5 задача

Создание, завершение процесса

1. Жизненный цикл процесса.
   a. Напишите программу, которая:
   i.
   создает и инициализирует переменную (можно две: локальную и
   глобальную);
   ii.
   выводит ее (их) адрес(а) и содержимое;
   iii.
   выводит pid;
   iv.
   порождает новый процесс (используйте fork(2)).
   v.
   в дочернем процессе выводит pid и parent pid.
   vi.
   в дочернем процессе выводит адреса и содержимое переменных,
   созданных в пункте a;
   vii.
   в дочернем процессе изменяет содержимое переменных и
   выводит их значение;
   viii.
   в родительском процессе выводит содержимое переменных;
   ix.
   в родительском процессе делает sleep(30);
   x.
   в дочернем процессе завершается с кодом “5” (exit(2)).
   xi.
   в родительском процессе дожидается завершения дочернего,
   вычитывает код завершения и выводит причину завершения и код завершения если он есть. В
   каком случае кода завершения не
   будет?
   b. Объясните результаты работы программы.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

// Глобальная переменная
int global_var = 10;

int main() {
    // Локальная переменная
    int local_var = 20;

    // ii. Вывод адресов и содержимого переменных
    printf("Parent process - global_var: address=%p, value=%d\n", &global_var, global_var);
```

```c
    printf("Parent process - local_var: address=%p, value=%d\n", &local_var, local_var);

    // iii. Вывод PID родительского процесса
    printf("Parent process - PID: %d\n", getpid());

    // iv. Создание нового процесса
    pid_t child_pid = fork();

    if (child_pid == -1) {
        perror("fork failed");
        exit(1);
    }

    if (child_pid == 0) {
        // Код выполняется в дочернем процессе
        // v. Вывод PID и PPID дочернего процесса
        printf("Child process - PID: %d, PPID: %d\n", getpid(), getppid());

        // vi. Вывод адресов и содержимого переменных в дочернем процессе
        printf("Child process - global_var: address=%p, value=%d\n", &global_var, global_var);
        printf("Child process - local_var: address=%p, value=%d\n", &local_var, local_var);

        // vii. Изменение переменных в дочернем процессе
        global_var = 100;
        local_var = 200;
        printf("Child process - changed values: global_var=%d, local_var=%d\n", global_var,
local_var);

        // x. Завершение дочернего процесса с кодом 5
        exit(5);
    } else {
        // Код выполняется в родительском процессе
        // viii. Вывод содержимого переменных в родительском процессе
        printf("Parent process - values after fork: global_var=%d, local_var=%d\n", global_var,
local_var);

        // ix. Ожидание 30 секунд
        sleep(30);

        // xi. Ожидание завершения дочернего процесса и получение статуса
        int status;
        pid_t terminated_pid = wait(&status);

        if (terminated_pid == -1) {
            perror("wait failed");
        } else {
            if (WIFEXITED(status)) {
                printf("Child process %d exited with status %d\n", terminated_pid,
WEXITSTATUS(status));
            } else if (WIFSIGNALED(status)) {
                printf("Child process %d was killed by signal %d\n", terminated_pid,
WTERMSIG(status));
            } else if (WIFSTOPPED(status)) {
                printf("Child process %d was stopped by signal %d\n", terminated_pid,
WSTOPSIG(status));
            }
        }
    }

    return 0;
}
```

Построчное объяснение кода:

1. `#include` - Подключение необходимых заголовочных файлов:
   - `stdio.h` - для ввода/вывода
   - `stdlib.h` - для exit()
   - `unistd.h` - для fork(), sleep(), getpid(), getppid()
   - `sys/types.h` и `sys/wait.h` - для wait()
2. `pid_t child_pid = fork();` - Создание нового процесса (пункт iv). После этого вызова будут выполняться два процесса.

`pid_t` - это **целочисленный тип данных**, используемый для хранения идентификаторов процессов. Это не структура.

`fork()` - это системный вызов в Unix-подобных системах, который создаёт **новый процесс** (дочерний), являющийся почти точной копией текущего процесса (родительского).

1. **Копирование структур ядра**:
   - Ядро создаёт новую запись в таблице процессов
   - Копирует структуру `task_struct` родительского процесса (это основная структура данных ядра, описывающая процесс)
2. **Копирование адресного пространства**:
   - Раньше создавалась полная копия памяти родительского процесса
   - Современные системы используют **Copy-On-Write (COW)**:
     - Память помечается как доступная только для чтения
     - Реальное копирование происходит только при попытке записи
3. **Настройка метаданных**:
   - Новому процессу назначается уникальный PID
   - PPID (Parent PID) устанавливается равным PID родителя
   - Сбрасываются статистические счётчики
4. **Возврат из вызова**:
   - В родительском процессе возвращается PID дочернего
   - В дочернем процессе возвращается 0

Особенности:

- После `fork()` оба процесса продолжают выполнение **с той же точки**
- Различать процессы можно по возвращаемому значению
- Все открытые файловые дескрипторы копируются (но ссылаются на одни и те же файлы)

2. `exit(5);` - Завершение дочернего процесса с кодом 5 (пункт x)
   Код завершения `5` в `exit(5)` - это произвольное значение, возвращаемое процессом операционной системе и родительскому процессу для указания статуса завершения.(по заданию попросили)
3. `int status; pid_t terminated_pid = wait(&status);` - Ожидание завершения дочернего процесса и получение статуса (пункт xi)

Функция `wait()` - это системный вызов в UNIX/Linux, который позволяет родительскому процессу ожидать завершения любого из своих дочерних процессов и получать информацию о статусе завершения.

