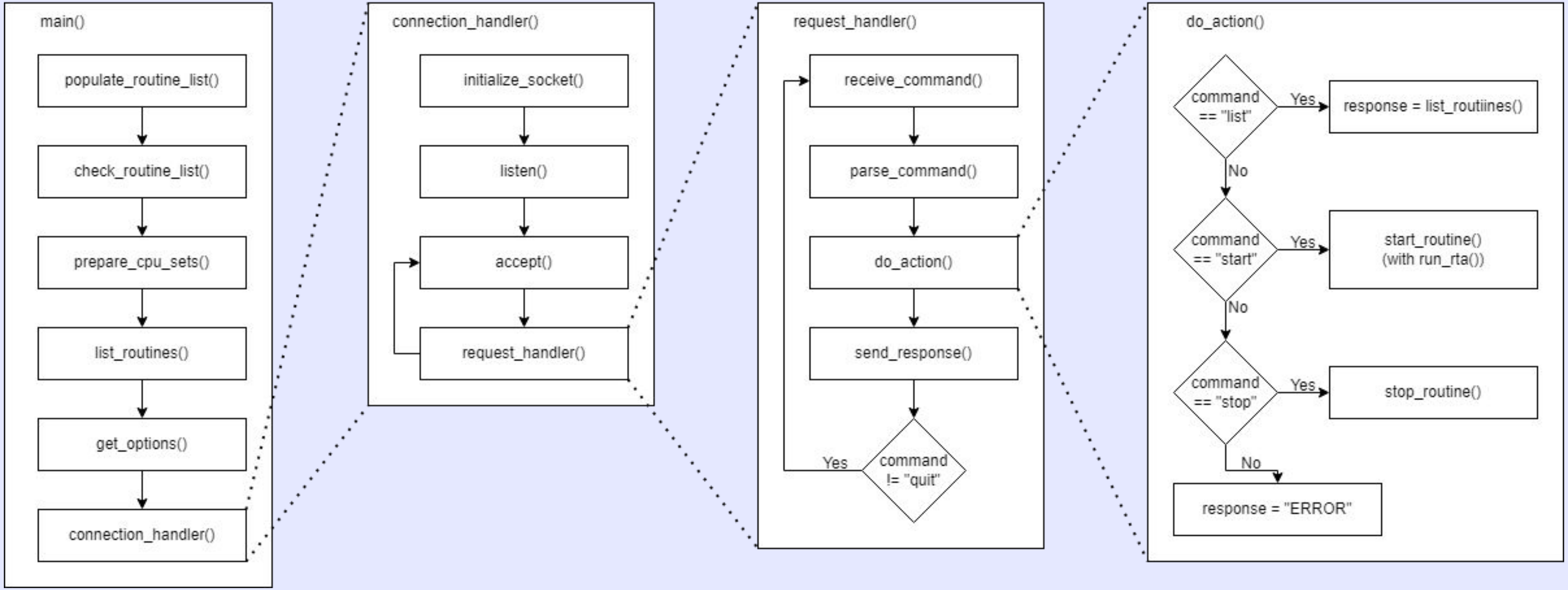
Concurrent and Real-Time Programming

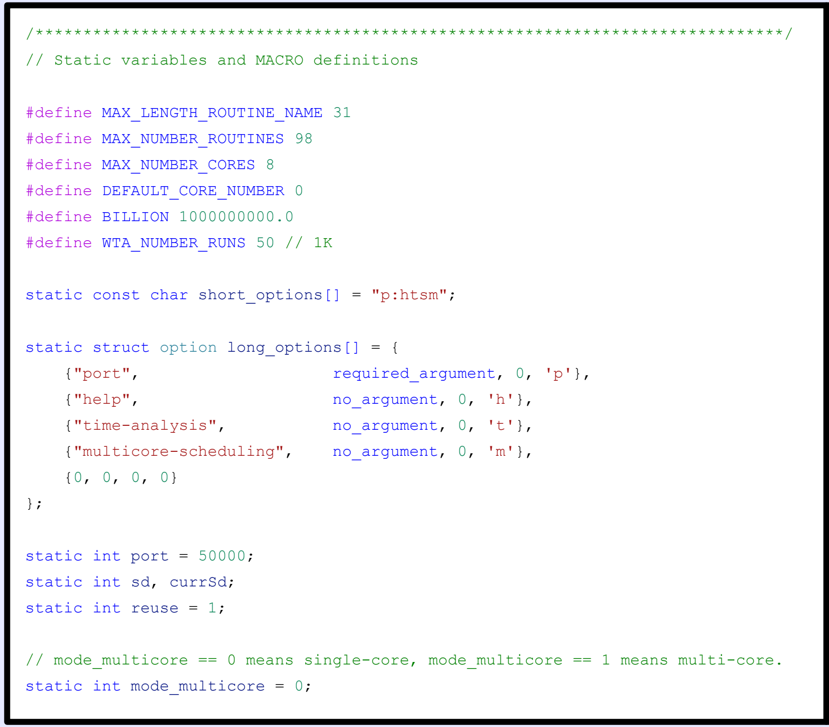
Exercise N°5

Lorenzo Cappellotto
2044728

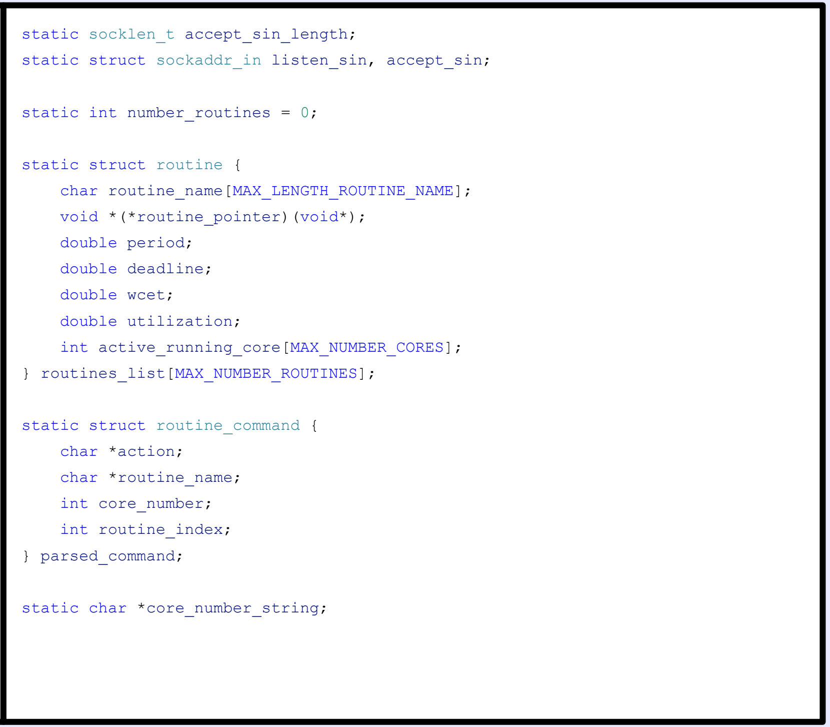
General Structure (Supervisor)



