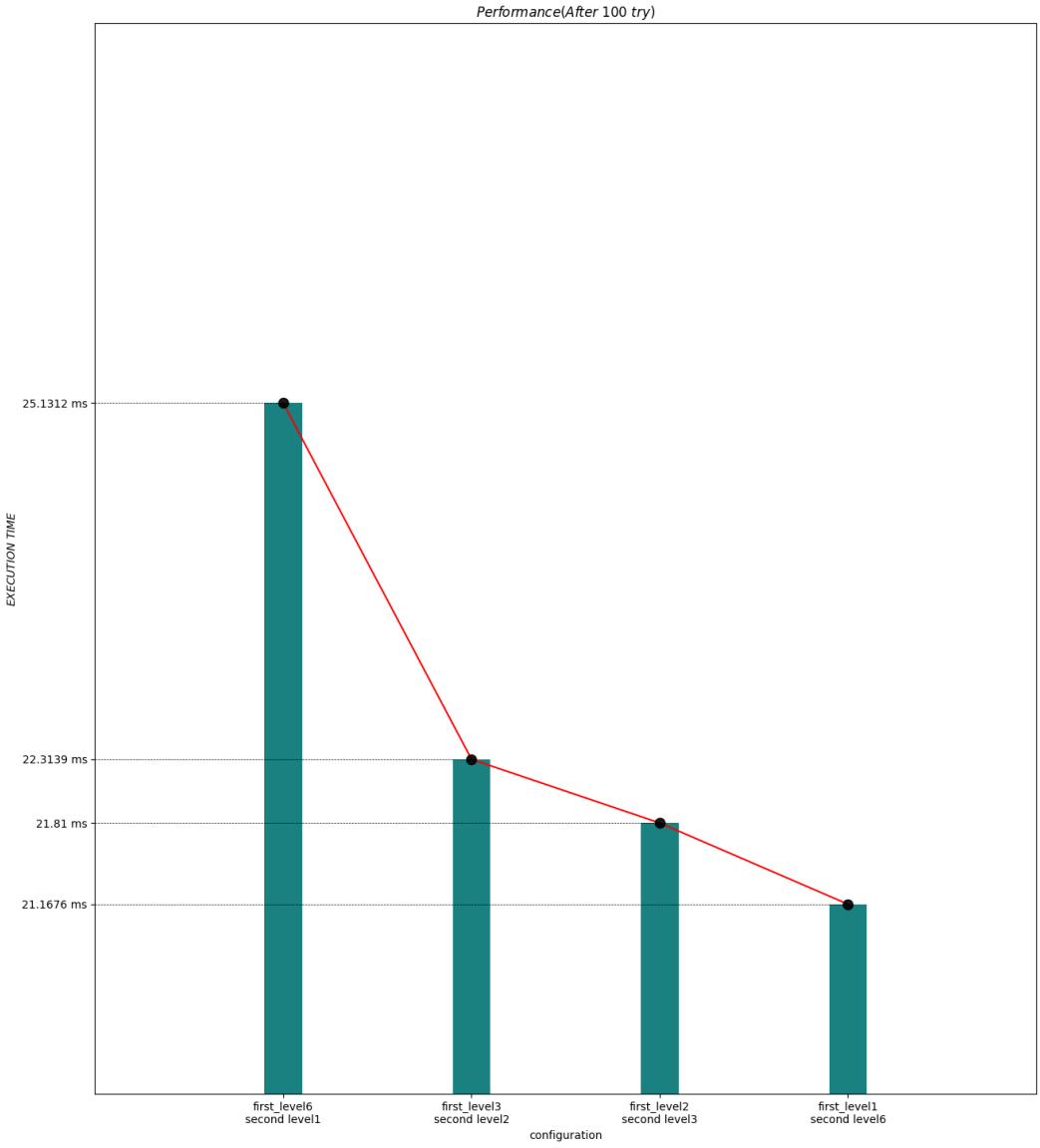
MohammadAmin Shafiee
(9727713)

در شکل زیر زمان اندازگیری هر configure مختلف امده است.
برای 6 دستور 4 حالت

1 6
2 3
3 2
6 1

را داریم. که در هر حالت زمان اندازه گرفته شده است.



این تصور زمان مورد استفاده هر کدام از configure های مختلف است. که بعد از اجرای ۱۰۰ بار code مورد نظر در زبان c می باشد.

تعداد core های cpu لپتاب من برابر7 است.(core i7) پس بنابراین هر چقدر تعداد process های ساخته شده به 7 نزدیک تر باشد، زمان اجرایی کمتر است.