```c
pid_t wait(int *status);
```

Возвращаемое значение

- **Успешное выполнение**:
  - Возвращает PID завершившегося дочернего процесса
  - Записывает статус завершения в переменную, на которую указывает `status`
- **Ошибка**:
  - Возвращает `-1` и устанавливает `errno` (например, если нет дочерних процессов)

Статус, записываемый в `status`, содержит информацию о том, как завершился процесс. Для его анализа используются специальные макросы:

1. Проверка типа завершения

Макрос	Описание
`WIFEXITED(status)`	Возвращает true, если процесс завершился нормально (через `exit()` или возврат из `main()`)
`WIFSIGNALED(status)`	Возвращает true, если процесс был завершен сигналом
`WIFSTOPPED(status)`	Возвращает true, если процесс был остановлен (не завершен)
`WIFCONTINUED(status)`	Возвращает true, если процесс продолжил выполнение после остановки

2. Получение деталей завершения

Макрос	Когда использовать	Что возвращает
`WEXITSTATUS(status)`	Если `WIFEXITED` true	Код завершения (0-255)
`WTERMSIG(status)`	Если `WIFSIGNALED` true	Номер сигнала, завершившего процесс
`WCOREDUMP(status)`	Если `WIFSIGNALED` true	Возвращает true, если создан core dump
`WSTOPSIG(status)`	Если `WIFSTOPPED` true	Номер сигнала, вызвавшего остановку

4. `if (WIFEXITED(status))` - Проверка, нормально ли завершился процесс
5. `printf("Child process %d exited with status %d\n"...` - Вывод кода завершения дочернего процесса
6. Остальные условия проверяют другие причины завершения (по сигналу и т.д.)

Объяснение результатов работы программы:

1. После вызова fork() создается точная копия родительского процесса, включая все переменные. Однако изменения переменных в одном процессе не влияют на другой процесс.
2. Адреса переменных в родительском и дочернем процессах будут одинаковыми, но это виртуальные адреса - физически это разные области памяти.
3. Код завершения может отсутствовать, если процесс был завершен сигналом (например, kill -9).

Наблюдение в procfs:

1. Можно посмотреть информацию о процессах в `/proc/[pid]/maps` - там будут видны адресные пространства.

2. Состояния процессов можно увидеть в `/proc/[pid]/status` или с помощью `ps aux`.

c. Понаблюдайте за адресными пространствами в procfs.

d. Понаблюдайте за состояниями процесса в procfs или с помощью утилиты

Наблюдение за процессами через procfs и утилиты

c. Наблюдение за адресными пространствами в procfs

Procfs (виртуальная файловая система `/proc`) предоставляет детальную информацию о процессах.

Как исследовать адресное пространство процесса:

1. **Найти PID процесса**:

```
ps aux | grep <имя_процесса>
```

2. **Просмотреть карту памяти процесса**:

```
cat /proc/<PID>/maps
```

или более читаемо:

```
pmap -x <PID>
```

Вывод показывает:
- Адресные диапазоны
- Права доступа (r/w/x)
- Смещение в файле
- Устройство
- Inode
- Имя файла/области

3. **Пример вывода `/proc/<PID>/maps`**:

