**Project 2 Reevaluation:**
We have commented "// CHANGE" into the places where change was made. Here we'll explain each change and the reason behind.

**shared.h** :
We didn't make any changes.

**CHANGE 1**
Previously we were using chars for semNo without terminating them with null.
Even though this will not affect semNo most of the time, there was a corner case where semNo[1] was also nonempty and would affect the name given to semaphore. This change was made to both **client.c** (starts at *line 24*) and **server.c** (starts at *line 45*).

```
for (i = 0; i < N + 1; i++) {
    char semNo = i + '0';
    strcpy(sem_name[i], argv[3]);
    strcat(sem_name[i], &semNo); /
    //sem_unlink(sem_name[i]);
```

Figure 1.1 Before change

```
for (i = 0; i < N + 1; i++) {
    char* semNo =(char*)malloc(2); // CHANGE 1
    semNo[0] = i + '0';
    semNo[1] = '\0';
    strcpy(sem_name[i], argv[3]);
    strcat(sem_name[i], semNo); //IF THAT CAST
    //sem_unlink(sem_name[i]);
```

Figure 1.2 After change

**CHANGE 2**
We didn't use semaphores when server was checking to see if request queue was empty before reevaluation. In some cases (1-2 out of all 20 tests in test case 1), this used to cause the server to take a default request with an empty keyword and search the file regarding that, which resulted in printing all 4 lines "1\n 2\n 3\n 4\n" in Test case 1. We didn't happen to see the problematic case in our own test cases before we submitted the program. After being aware of this possibility, we have implemented a concurrency safe version as seen below. This change was made to **server.c** (starts at *line 111*).

```
while (1) {
    fflush(stdout);
    //Wait for a request to arrive
    while (sdp->requestIn == sdp->requestOut) {
        //printf("request queue is empty\n");
    }

    //Retrieve request from request queue & Create a Thread
    sem_wait(sem[N]); //hold request queue
```

Figure 2.1 Before change

```
while (1) {
    fflush(stdout);
    //Wait for a request to arrive
    int has_req = 0;
    while (!has_req) { // CHANGE 2
        sem_wait(sem[N]); //hold request queue
        if (sdp->requestIn == sdp->requestOut) sem_post(sem[N]);
        else has_req = 1;
        //printf("request queue is empty\n");
    }

    //Retrieve request from request queue & Create a Thread
```

Figure 2.2 After change

**CHANGE 3**

We used to allocate parameter to server thread once in the beginning of server file. Even though we allocated it in heap, this caused all the threads to use the same parameter file, and therefore made us unable to run concurrently without problems. Therefore, we moved the parameter allocation from the beginning of file to the point where we create the thread. We moved

“ `par = (struct param *) malloc(sizeof(struct param));`
`par->req = (struct request *) malloc(sizeof(struct request));`
`par->sdp = (struct shared *) malloc(sizeof(struct shared));` ” from line 28 and

“ `strcpy(par->file_name, argv[2]);` ” from line 46 into *line 119* in **server.c** .