همانطور که دیده می شود در حالت های:

6 1 -> 12 processes will be created.
3 2 -> 9 processes will be created.
2 3 -> 8 processes will be created.
1 6 -> 7 processes will be created.

بنابراین حالت ۱ ۶ از همه تعداد process ها کمتر است و همچنین بیشترین مقدار موازی سازی را دارا است. پس باید از همه زمان کمتر باشد ، به طور کلی پس از run کردن ۱۰۰ بار کد مورد نظر حالت های ۲ ۳ و ۱ ۶ از همه تایمشان کمتر بود.

در زیر بخش اصلی کد که به ساختن فرایند های سطح ۱ و سطح ۲ می باشد می پردازیم.

```
for (int i = 0; i < command_st->proccess_count; i++)
{ // m
    memset(write_msg, 0, buff_size);

    pid_t pid = fork();
    if (pid < 0)
    {
        ERROR_HANDLER_AND_DIE("cant make the process.");
    }
    else if (pid == 0) // child
    {
        int fd_child[2 * command_st->child_proccess_count];
        pipe_creator(fd_child, command_st->child_proccess_count);
        pid_t process_childs[command_st->child_proccess_count];
        ChildInfo created_process_write[command_st->child_proccess_count];
        read(fd[2 * i + READ_END], read_msg, buff_size);

        for (int j = 0; j < command_st->child_process_count; j++)
        { // n
            memset(write_msg_child, 0, buff_size);

            pid_t child_pid = fork();

            if (child_pid < 0)
            {
                ERROR_HANDLER_AND_DIE("cant make the childs process.");
            }
            else if (child_pid == 0) // child of child
            {
                child_process(read_msg_child, fd_child, buff_size, j, 0);
            }

            strcpy(
                write_msg_child,
                split_the_generated_commnad(read_msg, j, command_st->child_proccess_count));
            process_childs[j] = child_pid;
            created_process_write[j] = create_new_process_info(
                child_pid, getpid(), IS_CHILD, IS_NOT_PARRENT, j + 1, i + 1);

            write(fd_child[2 * j + WRITE_END], write_msg_child, buff_size);
        }
```

در این بخش از کد ابتدا به اندازه سطح یک فرایند ساخته می شود سپس به اندازه سطح ۲ فرایند می سازیم.

```c
void child_process(char read_msg_child[], int fd_child[], int buff_size, int pipe_number, int retry)
{
    memset(read_msg_child, 0, buff_size);
    read(fd_child[pipe_number * 2 + READ_END], read_msg_child, buff_size);
    char **args = malloc(sizeof(char *) * MAX_ARGS_LEN);
    int args_len = parse_command_to_be_executed(read_msg_child, args);
    long time_of_command = strtol(args[args_len], NULL, 10);
    usleep(time_of_command * 1000);
    args[args_len] = NULL;

    if (!retry)
        dup2(fd_child[pipe_number * 2 + WRITE_END], STDOUT_FILENO);

    if ((execv(args[0], &args[1]) < 0))
    {
        char *err = (char *)malloc(sizeof(char) * LINE_SIZE);
        sprintf(err, "warning!!!!!!!: couldnt execute!. command: %s\n", read_msg_child);
        ERROR_HANDLER_AND_DIE(err);
    }
}
```

در این بخش از کد ابتدا به اندازه زمان گفته شده صبر می کنیم سپس دستور مورد نظر را اجرا می کنیم. و خروجی دستور را در pipe می نویسم. و به سطح بالا تر می فرستیم.

```
    for (int j = 0; j < command_st->child_proccess_count; j++)
    {
        int status;
        int t = 0;
        waitpid(process_childs[j], &status, 0);
        // retry
        if (status != 0)
        {
            pid_t ppid = fork();
            if (ppid == 0)
            {
                child_process(read_msg_child, fd_child, buff_size, j, 1);
            }
            strcpy(write_msg_child,
                   split_the_generated_commnad(read_msg, j, command_st->child_proccess_count));
            write(fd_child[2 * j + WRITE_END], write_msg_child, buff_size);
            wait(NULL);
            t = 1;
        }
        set_end_time_and_status_for_terminated_process(
            NULL, created_process_write, process_childs[j],
            status, command_st->child_proccess_count, t);
    }
    for (int j = 0; j < command_st->child_proccess_count; j++)
    {
        char reader[buff_size * 2];
        read(fd_child[2 * j + READ_END], reader, sizeof(reader));
        strcpy(created_process_write[j].output, reader);
    }
    ssize_t byets_written = write(
        fd[2 * i + WRITE_END], created_process_write,
        sizeof(ChildInfo) * command_st->child_proccess_count);
    exit(0);
}
children_id[i] = pid;
ChildInfo dummy;
all_childs_count = enqueue(
    childs,
    create_new_process_ptr_info(pid, main_routing_id, IS_NOT_CHILD, IS_PARRENT, i + 1, -1),
    dummy,
    all_childs_count, &counter);
```