```
00400000-00401000 r-xp 00000000 08:01 123456 /path/to/program
00600000-00601000 r--p 00000000 08:01 123456 /path/to/program
00601000-00602000 rw-p 00001000 08:01 123456 /path/to/program
7ffd100000-7ffd300000 rw-p 00000000 00:00 0    [heap]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0    [vsyscall]
```

4. **Дополнительные файлы для анализа**:
- `/proc/<PID>/smaps` - детальная информация об использовании памяти
- `/proc/<PID>/statm` - общая статистика по памяти
- `/proc/<PID>/exe` - симлинк на исполняемый файл

d. Наблюдение за состояниями процессов

Через procfs:

1. **Состояние процесса**:

```
cat /proc/<PID>/status
```

Ищите строку `State:` (например, "R (running)", "S (sleeping)", "D (disk sleep)", "Z (zombie)")

2. **Краткая информация**:

```
cat /proc/<PID>/stat
```

Второе поле - состояние процесса (одна буква)

Через утилиты:

1. **`ps` - основная утилита**:

```
ps aux
```

Колонка `STAT` показывает состояние:

- `R` - выполняется
- `S` - спит (прерываемый)
- `D` - непрерываемый сон (обычно I/O)
- `T` - остановлен
- `Z` - зомби

Более детально:

```
ps -eo pid,state,cmd
```

2. **`top` / `htop` - интерактивный мониторинг**:

```
top
```

или

```
htop
```

В колонке `S` (State) отображается состояние каждого процесса.

3. **`pstree` - просмотр дерева процессов**:

```
pstree -p
```

Коды состояний процессов (STAT в ps)

Код	Описание
R	Running (выполняется)
S	Sleeping (прерываемый сон)
D	Disk sleep (непрерываемый сон, обычно I/O)
T	Stopped (остановлен сигналом)
t	Tracing stop (остановлен отладчиком)
Z	Zombie (процесс-зомби)
X	Dead (полностью завершен)

Флаг	Значение
<	Высокий приоритет
N	Низкий приоритет
L	Заблокированные страницы в памяти
s	Лидер сессии
l	Многопоточный
+	Процесс в foreground группе

Процесс в состоянии зомби:

a. Модифицируйте предыдущую программу так чтобы дочерний процесс становился зомби.
b. Объясните какую проблему решает данное состояние.
c. Может ли родительский процесс оказаться в состоянии зомби? Если да, то что в этом случае произойдет с дочерним? Смоделируйте эту ситуацию.
4. Создание процесса

Что происходит при вызове fork:
У каждого процесса есть структура **task_struct**

task_struct — это структура в ядре Linux, которая описывает процесс и содержит всю информацию, необходимую для его нормальной работы. 1

Также её называют дескриптором процесса. 2

Некоторые основные категории информации в task_struct:

1. **Идентификатор** (pid) — уникальный идентификатор процесса, который используется для его отличия от других процессов. 1
2. **Состояние** (state) — состояние задачи, код выхода, сигнал выхода и т. д.. 1
3. **Приоритет** (priority) — приоритет относительно других процессов. 1

4. **Счётчик программы** (program counter) — адрес следующей инструкции, которую нужно выполнить в программе. 1
5. **Указатель на память** (memory pointer) — включает указатели на код программы и данные, связанные с процессом, а также указатели на блоки общей памяти с другими процессами. 1
6. **Данные контекста** (context data) — данные в регистрах процессора во время выполнения процесса. 1
7. **Информация о состоянии ввода-вывода** (I/O state information) — включает отображаемые запросы ввода-вывода, выделенные устройства ввода-вывода процесса и список используемых процессом файлов. 1
8. **Информация для учёта** (accounting information) — может включать общее время процессора, общее время такта, лимиты времени, номера учёта и т. д.. 1

Описание структуры task_struct можно найти в файле include/linux/sched.h исходников ядра.

Каждый процесс имеет своё **собственное адресное пространство** , которое представлено:

🔹 `mm_struct`

Структура, описывающая память процесса. В ней есть:

Поле	Назначение
`pgd` (Page Global Directory)	Указатель на корень таблицы трансляции (page tables)
`mmap`	Список отображённых областей памяти (например, код, данные, стек)
`start_code` , `end_code` , `start_data` , `end_data`	Адресные границы различных секций
`rss`	Число страниц, выделенных процессу в физической памяти

Когда ты вызываешь `fork()` , ядро:

1. Создаёт новый `task_struct` для дочернего процесса.
2. **Копирует содержимое** `mm_struct` **родителя** .
3. **Создаёт новую таблицу трансляции (page tables)** , но она ссылается на **те же физические страницы** , что и у родителя.
4. Помечает эти страницы как **read-only** → это механизм **Copy-on-Write (COW)** .

Как происходит копирование page tables при `fork()` ?

При `fork()` :

1. Ядро выделяет новую структуру `task_struct` и `mm_struct` .
2. Таблицы трансляции копируются **глубоко** — создается **новая иерархия PGD/PUD/...** , но:
   - Все записи указывают на **те же физические страницы** , что и у родителя.
   - Эти страницы помечаются как **read-only** для обоих процессов.
3. При попытке **записи** в такую страницу возникает **page fault** → COW.

Вот **9 ключевых отличий между родительским и дочерним процессами** после вызова `fork()`:

---

1. PID (Process ID)

- **Родитель** : имеет свой уникальный PID.
- **Дочерний** : получает новый, уникальный PID.

> Пример: `getpid()` возвращает разные значения в каждом процессе.

---

2. PPID (Parent Process ID)

- **Родитель** : его PPID — это PID его собственного родителя.
- **Дочерний** : его PPID равен PID родительского процесса.

> `getppid()` показывает, кто является родителем.

---

3. Возвращаемое значение `fork()`

- **Родитель** : получает PID дочернего процесса (`pid > 0`).
- **Дочерний** : получает `0` .

> Это основной способ, как код определяет, в каком процессе он выполняется.

---

4. Счётчики ссылок на ресурсы

- Некоторые ресурсы (например, файловые дескрипторы) копируются, но их **счётчики ссылок увеличиваются** .

> Например, если оба процесса закроют один и тот же файл, реальное закрытие произойдёт только при последнем `close()` .

---

5. Состояние памяти (COW)

- Оба процесса имеют **одинаковое содержимое памяти** , но:
  - Память помечена как **Copy-on-Write (COW)** .
  - При изменении какой-либо переменной в одном из процессов — создаётся **копия страницы** .

> Таким образом, изменения в одном процессе **не влияют** на другой.

---

6. Порядок выполнения

- Ядро может запускать процессы **в любом порядке** — планировщик решает, кому отдать процессор первым.
- Без синхронизации (например, `sleep()`, `wait()`) нельзя предсказать, что выполнится раньше: родитель или ребёнок.

---

7. Таблицы управления памятью

- У каждого процесса своё **адресное пространство**.
- Виртуальные адреса совпадают, но физически используются разные страницы (после COW).

> `%p` для переменных будет одинаковым, но это **виртуальная память**, а не физическая.

---

8. Зависимость времени жизни

- Родительский процесс **может ожидать завершения дочернего** через `wait()` / `waitpid()`.
- Если родитель завершается раньше, чем ребёнок — дочерний становится "сиротой" и прикрепляется к `init` (или `systemd`).

---

9. Ресурсы и ограничения

- Дочерний процесс **наследует большинство ресурсов** родителя (файловые дескрипторы, текущую директорию, среду и т.д.), но:
  - Ограничения по ресурсам (RLIMIT_*) могут быть унаследованы с ограничениями.
  - Некоторые специфичные данные (например, POSIX-таймеры) **не копируются**.

---

2. Процесс в состоянии зомби

Модифицированная программа для создания зомби-процесса:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t child_pid = fork();

    if (child_pid == -1) {
        perror("fork failed");
        exit(1);
    }

    if (child_pid == 0) {
        // Дочерний процесс
        printf("Child process - PID: %d\n", getpid());
        exit(0);
```

```
    } else {
        // Родительский процесс
        printf("Parent process - PID: %d, child PID: %d\n", getpid(), child_pid);
        printf("Parent is sleeping for 60 seconds, child will become zombie\n");
        sleep(60); // Не вызывает wait() для дочернего процесса
    }

    return 0;
}
```

Объяснение:

1. Зомби-процесс (процесс-зомби) возникает, когда дочерний процесс завершается, но родительский процесс не вызвал wait() для получения его статуса завершения.
2. Проблема, которую решает состояние зомби:
   - Это механизм ядра для хранения информации о завершенном процессе до тех пор, пока родительский процесс не прочитает его статус.
   - Позволяет родительскому процессу узнать, как завершился дочерний процесс.
3. Может ли родительский процесс стать зомби:
   - Да, если родительский процесс завершится до своего дочернего процесса, а новый родитель (обычно init, PID 1) не вызовет wait().
   - В этом случае дочерний процесс становится "осиротевшим" и будет унаследован процессом init, который периодически вызывает wait() для всех своих дочерних процессов, предотвращая их превращение в зомби.