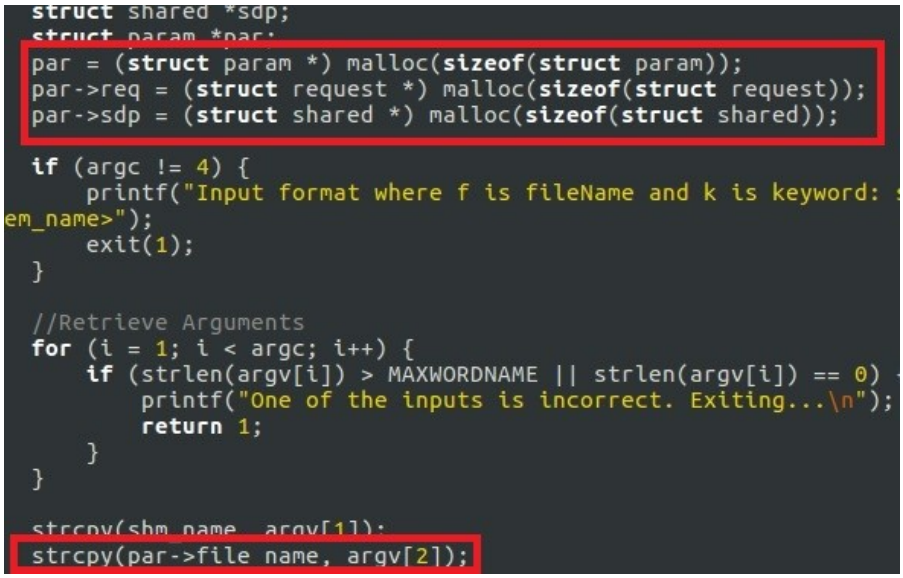


Figure 3.1 Before change

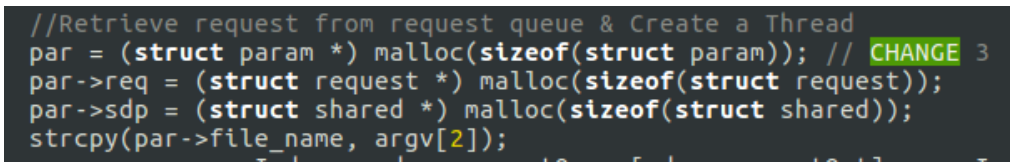


Figure 3.2 After change

**CHANGE 4**

We used to not check if result queue was full, to add -1 to result queue between server thread and client. We have added a while loop to *line 209* in **server.c.**

```
//Add a -1 to the end
sem_wait(p->sem_resultq); //hold result queue of qindex
p->sdp->resultQueues[qindex][p->sdp->inout[qindex][IN]] = -1;
p->sdp->inout[qindex][IN] = (p->sdp->inout[qindex][IN] + 1) % BUFSIZE;
sem_post(p->sem_resultq); //release result queue of qindex
```

Figure 2.1 Before change

```
//Add a -1 to the end
while ((p->sdp->inout[qindex][IN] + 1) % BUFSIZE == p->sdp->inout[qindex][OUT]) {} // CHANGE 4
sem_wait(p->sem_resultq); //hold result queue of qindex
p->sdp->resultQueues[qindex][p->sdp->inout[qindex][IN]] = -1;
p->sdp->inout[qindex][IN] = (p->sdp->inout[qindex][IN] + 1) % BUFSIZE;
sem_post(p->sem_resultq); //release result queue of qindex
```

Figure 4.2 After change

**CHANGE 5**

Apparently, we have accidentally deleted the line where we used to check if result queue was empty. We have added the line back to *line 129* in **client.c.**

```
while (1) {
    /*int *t;
    while(*t <= 2) {
        sem_post(sem[i]);
        sem_getvalue(sem[i], t);
    }*/
    sem_wait(sem[i]); //hold result queue i's semaphore
```

Figure 5.1 Before change

```
while (1) {
    /*int *t;
    while(*t <= 2) {
        sem_post(sem[i]);
        sem_getvalue(sem[i], t);
    }*/
    while((sdp->inout[i][IN]) % BUFSIZE == sdp->inout[i][OUT]) {} // CHANGE 5

    sem_wait(sem[i]); //hold result queue i's semaphore
    if(sdp->resultQueues[i][sdp->inout[i][OUT]] == -1) {
```

Figure 5.2 After change

**CHANGE 6**

As in change 5, we have readded a line that was accidentally deleted and went unnoticed.
We used to increment out variable of the particular result queue even when we read -1, and we prepared logic of the program according to that. When we deleted the incrementing statement before the break statement, this caused the according problematic case:
Assume a client used result queue 1, saw -1, didn't increment 'out' variable. Then it rewrote result queue's data to 0 and exited.
Assume another client started using result queue 1, this time this client will first read the data at the point where the previous client left, which means new client will print "0" which will be wrong.

We have readded the incrementing statement at *line 134* in **client.c.**

```
if(sdp->resultQueues[i][j] == -1) {
    sem_post(sem[i]);
    break;
}
```

Figure 6.1 Before change

```
sem_wait(sem[i]); //hold result queue i's semaphore
if(sdp->resultQueues[i][sdp->inout[i][OUT]] == -1) {
    sem_post(sem[i]);
    sdp->inout[i][OUT] = (sdp->inout[i][OUT] + 1) % BUFSIZE; // CHANGE 6
    break;
}
```

Figure 6.2 After change