {
            char enqueue_time[TIME_BUFFER_LEN];
            get_time_spec_to_string(enqueue_time, TIME_BUFFER_LEN);
            printf("\n%s: Main: Accepting New Connection: %d\n", enqueue_time,
            new_client_sd);

            handler_arg_t *arg = (handler_arg_t*)malloc(sizeof(handler_arg_t));
            arg->port = config.port;
            arg->client_socket = new_client_sd;

```

```
204                 printf("%s: Main: Adding New Client to the Job queue...\n", enqueue_time);
205                 /* Locks the queue to add job */
206                 pthread_mutex_lock(&(config.lock));
207 
208                 /* add connectiong to queue */
209 
210                 if (config.scheduling_type == SJF) {
211                     get_sjf(arg);
212                 } else
213                     enqueue(config.job_queue, (void*) arg);
214 
215                 /* give up the lock on the queue */
216                 pthread_mutex_unlock(&(config.lock));
217 
218                 get_time_spec_to_string(enqueue_time, TIME_BUFFER_LEN);
219                 printf("%s: Main: Added New Client to the Job queue\n", enqueue_time);
220 
221             }
222         }
223     }
224 
225     int parse_configuration(server_config_t *config, char *configrc) {
226         if (configrc == NULL) return -1;
227         if (config == NULL) config = malloc(sizeof(server_config_t));
228 
229         FILE * c_file = fopen(configrc, "r");
230         if (c_file == NULL) return -1;
231 
232         char buf[CONFIG_BUFFER];
233 
234         while (fgets(buf, CONFIG_BUFFER, c_file) != NULL) {
235             if (strstr(buf, "#")) *(strstr(buf, "#")) = '\0';
236             if (strstr(buf, "\n")) *(strstr(buf, "\n")) = '\0';
237             if (buf[0] == '\0') continue;
238             if (strstr(buf, ":") == NULL) return -2;
239 
240             char *split = strstr(buf, ": ");
241             *split = '\0';
242 
243             char *key = buf;
244             char *value = split + 2;
245 
246             // Fast and loose config parsing, only checking to see if config
247             // line contains the key value, thus, if you have something like:
248             // PortNumThreads: 5, it will match to PortNum and nothing else.
249             if (strstr(key, "PortNum")) config->port = atoi(value);
250             else if (strstr(key, "Threads")) config->num_threads = atoi(value);
251             else if (strstr(key, "Sched")) {
252                 if (strcmp(value, "FIFO") == 0) config->scheduling_type = FIFO;
253                 else if (strcmp(value, "Random") == 0) config->scheduling_type = RANDOM;
254                 else if (strcmp(value, "SJF") == 0) config->scheduling_type = SJF;
255             } else if (strstr(key, "Directory")) {
256                 config->directory = malloc(strlen(value));
257                 strcpy(config->directory, value);
258             }
259         }
260     }
261 
262     int print_configuration(server_config_t *config) {
263         printf("*******Server configuration*******\n");
264         printf("Port Number: %d\n", config->port);
265         printf("Num Threads: %d\n", config->num_threads);
266         printf("Max Reqs: %d\n", config->max_requests);
267         printf("Media Path: %s \n", config->directory);
268         printf("Sched type: %d\n", config->scheduling_type);
269         printf("***********************************\n");
270         return 0;
271     }
272 
```