در این بخش از کد به اندازه فرایند های سطح ۲ ابتدا فرایند می سازیم سپس برای این فرایند های ساخته شده با استفاده از تابع waitpid صبر می کنیم. اگر با status ب غیر از 0 فرایند سطح ۲ تمام شده باشد یک فرایند دیگر ساخته می شود. سپس این فرایند های ساخته شده را به سطح بالا تر می فرستیم.

```c
void handeling_wait_for_proccess(
    Command *command_st,
    int counter,
    int is_parrent,
    int fd[],
    ChildInfo childs_reader[],
    ChildInfo **holder,
    int *all_childs_count,
    pid_t children_id[],
    ChildInfo **childs,
    int *count)
{
    for (int i = 0; i < counter; i++)
    {
        // memset(childs_reader, 0, sizeof(ChildInfo) * command_st->child_proccess_count);
        int status;
        waitpid(children_id[i], &status, 0);
        set_end_time_and_status_for_terminated_process(
            holder, NULL, children_id[i], status, command_st->commands_count, 0);
    }

    for (int i = 0; i < command_st->proccess_count; i++)
    {
        ssize_t readed_bytes_size = read(fd[2 * i + READ_END],
                                         childs_reader,
                                         sizeof(ChildInfo) * command_st->child_proccess_count);
        if (readed_bytes_size > 0)
            for (int j = 0; j < command_st->child_proccess_count; j++)
                *all_childs_count = enqueue(childs, NULL, childs_reader[j], *all_childs_count, count);
    }
}
```

در این بخش از کد برای فرایند های سطح ۱ با استفاده از تابع waitpid صبر می کنیم. سپس به خواندن pipe می پردازیم تا فرایند های ایجاد شده را از pipe بخوانیم.

شرح عملکرد:

شروع کد از فایل main.c تابع start_execution است.
کد اصلی ساختن process های مختلف در فایل process_creator.c قرار دارد، در تابع creating_process. ابتدا configure های مختلف ساخته می شوند سپس این تابع با configure های مورد نظر صدا زده می شود. و زمان اجرای این تابع برای configure های مختلف اندازگیری می شود، سپس یک فایل json از configure های مختلف به همراه تمام فرایند های ساخته شده در ان configure به عنوان خروجی داده می شود. Struct ChildInfo

تمام اطلاعات فرایند های ساخته شده را نگه داشته است.

```c
typedef struct
{
    pid_t parrent_id;
    pid_t id;
    struct timeval start_time;
    struct timeval end_time;
    double execution_time;
    int is_parrent;
    int is_child;
    int process_number;
    int parrent_number;
    int exit_status;
    char *command;
    int number_of_trys;
    char output[LINE_SIZE * 10];
} ChildInfo;
```

```c
typedef struct
{
    int configuration[2];
    ChildInfo **process_created_inconfiguration;
    int childs_size;
    struct timeval start, end;
    double execution_time;
    double actual_time;
} ProcessConfigurations;
```

در صورت غیر موفقیت امیز بودن اجرا دوباره  فرایند جدید ساخته شده و انجام می شود.

فایل های ضمیمه:

فایل های c در فولدر src قرار دارند.

Main.c
Utility_functions.c
Process_creatort.c
Headers.h

که utility_functions دارای تابع های نظیر خواندن از فایل و parse کردن command ها است. Headers دارای struct ها و difinition تابع ها است.

Makefile
.gitignore
.git
.dockerignore
Dockerfile
Requirements.txt
Output.json // فایل اجرایی که زمان های ان در نمودار بالا امده است.

Monitor.py
که در این فایل می توان نمودار ها را کشید و همچنین می توان تعداد دفعات run شدن فایل c را تعیین کرد.
به ۴ شیوه می توان برنامه را اجرا کرد.

-> Make run
Or
-> (venv) pip install -r requirements.txt
-> (venv) Python Monito.py // بار اجرا می کند 10 اگر مقدار نداشته باشد
Or
-> (venv) pip install -r requirements.txt
-> (venv) python monitor.py 10 // بار کد زبان سی را اجرا می کند 10
Or
-> docker build -t image_name
-> docker run image_name // البته gcc ): اینجا خیلی جالب رفتار نمی کنه