Системні виклики
Лабораторна робота №1

Налаштування середовища

Попередньо було обумовлено, що на лабораторних роботах основна частина буде саме про написання ядра операційної системи з використанням мови rust. Перша лабораторна робота фокусується на використанні системних викликів. Поки ядро ще не пишемо.

В лабораторних роботах будуть використовуватися засоби доступні на операційній системі Linux. Однак, якщо студент чи студентка не зацікавлені в використанні ОС Linux, він чи вона можуть виконувати завдання використовуючи іншу операційну систему та набір засобів, але це означає, що студент чи студентка мають самостійно провести аналіз засобів. Більшість задач можна буде виконати майже ідентичним чином використовуючи MacOS. Зазвичай виконання декількох команд в терміналі - це все що треба. Ви можете використати один з наступних варіантів (від більш радикальних до менш радикальних):

1. Встановити Linux, як основну систему. Дистрибутив - не принципово, рекомендованими є Ubuntu, Linux Mint. На сайті дистрибутивів є інструкції, як встановити ОС.

2. Встановити Linux, як другу систему, якщо у вас, наприклад, встановлений Windows. Є багато інструкцій, як це можна зробити. Сам процес відносно простий. Почитати можна, наприклад, за посиланням тут чи тут.

3. Встановити Linux, як операційну систему всередині віртуальної машини. Мабуть, найпопулярнішим вибором тут буде Virtual Box + Ubuntu. Інструкція, як це можна зробити, наприклад, доступна за посиланням ось тут.

4. Використати Docker. Docker доступний на всіх сучасних операційних системах. На Windows та MacOS Docker запускається у з використанням засобів віртуалізації, але його має бути достатньо, щоб скомпілювати та зачасту виконувати програми. Наприклад, його достатньо для виконання цієї лабораторної роботи. Інструкція ось тут.

5. Використати WSL 2, якщо у вас Windows, можна спробувати. Але можливо треба буде робити додаткові зусилля для отримання програмного забезпечення, яке потрібно в лабораторних роботах. Ось інструкція.

Оскільки Docker буде корисний і для інших речей, то зробимо припущення, що ви обрали саме віріант №4. Нехай `D:/os-labs-2022` - це директорія з кодом вашої лабораторної, тоді виконавши наступну команду з командного рядку ви створите контейнер з усіма потрібними засобами у ньому. Зберігаєте результат в `D:/os-labs-2022`, а тестуєте в контейнері оперуючи тим же контентом в `/os-labs`.

```
docker run -ti -v D:/os-labs-2022:/os-labs yevhenii0/ubuntu-os:22.04.1
```

У випадку, якщо у вас є Linux встановіть `gcc` та `pstree` - `sudo apt-get install -y gcc psmisc`.

Контекст

Системні виклики - це механізм використання сервісів операційної системи. Кожна операційна система має свій набір системних викликів. Дана лабораторна робота дозволить засвоїти основи роботи з системними викликами на прикладі створення та маніпулювання процесами операційної системи.

```
1 #include "sys/wait.h"
2 #include "stdio.h"
3 #include "unistd.h"
4
5 int main(int argc, char *argv[]) {
6     int pid = fork();
7     if (pid == 0) {
8         printf("child process, pid %d\n", getpid());
9         sleep(5);
10        execlp("/bin/echo", "echo", "Hey!", NULL);
11        printf("child process 2, pid %d\n", getpid());
12    } else {
13        waitpid(pid, NULL, 0);
14        printf("parent process, pid %d\n", getpid());
15    }
16    return 0;
17 }
```

Пояснення основних елементів програми, номер в списку - номер рядка в програмі.

6. Створення нового процесу, що є копією поточного. Зверніть увагу, що виконання програми в дочірньому процесі продовжиться точно з такого ж місця де буде батьківський процес після виклику функції `fork()`. Єдина різниця (окрім деталей копіювання ресурсів, які можна дізнатися з опису команди - `man fork`) це те, що після виклику `fork()` батьківський і дочірній процеси будуть мати різне значення в змінній `pid`. Для батьківського процесу змінна `pid` буде містити ідентифікатор дочірнього процесу. Для дочірнього процесу змінна `pid` буде містити значення `0`.