Used Data Structures

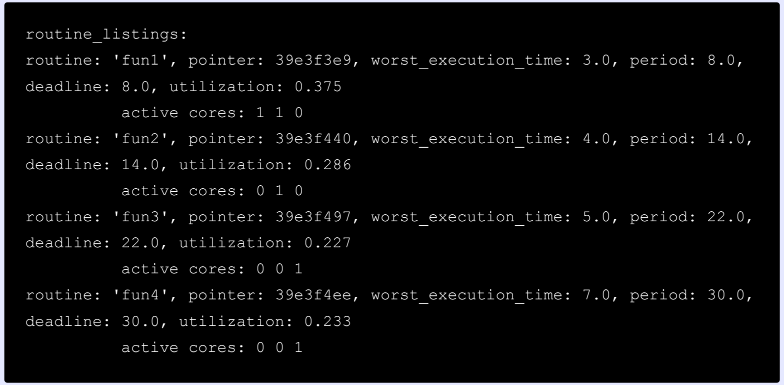


```c
/******************************************************************************/
// Static variables and MACRO definitions

#define MAX_LENGTH_ROUTINE_NAME 31
#define MAX_NUMBER_ROUTINES 98
#define MAX_NUMBER_CORES 8
#define DEFAULT_CORE_NUMBER 0
#define BILLION 1000000000.0
#define WTA_NUMBER_RUNS 50 // 1K

static const char short_options[] = "p:htsm";

static struct option long_options[] = {
    {"port",                    required_argument, 0, 'p'},
    {"help",                    no_argument, 0, 'h'},
    {"time-analysis",           no_argument, 0, 't'},
    {"multicore-scheduling",    no_argument, 0, 'm'},
    {0, 0, 0, 0}
};

static int port = 50000;
static int sd, currSd;
static int reuse = 1;

// mode_multicore == 0 means single-core, mode_multicore == 1 means multi-core.
static int mode_multicore = 0;
```

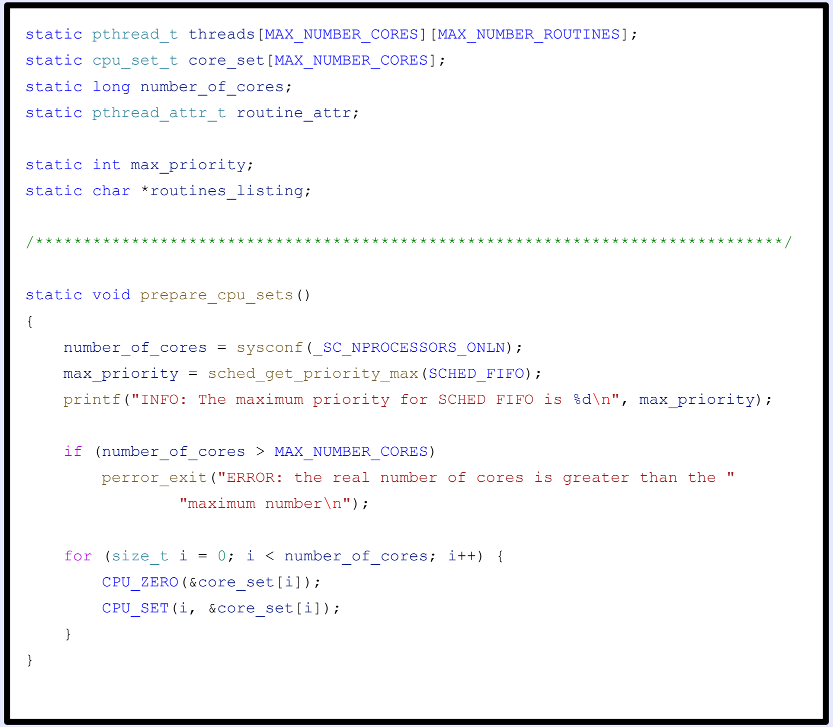


```c
static socklen_t accept_sin_length;
static struct sockaddr_in listen_sin, accept_sin;

static int number_routines = 0;

static struct routine {
    char routine_name[MAX_LENGTH_ROUTINE_NAME];
    void *(*routine_pointer)(void*);
    double period;
    double deadline;
    double wcet;
    double utilization;
    int active_running_core[MAX_NUMBER_CORES];
} routines_list[MAX_NUMBER_ROUTINES];

static struct routine_command {
    char *action;
    char *routine_name;
    int core_number;
    int routine_index;
} parsed_command;

static char *core_number_string;
```

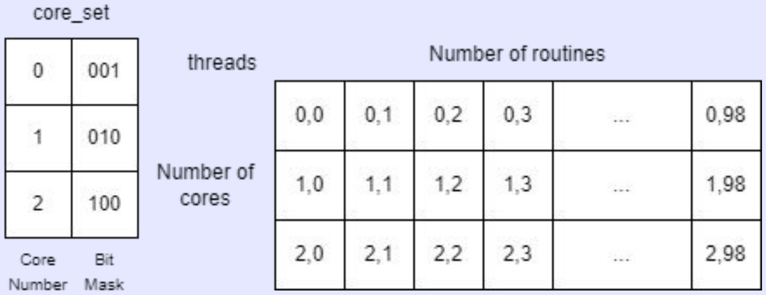
Used Data Structures and CPU_SET Preparation



```
routine_listings:
routine: 'fun1', pointer: 39e3f3e9, worst_execution_time: 3.0, period: 8.0,
deadline: 8.0, utilization: 0.375
          active cores: 1 1 0
routine: 'fun2', pointer: 39e3f440, worst_execution_time: 4.0, period: 14.0,
deadline: 14.0, utilization: 0.286
          active cores: 0 1 0
routine: 'fun3', pointer: 39e3f497, worst_execution_time: 5.0, period: 22.0,
deadline: 22.0, utilization: 0.227
          active cores: 0 0 1
routine: 'fun4', pointer: 39e3f4ee, worst_execution_time: 7.0, period: 30.0,
deadline: 30.0, utilization: 0.233
          active cores: 0 0 1
```



```c
static pthread_t threads[MAX_NUMBER_CORES][MAX_NUMBER_ROUTINES];
static cpu_set_t core_set[MAX_NUMBER_CORES];
static long number_of_cores;
static pthread_attr_t routine_attr;

static int max_priority;
static char *routines_listing;

/******************************************************************************/

static void prepare_cpu_sets()
{
    number_of_cores = sysconf(_SC_NPROCESSORS_ONLN);
    max_priority = sched_get_priority_max(SCHED_FIFO);
    printf("INFO: The maximum priority for SCHED FIFO is %d\n", max_priority);

    if (number_of_cores > MAX_NUMBER_CORES)
        perror_exit("ERROR: the real number of cores is greater than the "
                "maximum number\n");

    for (size_t i = 0; i < number_of_cores; i++) {
        CPU_ZERO(&core_set[i]);
        CPU_SET(i, &core_set[i]);
    }
}
```