```
273  const char *get_file_ext(const char *filename) {
274      const char *dot_loc = strrchr(filename, '.');
275      if(!dot_loc || dot_loc == filename) {
276          return "Unknown";
277      }
278      return dot_loc + 1;
279  }
280
281  int initialize_thread_pool(server_config_t *config) {
282      if (pthread_mutex_init(&(config->lock), NULL) != 0) {
283          printf("\n mutex init has failed\n");
284          return -1;
285      }
286      int i;
287      for (i = 0; i < config->num_threads; ++i) {
288          if(pthread_create(&(config->handlers[i]), NULL, watch_requests, (void*)config)
          != 0) {
289              printf("Failed to create a thread");
290              exit(1);
291          }
292      }
293      return 0;
294  }
295
296  void *watch_requests(void *arg) {
297
298      server_config_t *config = (server_config_t*)arg;
299
300      void *job = NULL;
301
302      while(1) {
303
304          pthread_mutex_lock(&(config->lock));
305
306          if (config->scheduling_type == SJF) {
307              job = get_job(config->job_list);
308          }
309          else if(!isEmpty(config->job_queue)) {
310              if(config->scheduling_type == FIFO) {
311                  job = dequeue(config->job_queue);
312              }
313              else if (config->scheduling_type == RANDOM) {
314                  job = random_dequeue(config->job_queue);
315              }
316          }
317
318          pthread_mutex_unlock(&(config->lock));
319
320          if(job) {
321              char time_processing_start[TIME_BUFFER_LEN];
322              get_time_spec_to_string(time_processing_start, TIME_BUFFER_LEN);
323              printf("%s: Watch Request: Thead %lu: Handling client %d\n",
              time_processing_start, pthread_self(), ((handler_arg_t*)job)->client_socket);
324
325              if (config->scheduling_type == SJF)
326                  handle_sjf(job);
327              else
328                  handle_request(job);
329          }
330
331          job = NULL;
332      }
333  }
334
335  void get_sjf (void *arg)
336  {
337      /* Some vairable declaration */
338      char time_buf[TIME_BUFFER_LEN];
339      handler_arg_t* info = ((handler_arg_t*)arg);
```

```
340
341         /* Print out client information */
342         struct sockaddr_in client_socket_addr;
343         socklen_t len;
344         len = sizeof(client_socket_addr);
345         char client_ip[32];
346         unsigned int ephemeral_port;
347
348         bzero(&client_socket_addr, len);
349
350         if (getsockname(info->client_socket, (struct sockaddr *)&client_socket_addr, &len)
            == 0) {
351             /* get ip and the temp port*/
352             inet_ntop(AF_INET, &client_socket_addr.sin_addr, client_ip, sizeof(client_ip));
353             ephemeral_port = ntohs(client_socket_addr.sin_port);
354
355             /* print contents of ss*/
356             get_time_spec_to_string(time_buf, TIME_BUFFER_LEN);
357             printf("%s: Handle Request: Client IP: %s Ephemeral Port: %d\n", time_buf,
                client_ip, ephemeral_port);
358             fflush(stdout);
359         }
360
361             char buf[BUFLEN] = {0};
362             char *bp = buf;
363             int  bytes_to_read = BUFLEN;
364             int  n = 0;
365             while ((n = read(info->client_socket, bp, bytes_to_read)) > 0) {
366                 bp += n;
367                 bytes_to_read -= n;
368             }
369
370             if (bp <= 0) {
371                 // client probably disconnected
372                 close(info->client_socket);
373             }
374             get_time_spec_to_string(time_buf, BUFLEN);
375             printf("%s: Handle Request: Client IP: %s Ephemeral Port: %d : Command Recevied
                string: %s", time_buf, client_ip, ephemeral_port, buf);
376
377             /* put a null character at the end */
378             int size = strlen(buf);
379             buf[strcspn(buf, "\n")] = 0;
380
381             char *job = malloc(strlen(buf));
382             strcpy(job, buf);
383
384             switch(get_command_from_request(buf)) {
385                 case LIST:
386                     add_job(config.job_list, job, 1, arg);
387                     break;
388                 case GET: {
389                     FILE *fp = fopen(&(buf[4]), "rb");
390
391                     if (fp == NULL) {
392                         break;
393                     }
394
395                     fseek(fp, 0L, SEEK_END);
396                     size_t len = ftell(fp);
397                     fseek(fp, 0L, SEEK_SET);
398                     fclose(fp);
399
400                     add_job(config.job_list, job, len, arg);
401                     break;
402                 }
403                 default:
404                     add_job(config.job_list, job, 0, arg);
405                     break;
```