7. Перевірка значення `pid` дає змогу зрозуміти в якому процесі програма продовжує виконання. Так, наприклад, перевірка на рівність нулю буде істинною для дочірнього процесу та хибною для батьківського.

8. Вивести ідентифікатор процесу в якому виконується програма. Щоб отримати ідентифікатор використовується функція `getpid()`. У даному випадку буде виведений ідентифікатор дочірнього процесу.

9. Не виконувати подальні інструкції наступні 5 секунд (заснути).

10. Замістити програму, що виконується - програмою що вказана в параметрах. У даному випадку це означає виконати команду `/bin/echo Hey!`. Оскільки програма була повністю заміщена новою програмою, дочірній процес завершиться після виконання вказаної програми, тому рядок 11 ніколи не буде виконаний. Для ознайомлення з можливими варіаціями `exec` можна почитати мануал - `man exec`.

13. Зачекати поки процес з ідентифікатором `pid` (дочірній процес) закінчить виконання. Зверніть увагу, що `else` гілка - це гілка, яку виконує батьківський процес, оскільки в змінній `pid` знаходиться ідентифікатор дочірнього процесу (не нуль).

Описаних функцій є достатньо для виконання завдань в цій лабораторній роботі.

Завдання

Перше завдання (обов'язкове)

Створити програму, яка породжує дерево процесів відповідно до варіанту. Рекомендовано використати функцію `sleep` із значенням достатньо великим, щоб можна було перевірити структуру дерева процесів. Нехай main.c - це код вашої програми. Тоді для перевірки правильності виконання буде використаний наступний процес.

```
gcc main.c -o lab1_task1
./lab1_task1 &
lab1_task1_pid=$(echo $!)
pstree -A -p $lab1_task1_pid
```

В заданих деревах ідентифікатори не є важливими, як не є важливою послідовність створення, головне повторити структуру. Варіанти:

1.
```
p1
\_ p2
|   \_ p3
\_ p4
|   \_p5
|   \_p6
\_ p7
```

2.
```
p1
\_ p2
|   \_ p3
|   \_ p4
|   \_ p5
|_ p6
    \_ p7
```

3.
```
p1
\_ p2
    \_ p3
        \_ p4
            \_ p5
                \_ p6
                    \_ p7
```

4.
```
p1
\_ p2
|   \_ p3
\_ p4
|   \_ p5
|       \_ p6
\_ p7
```

5. p1
   _ p2
   _ p3
   _ p4
   _ p5
   | _ p6
   _ p7

Друге завдання (обов'язкове)

Написати програму, що виконує іншу програму передану в командному рядку, як параметр. Весести `"Success!"`, якщо програма завершилась успішно та вивести `"Failed, exit code = ?"` (де знак питання це значення коду), якщо виконання не було успішним. Для виконання переданої програми можете використати довільну `exec` функцію, зручною є `execvp`. При формуванні параметрів виклику `execvp(char *f, char *argv[])` зверніть увагу на те, що першим елементом в масиві `argv` має бути назва програми (у нашому випадку `*f == argv[0]`, а останнім елементом `NULL`. Таким чином, якщо нам потрібно виконати програму `sleep 100`, то передати в `execvp` треба буде `char *f = "sleep"`, `char *argv[3] = {"sleep", "100", NULL}`. Для отримання статусу виконання процесу (exit code) треба використати другий параметр функції `waitpid(pid, &status, 0)`.

Нехай main.c - це код вашої програми, тоді приклад використання команд для перевірки:

```
gcc main.c -o lab1_task2
./lab1_task2 sleep 5
./lab1_task2 echo Hello everybody!
./lab1_task2 cat non_existing_file
```

Третє завдання (на додаткові бали)

Написати програму, що зможе створити довільне дерево процесів, яке передається у довільному форматі (як автор програми захоче) з командного рядка. Тестування буде в індивідуальному порядку.

Здача роботи

Посилання на github репозиторій з результатом залишити в класрумі. Обговорювати будемо в індивідуальному порядку. Для зручності можете використати ось цей шаблон.