core_set

Core Number	Bit Mask
0	001
1	010
2	100

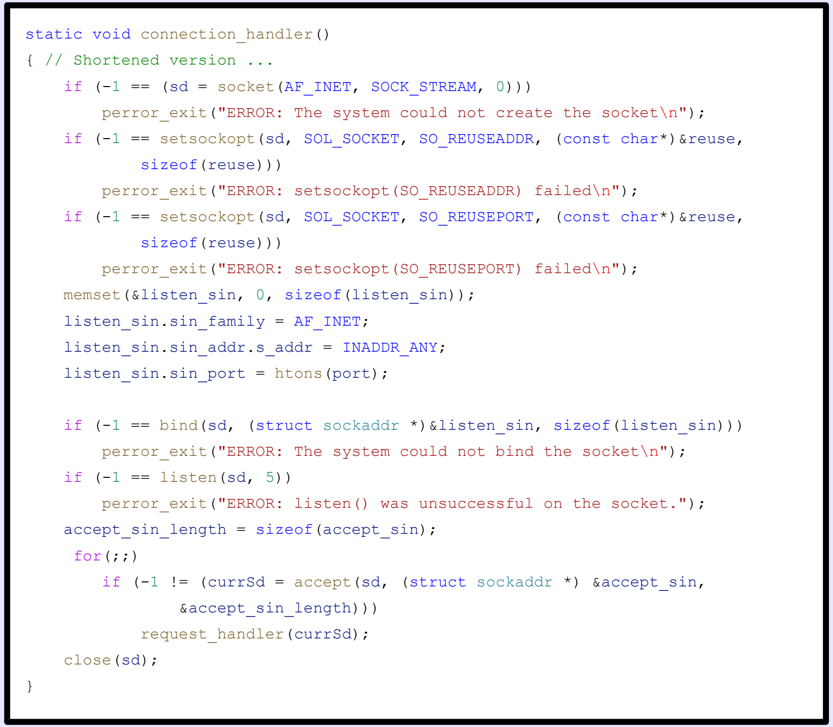
threads

Number of routines

Number of cores

0,0	0,1	0,2	0,3	...	0,98
1,0	1,1	1,2	1,3	...	1,98
2,0	2,1	2,2	2,3	...	2,98

Client/Server Model



```
static void connection_handler()
{ // Shortened version ...
  if (-1 == (sd = socket(AF_INET, SOCK_STREAM, 0)))
    perror_exit("ERROR: The system could not create the socket\n");
  if (-1 == setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (const char*)&reuse,
      sizeof(reuse)))
    perror_exit("ERROR: setsockopt(SO_REUSEADDR) failed\n");
  if (-1 == setsockopt(sd, SOL_SOCKET, SO_REUSEPORT, (const char*)&reuse,
      sizeof(reuse)))
    perror_exit("ERROR: setsockopt(SO_REUSEPORT) failed\n");
  memset(&listen_sin, 0, sizeof(listen_sin));
  listen_sin.sin_family = AF_INET;
  listen_sin.sin_addr.s_addr = INADDR_ANY;
  listen_sin.sin_port = htons(port);

  if (-1 == bind(sd, (struct sockaddr *)&listen_sin, sizeof(listen_sin)))
    perror_exit("ERROR: The system could not bind the socket\n");
  if (-1 == listen(sd, 5))
    perror_exit("ERROR: listen() was unsuccessful on the socket.");
  accept_sin_length = sizeof(accept_sin);
   for(;;)
    if (-1 != (currSd = accept(sd, (struct sockaddr *) &accept_sin,
        &accept_sin_length)))
      request_handler(currSd);
  close(sd);
}
```

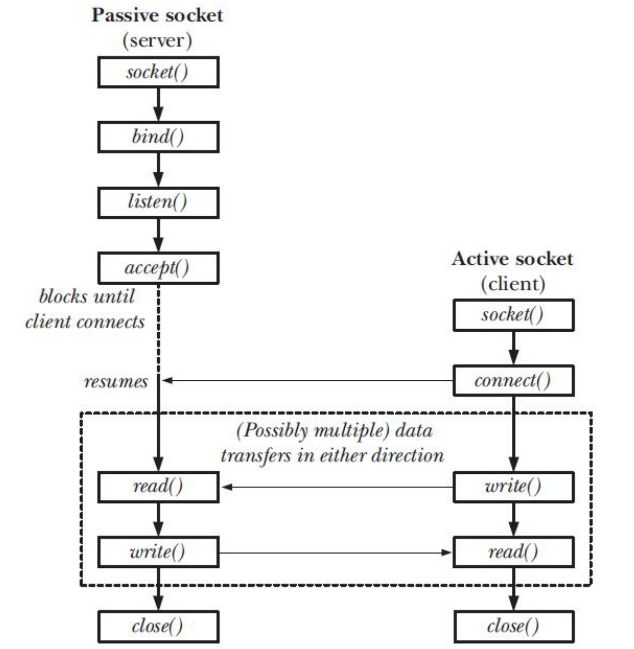
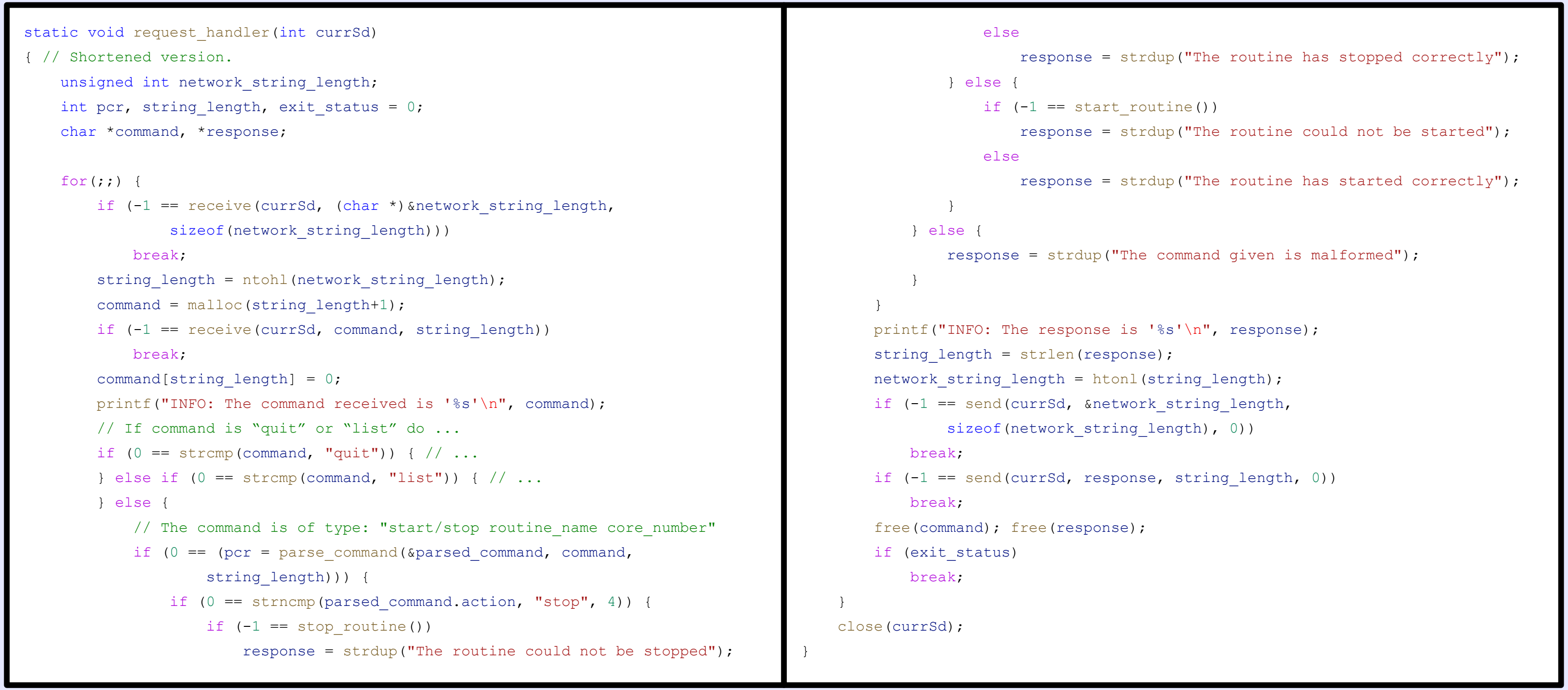