```
406             }
407     }
408
409     void handle_sjf (void *client_request)
410     {
411         char time_buf[TIME_BUFFER_LEN];
412
413         struct node *req = (struct node *)client_request;
414         handler_arg_t* info = ((handler_arg_t*)req->owner);
415
416         /* Print out client information */
417         struct sockaddr_in client_socket_addr;
418         socklen_t len;
419         len = sizeof(client_socket_addr);
420         char client_ip[32];
421         unsigned int ephemeral_port;
422
423         bzero(&client_socket_addr, len);
424
425         if (getsockname(info->client_socket, (struct sockaddr *)&client_socket_addr, &len)
            == 0) {
426             /* get ip and the temp port*/
427             inet_ntop(AF_INET, &client_socket_addr.sin_addr, client_ip, sizeof(client_ip));
428             ephemeral_port = ntohs(client_socket_addr.sin_port);
429
430             /* print contents of ss*/
431             get_time_spec_to_string(time_buf, TIME_BUFFER_LEN);
432             printf("%s: Handle Request: Client IP: %s Ephemeral Port: %d\n", time_buf,
                client_ip, ephemeral_port);
433             fflush(stdout);
434         }
435
436         char *job = (char *)req->job;
437
438         switch(get_command_from_request(job)) {
439             case LIST: {
440                 char listing[1024];
441                 get_media_list(".", listing, 1024);
442                 // send the header packet
443                 send_header(info->client_socket, info->port, strlen(listing), "Text", 100);
444                 if(send(info->client_socket, listing, strlen(listing), 0) == -1) {
445                     get_time_spec_to_string(time_buf, TIME_BUFFER_LEN);
446                     printf("%s: Handle Request: Client IP: %s Ephemeral Port: %d : Error
                        sending list\n", time_buf, client_ip, ephemeral_port);
447                 }
448                 break;
449             }
450             case GET: {
451                 // get the length of the file needed to be read.
452                 FILE *fp = fopen(&(job[4]), "rb");
453
454                 if (fp == NULL) {
455                     send_header(info->client_socket, info->port, 0, "", 404);
456                     break;
457                 }
458
459                 fseek(fp, 0L, SEEK_END);
460                 size_t len = ftell(fp);
461                 fseek(fp, 0L, SEEK_SET);
462                 fclose(fp);
463
464                 // get file extension
465                 const char *extension = get_file_ext(job + 4);
466
467                 // send header information
468                 send_header(info->client_socket, info->port, len, extension, 100);
469
470                 get_time_spec_to_string(req->job, TIME_BUFFER_LEN);
471                 printf("%s: Handle Request: Client IP: %s Ephemeral Port: %d : Sent Header
```

```
                Information\n", time_buf, client_ip, ephemeral_port);
472
473             // send requested media
474             send_media(info->client_socket, job + 4, len);
475
476             get_time_spec_to_string(time_buf, TIME_BUFFER_LEN);
477             printf("%s: Handle_Request: Client IP: %s Ephemeral Port: %d : Sent: %s\n",
                time_buf, client_ip, ephemeral_port, job);
478             break;
479         }
480         case EXIT:
481             close(info->client_socket);
482             get_time_spec_to_string(time_buf, TIME_BUFFER_LEN);
483             printf("%s: Handle_Request: Client IP: %s Ephemeral Port: %d : Closed
                connection with client: %d\n", time_buf, client_ip, ephemeral_port,
                info->client_socket);
484             return ;
485         default:
486             // invalid request header
487             send_header(info->client_socket, info->port, 0, "", 301);
488             get_time_spec_to_string(time_buf, TIME_BUFFER_LEN);
489             printf("%s: Handle_Request: Client IP: %s Ephemeral Port: %d : Invalid
                request\n", time_buf, client_ip, ephemeral_port);
490         break;
491     }
492
493 }
494
495 void handle_request(void *client_sd)
496 {
497     /* Some vairable declaration */
498     char time_buf[TIME_BUFFER_LEN];
499     handler_arg_t* info = ((handler_arg_t*)client_sd);
500
501     /* Print out client information */
502     struct sockaddr_in client_socket_addr;
503     socklen_t len;
504     len = sizeof(client_socket_addr);
505     char client_ip[32];
506     unsigned int ephemeral_port;
507
508     bzero(&client_socket_addr, len);
509
510     if (getsockname(info->client_socket, (struct sockaddr *)&client_socket_addr, &len)
    == 0) {
511         /* get ip and the temp port*/
512         inet_ntop(AF_INET, &client_socket_addr.sin_addr, client_ip, sizeof(client_ip));
513         ephemeral_port = ntohs(client_socket_addr.sin_port);
514
515         /* print contents of ss*/
516         get_time_spec_to_string(time_buf, TIME_BUFFER_LEN);
517         printf("%s: Handle Request: Client IP: %s Ephemeral Port: %d\n", time_buf,
            client_ip, ephemeral_port);
518         fflush(stdout);
519     }
520
521     while(1) {
522         char buf[BUFLEN] = {0};
523         char *bp = buf;
524         int  bytes_to_read = BUFLEN;
525         int  n = 0;
526         while ((n = read(info->client_socket, bp, bytes_to_read)) > 0) {
527             bp += n;
528             bytes_to_read -= n;
529         }
530
531         if (bp <= 0) {
532             // client probably disconnected
533             close(info->client_socket);
```