Figure 56-1: Overview of system calls used with stream sockets

Client/Server Model



```c
static void request_handler(int currSd)
{ // Shortened version.
    unsigned int network_string_length;
    int pcr, string_length, exit_status = 0;
    char *command, *response;

    for(;;) {
        if (-1 == receive(currSd, (char *)&network_string_length,
                sizeof(network_string_length)))
            break;
        string_length = ntohl(network_string_length);
        command = malloc(string_length+1);
        if (-1 == receive(currSd, command, string_length))
            break;
        command[string_length] = 0;
        printf("INFO: The command received is '%s'\n", command);
        // If command is “quit” or “list” do ...
        if (0 == strcmp(command, "quit")) { // ...
        } else if (0 == strcmp(command, "list")) { // ...
        } else {
            // The command is of type: "start/stop routine_name core_number"
            if (0 == (pcr = parse_command(&parsed_command, command,
                    string_length))) {
                if (0 == strncmp(parsed_command.action, "stop", 4)) {
                    if (-1 == stop_routine())
                        response = strdup("The routine could not be stopped");
                    else
                        response = strdup("The routine has stopped correctly");
                } else {
                    if (-1 == start_routine())
                        response = strdup("The routine could not be started");
                    else
                        response = strdup("The routine has started correctly");
                }
            } else {
                response = strdup("The command given is malformed");
            }
        }
        printf("INFO: The response is '%s'\n", response);
        string_length = strlen(response);
        network_string_length = htonl(string_length);
        if (-1 == send(currSd, &network_string_length,
                sizeof(network_string_length), 0))
            break;
        if (-1 == send(currSd, response, string_length, 0))
            break;
        free(command); free(response);
        if (exit_status)
            break;
    }
    close(currSd);
}
```

Client/Server Model

```
int main (int argc, char **argv)
{ // Shortened version ...
    hp = gethostbyname(hostname);
    if (0 == hp)
        perror_exit("ERROR: The system could not translate the hostname");

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = ((struct in_addr*)(hp->h_addr_list[0]))->s_addr;
    sin.sin_port = htons(port);

    if (-1 == (sd = socket(AF_INET, SOCK_STREAM, 0)))
        perror_exit("ERROR: The system could not create the socket\n");
    if (-1 == connect(sd, (struct sockaddr*)&sin, sizeof(sin)))
        perror_exit("ERROR: Impossible to connect to the server's socket");

    for(;;) {
        printf("Enter command (start/stop <task_name>, list, help , quit): ");
        fgets(command, MAX_COMMAND_SIZE, stdin);
        string_length = strlen(command);
        if ((string_length > 0) && (command[string_length-1] == '\n'))
            command[string_length-1] = '\0';

        if(0 == strcmp(command, "help")){
            usage(stdout); continue;
        }
```

```
        network_string_length = htonl(string_length);
        if (-1 == send(sd, &network_string_length,
                sizeof(network_string_length), 0))
            perror_exit("ERROR: Impossible to send the length");
        if (-1 == send(sd, command, string_length, 0))
            perror_exit("ERROR: Impossible to send the command");

        if (-1 == receive(sd, (char *)&network_string_length,
                sizeof(network_string_length)))
            perror_exit("ERROR: Impossible to receive the length");
        string_length = ntohl(network_string_length);
        response = malloc(string_length + 1);
        if (-1 == receive(sd, response, string_length))
            perror_exit("ERROR: Impossible to rece
        response[string_length] = 0;
        printf("INFO: The response is '%s'\n", res
        free(response);

        if(0 == strcmp(command, "quit"))
            break;
    }

    close(sd);
    printf("INFO: Connection Terminated\n");
    return 0;
}
```

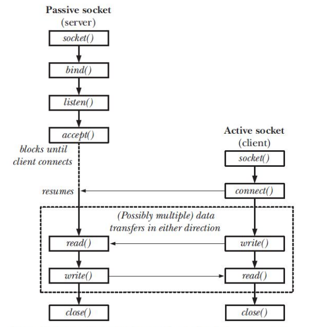
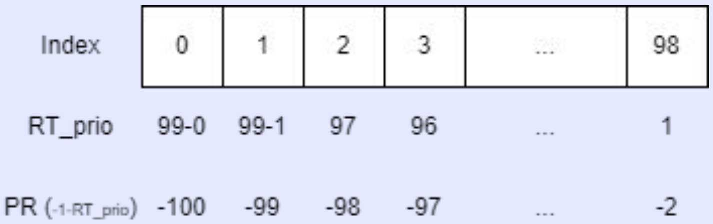
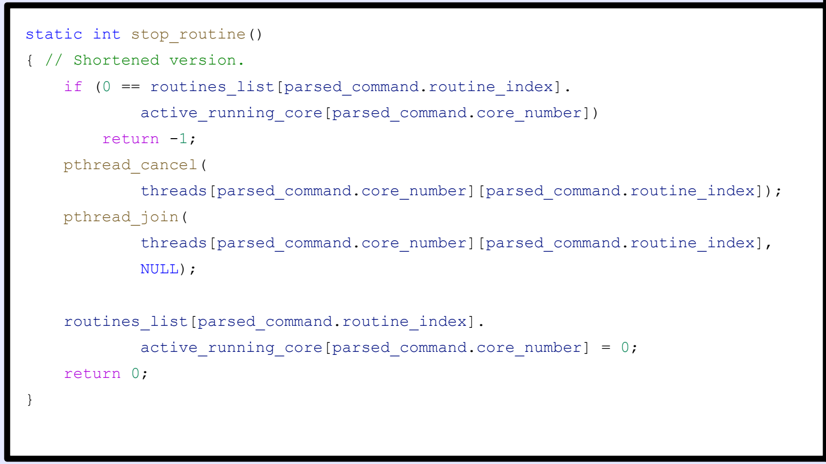


Figure 56-1: Overview of system calls used with stream sockets

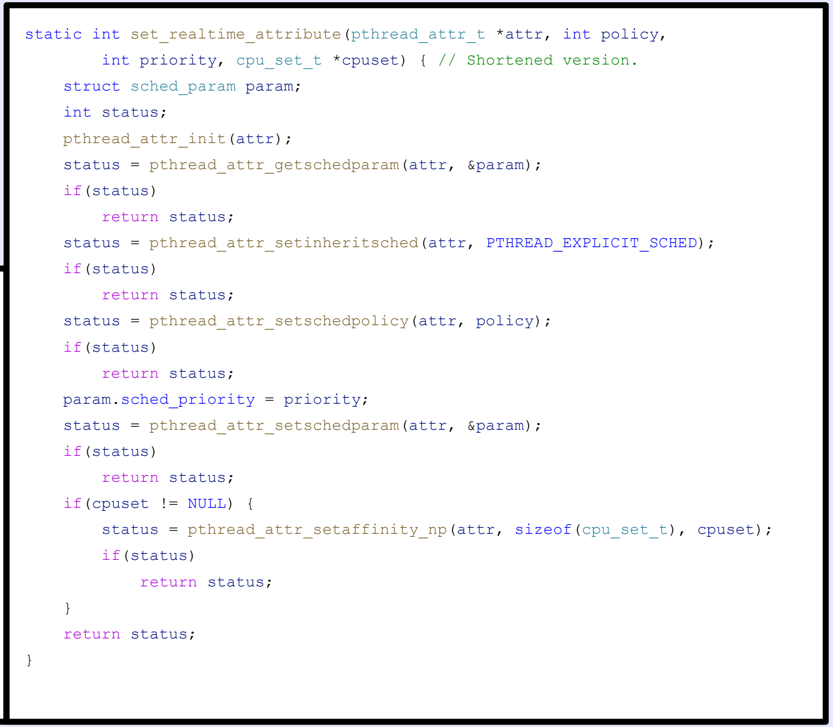
Deadline Monotonic Scheduling



Index	0	1	2	3	...	98
RT_prio	99-0	99-1	97	96	...	1
PR (-1-RT_prio)	-100	-99	-98	-97	...	-2



```
static int stop_routine()
{ // Shortened version.
    if (0 == routines_list[parsed_command.routine_index].
            active_running_core[parsed_command.core_number])
        return -1;
    pthread_cancel(
            threads[parsed_command.core_number][parsed_command.routine_index]);
    pthread_join(
            threads[parsed_command.core_number][parsed_command.routine_index],
            NULL);

    routines_list[parsed_command.routine_index].
            active_running_core[parsed_command.core_number] = 0;
    return 0;
}
```

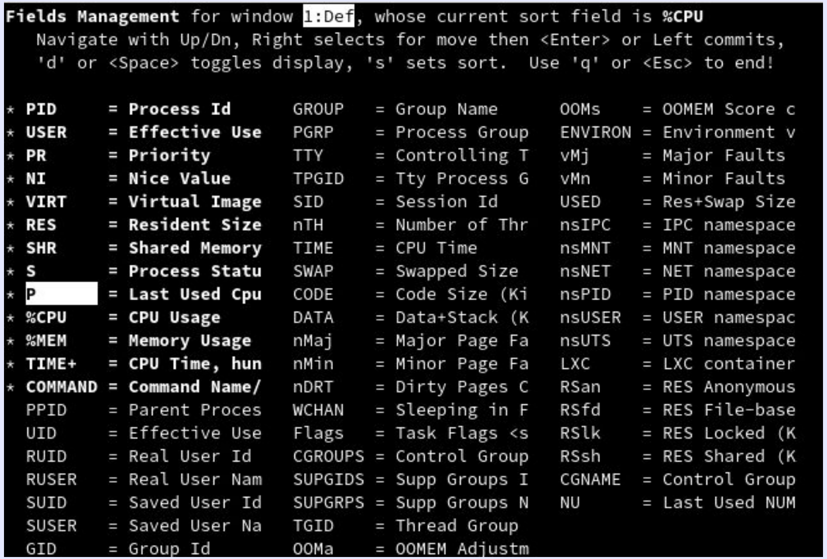


```
static int set_realtime_attribute(pthread_attr_t *attr, int policy,
        int priority, cpu_set_t *cpuset) { // Shortened version.
    struct sched_param param;
    int status;
    pthread_attr_init(attr);
    status = pthread_attr_getschedparam(attr, &param);
    if(status)
        return status;
    status = pthread_attr_setinheritsched(attr, PTHREAD_EXPLICIT_SCHED);
    if(status)
        return status;
    status = pthread_attr_setschedpolicy(attr, policy);
    if(status)
        return status;
    param.sched_priority = priority;
    status = pthread_attr_setschedparam(attr, &param);
    if(status)
        return status;
    if(cpuset != NULL) {
        status = pthread_attr_setaffinity_np(attr, sizeof(cpu_set_t), cpuset);
        if(status)
            return status;
    }
    return status;
}
```