```
534             }
535             get_time_spec_to_string(time_buf, BUFLEN);
536             printf("%s: Handle_Request: Client IP: %s Ephemeral Port: %d : Command Recevied
                string: %s", time_buf, client_ip, ephemeral_port, buf);
537
538             /* put a null character at the end */
539             int size = strlen(buf);
540             buf[strcspn(buf, "\n")] = 0;
541
542             switch(get_command_from_request(buf)) {
543                 case LIST: {
544                     char listing[1024];
545                     get_media_list(".", listing, 1024);
546                     // send the header packet
547                     send_header(info->client_socket, info->port, strlen(listing), "Text",
                        100);
548                     if(send(info->client_socket, listing, strlen(listing), 0) == -1) {
549                         get_time_spec_to_string(time_buf, TIME_BUFFER_LEN);
550                         printf("%s: Handle_Request: Client IP: %s Ephemeral Port: %d :
                            Error sending list\n", time_buf, client_ip, ephemeral_port);
551                     }
552                     break;
553                 }
554                 case GET: {
555                     // get the length of the file needed to be read.
556                     FILE *fp = fopen(&(buf[4]), "rb");
557
558                     if (fp == NULL) {
559                         send_header(info->client_socket, info->port, 0, "", 404);
560                         break;
561                     }
562
563                     fseek(fp, 0L, SEEK_END);
564                     size_t len = ftell(fp);
565                     fseek(fp, 0L, SEEK_SET);
566                     fclose(fp);
567
568                     // get file extension
569                     const char *extension = get_file_ext(buf + 4);
570
571                     // send header information
572                     send_header(info->client_socket, info->port, len, extension, 100);
573
574                     get_time_spec_to_string(time_buf, TIME_BUFFER_LEN);
575                     printf("%s: Handle_Request: Client IP: %s Ephemeral Port: %d : Sent
                        Header Information\n", time_buf, client_ip, ephemeral_port);
576
577                     // send requested media
578                     send_media(info->client_socket, buf + 4, len);
579
580                     get_time_spec_to_string(time_buf, TIME_BUFFER_LEN);
581                     printf("%s: Handle_Request: Client IP: %s Ephemeral Port: %d : Sent:
                        %s\n", time_buf, client_ip, ephemeral_port, buf);
582                     break;
583                 }
584                 case EXIT:
585                     close(info->client_socket);
586                     get_time_spec_to_string(time_buf, TIME_BUFFER_LEN);
587                     printf("%s: Handle_Request: Client IP: %s Ephemeral Port: %d : Closed
                        connection with client: %d\n", time_buf, client_ip, ephemeral_port,
                        info->client_socket);
588                     return ;
589                 default:
590                     // invalid request header
591                     send_header(info->client_socket, info->port, 0, "", 301);
592                     get_time_spec_to_string(time_buf, TIME_BUFFER_LEN);
593                     printf("%s: Handle_Request: Client IP: %s Ephemeral Port: %d : Invalid
                        request\n", time_buf, client_ip, ephemeral_port);
594                 break;
```

```
595             }
596         }
597     }
598
```