Deadline Monotonic Scheduling

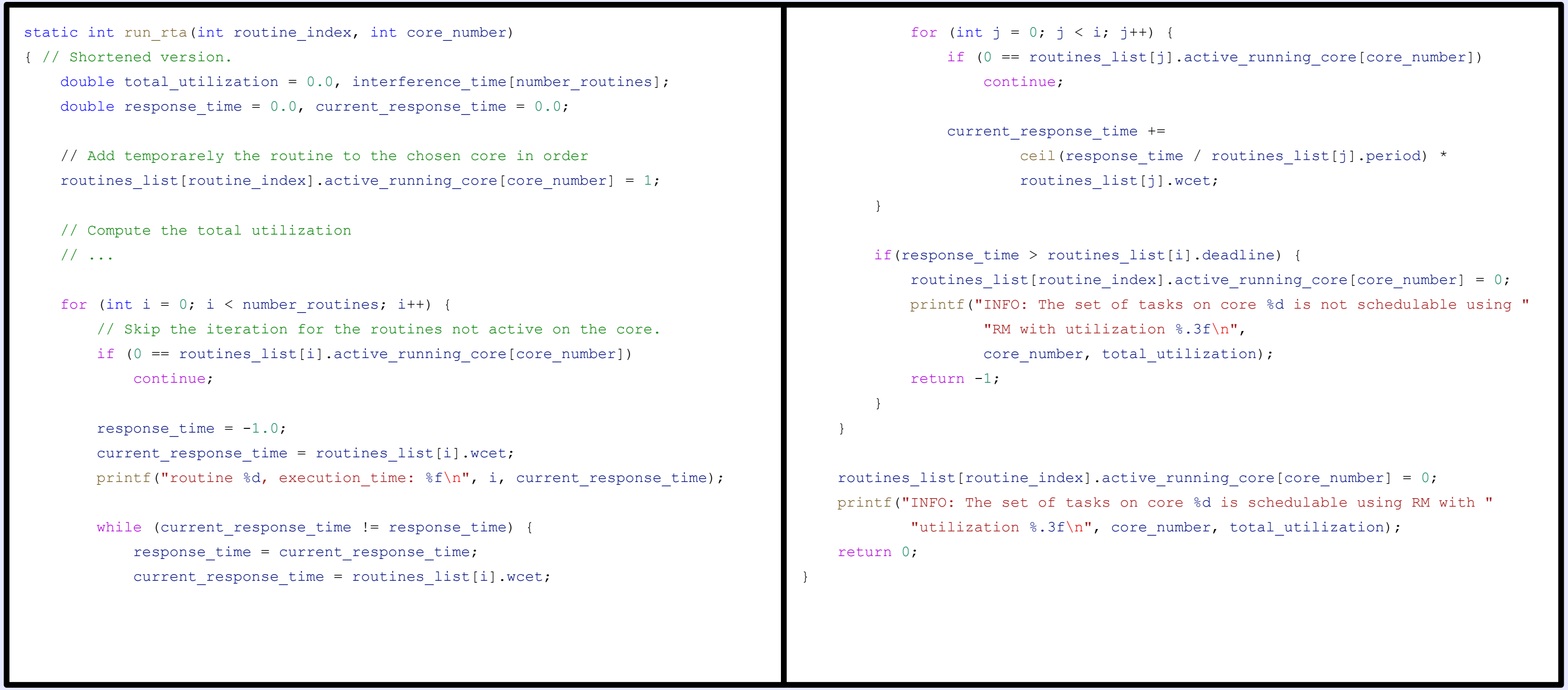
```
static int start_routine()
{ // Shortened version.
    int schedulable;
    if (1 == routines_list[parsed_command.routine_index].
            active_running_core[parsed_command.core_number])
        return -1;
    if (-1 == (schedulable =
            run_rta(parsed_command.routine_index, parsed_command.core_number)))
        return -1;

    pthread_attr_t routine_attr;
    set_realtime_attribute(
            &routine_attr,
            SCHED_FIFO,
            max_priority - parsed_command.routine_index,
            &core_set[parsed_command.core_number]);
    pthread_create(
            &threads[parsed_command.core_number][parsed_command.routine_index],
            &routine_attr,
            routine_wrapper,
            &routines_list[parsed_command.routine_index]);

    routines_list[parsed_command.routine_index].
            active_running_core[parsed_command.core_number] = 1;
    return 0;
}
```

```
$ top -H -p $(ps -a | grep 'supervisor' | awk -F' '  '{print $1}')
```



```
Fields Management for window 1:Def, whose current sort field is %CPU
   Navigate with Up/Dn, Right selects for move then <Enter> or Left commits,
   'd' or <Space> toggles display, 's' sets sort.  Use 'q' or <Esc> to end!

* PID     = Process Id      GROUP   = Group Name      OOMs    = OOMEM Score c
* USER    = Effective Use   PGRP    = Process Group   ENVIRON = Environment v
* PR      = Priority        TTY     = Controlling T   vMj     = Major Faults
* NI      = Nice Value      TPGID   = Tty Process G   vMn     = Minor Faults
* VIRT    = Virtual Image   SID     = Session Id      USED    = Res+Swap Size
* RES     = Resident Size   nTH     = Number of Thr   nsIPC   = IPC namespace
* SHR     = Shared Memory   TIME    = CPU Time        nsMNT   = MNT namespace
* S       = Process Statu   SWAP    = Swapped Size    nsNET   = NET namespace
* P       = Last Used Cpu   CODE    = Code Size (Ki   nsPID   = PID namespace
* %CPU    = CPU Usage       DATA    = Data+Stack (K   nsUSER  = USER namespac
* %MEM    = Memory Usage    nMaj    = Major Page Fa   nsUTS   = UTS namespace
* TIME+   = CPU Time, hun   nMin    = Minor Page Fa   LXC     = LXC container
* COMMAND = Command Name/   nDRT    = Dirty Pages C   RSan    = RES Anonymous
  PPID    = Parent Proces   WCHAN   = Sleeping in F   RSfd    = RES File-base
  UID     = Effective Use   Flags   = Task Flags <s   RSlk    = RES Locked (K
  RUID    = Real User Id    CGROUPS = Control Group   RSsh    = RES Shared (K
  RUSER   = Real User Nam   SUPGIDS = Supp Groups I   CGNAME  = Control Group
  SUID    = Saved User Id   SUPGRPS = Supp Groups N   NU      = Last Used NUM
  SUSER   = Saved User Na   TGID    = Thread Group
  GID     = Group Id        OOMa    = OOMEM Adjustm
```

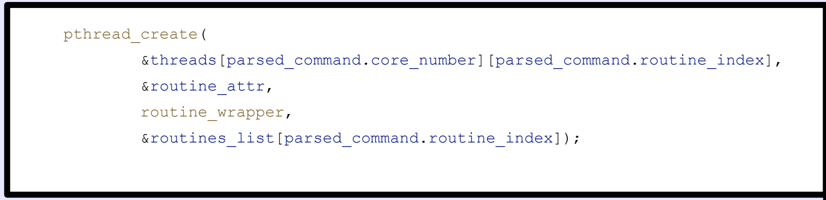
Response Time Analysis



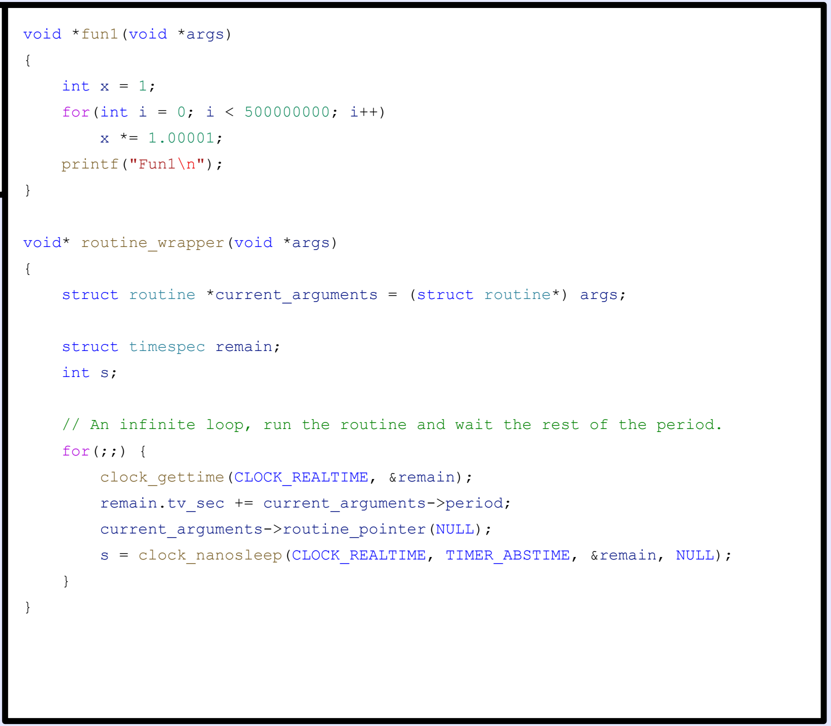
```
static int run_rta(int routine_index, int core_number)
{ // Shortened version.
    double total_utilization = 0.0, interference_time[number_routines];
    double response_time = 0.0, current_response_time = 0.0;

    // Add temporarely the routine to the chosen core in order
    routines_list[routine_index].active_running_core[core_number] = 1;

    // Compute the total utilization
    // ...

    for (int i = 0; i < number_routines; i++) {
        // Skip the iteration for the routines not active on the core.
        if (0 == routines_list[i].active_running_core[core_number])
            continue;

        response_time = -1.0;
        current_response_time = routines_list[i].wcet;
        printf("routine %d, execution_time: %f\n", i, current_response_time);

        while (current_response_time != response_time) {
            response_time = current_response_time;
            current_response_time = routines_list[i].wcet;
```

```
            for (int j = 0; j < i; j++) {
                if (0 == routines_list[j].active_running_core[core_number])
                    continue;

                current_response_time +=
                        ceil(response_time / routines_list[j].period) *
                        routines_list[j].wcet;
        }

        if(response_time > routines_list[i].deadline) {
            routines_list[routine_index].active_running_core[core_number] = 0;
            printf("INFO: The set of tasks on core %d is not schedulable using "
                    "RM with utilization %.3f\n",
                    core_number, total_utilization);
            return -1;
        }
    }

    routines_list[routine_index].active_running_core[core_number] = 0;
    printf("INFO: The set of tasks on core %d is schedulable using RM with "
            "utilization %.3f\n", core_number, total_utilization);
    return 0;
}
```

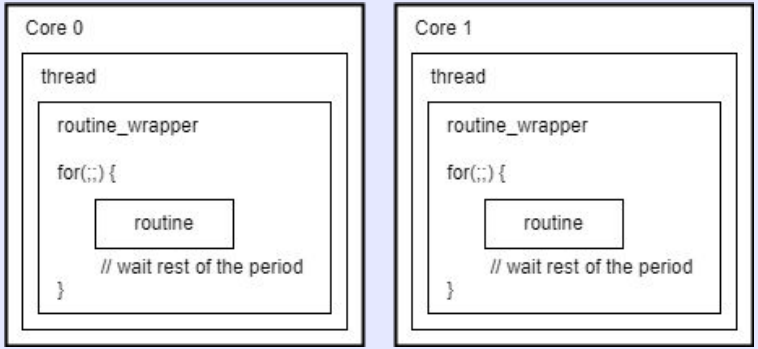
Routine Wrapping and Period Waiting



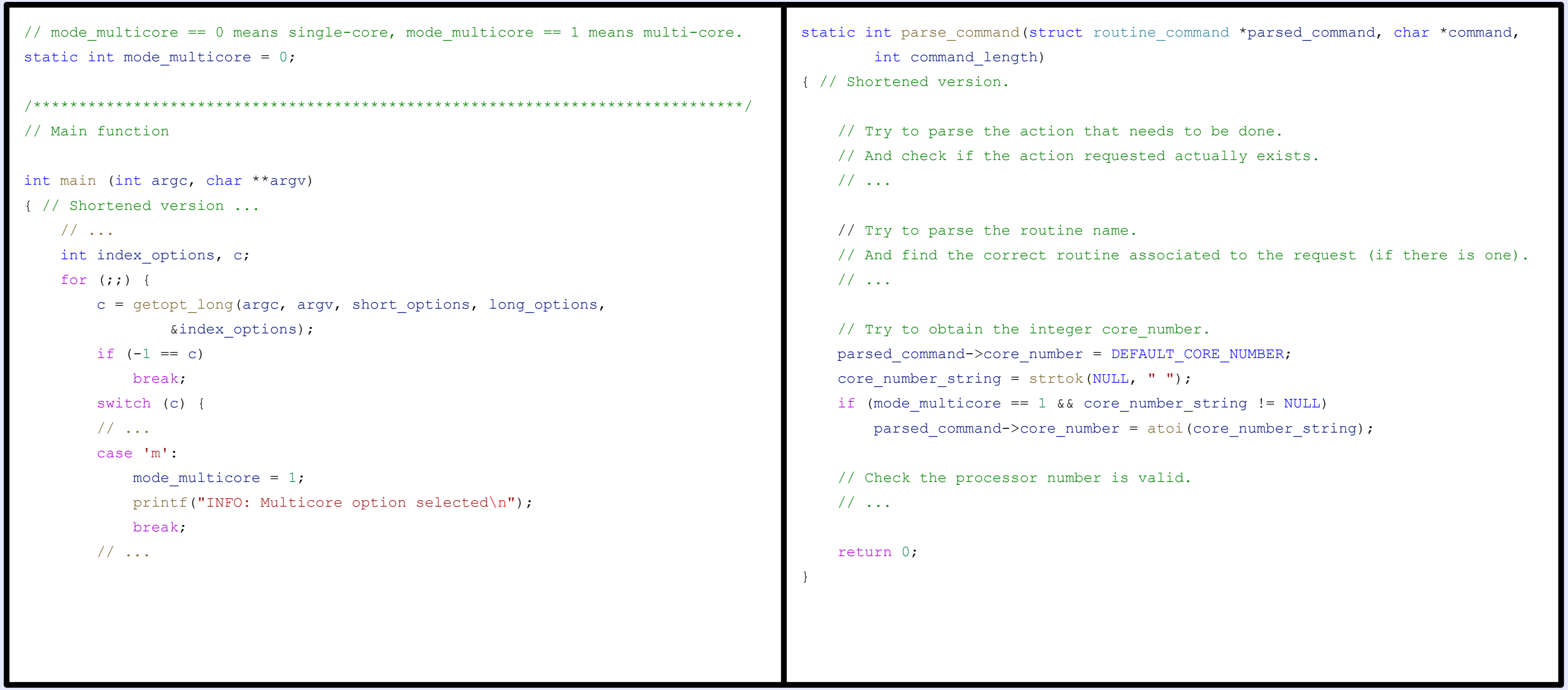
```
pthread_create(
        &threads[parsed_command.core_number][parsed_command.routine_index],
        &routine_attr,
        routine_wrapper,
        &routines_list[parsed_command.routine_index]);
```



```
void *fun1(void *args)
{
    int x = 1;
    for(int i = 0; i < 500000000; i++)
        x *= 1.00001;
    printf("Fun1\n");
}

void* routine_wrapper(void *args)
{
    struct routine *current_arguments = (struct routine*) args;

    struct timespec remain;
    int s;

    // An infinite loop, run the routine and wait the rest of the period.
    for(;;) {
        clock_gettime(CLOCK_REALTIME, &remain);
        remain.tv_sec += current_arguments->period;
        current_arguments->routine_pointer(NULL);
        s = clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &remain, NULL);
    }
}
```



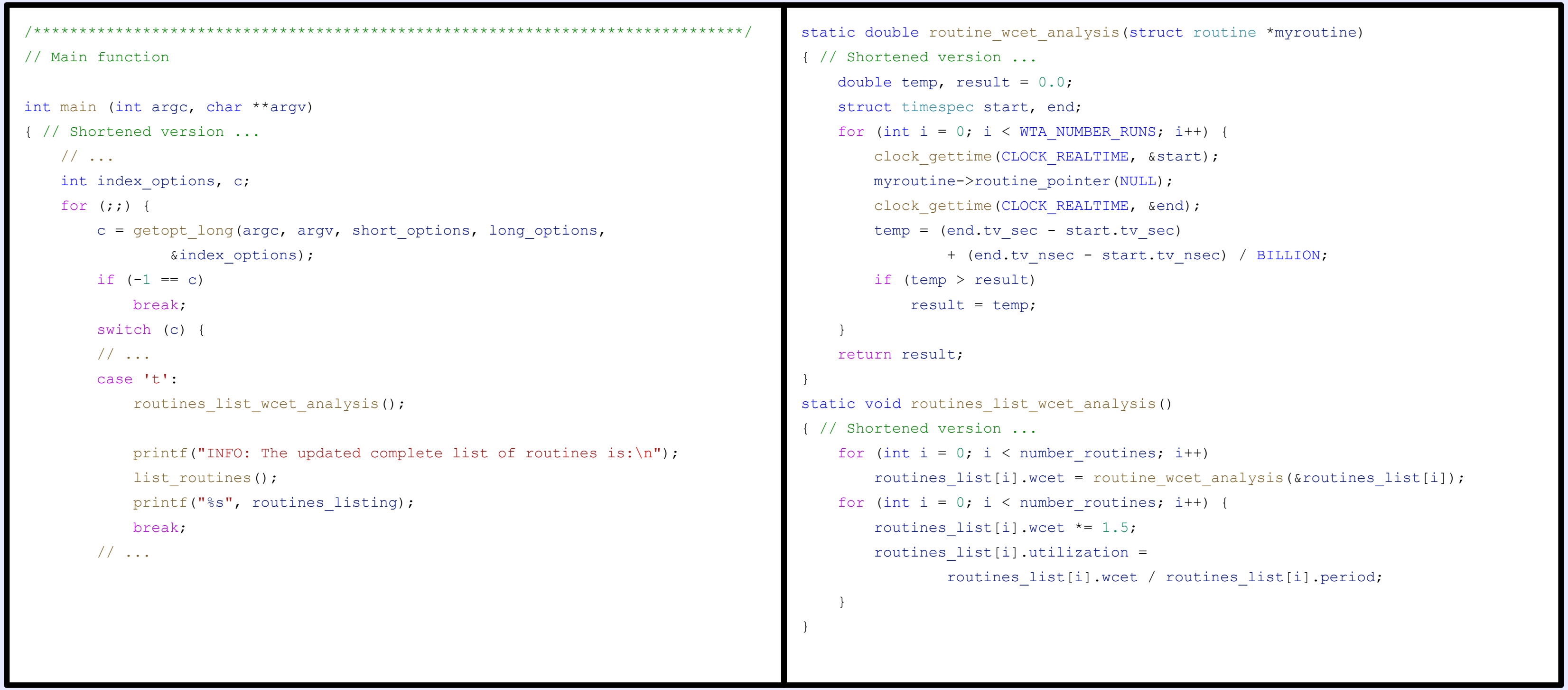
Multicore Execution



```c
// mode_multicore == 0 means single-core, mode_multicore == 1 means multi-core.
static int mode_multicore = 0;

/*******************************************************************************/
// Main function

int main (int argc, char **argv)
{ // Shortened version ...
    // ...
    int index_options, c;
    for (;;) {
        c = getopt_long(argc, argv, short_options, long_options,
                &index_options);
        if (-1 == c)
            break;
        switch (c) {
        // ...
        case 'm':
            mode_multicore = 1;
            printf("INFO: Multicore option selected\n");
            break;
        // ...
```

```c
static int parse_command(struct routine_command *parsed_command, char *command,
        int command_length)
{ // Shortened version.

    // Try to parse the action that needs to be done.
    // And check if the action requested actually exists.
    // ...

    // Try to parse the routine name.
    // And find the correct routine associated to the request (if there is one).
    // ...

    // Try to obtain the integer core_number.
    parsed_command->core_number = DEFAULT_CORE_NUMBER;
    core_number_string = strtok(NULL, " ");
    if (mode_multicore == 1 && core_number_string != NULL)
        parsed_command->core_number = atoi(core_number_string);

    // Check the processor number is valid.
    // ...

    return 0;
}
```

Estimating Routines’ Worst Case Execution Time



```c
/*******************************************************************************/
// Main function

int main (int argc, char **argv)
{ // Shortened version ...
    // ...
    int index_options, c;
    for (;;) {
        c = getopt_long(argc, argv, short_options, long_options,
                &index_options);
        if (-1 == c)
            break;
        switch (c) {
        // ...
        case 't':
            routines_list_wcet_analysis();

            printf("INFO: The updated complete list of routines is:\n");
            list_routines();
            printf("%s", routines_listing);
            break;
        // ...
```

```c
static double routine_wcet_analysis(struct routine *myroutine)
{ // Shortened version ...
    double temp, result = 0.0;
    struct timespec start, end;
    for (int i = 0; i < WTA_NUMBER_RUNS; i++) {
        clock_gettime(CLOCK_REALTIME, &start);
        myroutine->routine_pointer(NULL);
        clock_gettime(CLOCK_REALTIME, &end);
        temp = (end.tv_sec - start.tv_sec)
                + (end.tv_nsec - start.tv_nsec) / BILLION;
        if (temp > result)
            result = temp;
    }
    return result;
}
static void routines_list_wcet_analysis()
{ // Shortened version ...
    for (int i = 0; i < number_routines; i++)
        routines_list[i].wcet = routine_wcet_analysis(&routines_list[i]);
    for (int i = 0; i < number_routines; i++) {
        routines_list[i].wcet *= 1.5;
        routines_list[i].utilization =
                routines_list[i].wcet / routines_list[i].period;
    }
}
```

The end

Thank you for your attention!