Системное программирование

Введение.

Системное программирование — это область программирования, направленная на создание системного программного обеспечения, которое управляет аппаратными и программными ресурсами компьютера. Это включает операционные системы, драйверы устройств, утилиты, и другие программы, которые обеспечивают функционирование компьютерной системы на низком уровне.

Сферы применения:

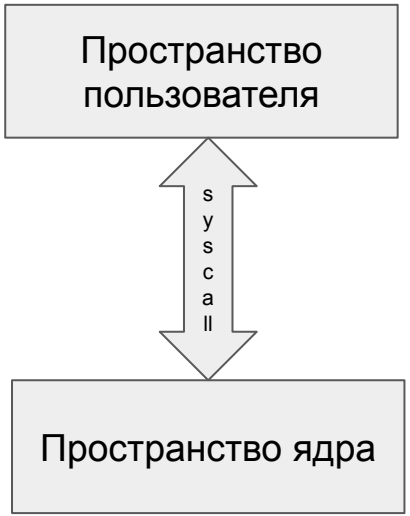
1. **Разработка ОС:** Системные программисты создают и поддерживают операционные системы (например, Windows, Linux, macOS), которые являются основой работы компьютеров и устройств.
2. **Драйверы устройств:** Создание драйверов, которые обеспечивают взаимодействие операционной системы с аппаратными компонентами компьютера, такими как принтеры, видеокарты, сетевые адаптеры и т.д.
3. **Системные утилиты:** Разработка программ, которые выполняют специализированные задачи, такие как управление файловой системой, архивирование данных, мониторинг состояния системы и т.д.
4. **Интеграция с аппаратным обеспечением:** Работа с микроконтроллерами и встроенными системами, разработка прошивок (firmware) для различных устройств, таких как бытовая техника, автомобили, медицинское оборудование и т.д.
5. **Компиляторы и интерпретаторы:** Создание компиляторов и интерпретаторов для языков программирования, которые переводят высокоуровневый код в машинные инструкции.
6. **Средства виртуализации:** Разработка программного обеспечения для виртуализации (например, VirtualBox, VMware), которое позволяет запускать несколько операционных систем на одном физическом компьютере.
7. **Сетевое программирование:** Разработка программ для управления сетевыми протоколами и коммуникациями, таких как маршрутизаторы, сетевые мосты, файерволы и другие сетевые устройства.
8. **Безопасность систем:** Создание программного обеспечения для защиты операционных систем и приложений от угроз, включая антивирусные программы, системы обнаружения вторжений и т.д.

ОС, определение, основные функции;

Операционная система (ОС) - программный комплекс, управляющий аппаратным обеспечением компьютера и предоставляющий доступ к нему.

Основные функции операционной системы:

1. Управление аппаратным обеспечением: распознавание устройств, операции ввода-вывода и др. stdin/stdout/stderr - потоки ввода/вывода/ошибок;
2. Предоставление доступа к программам через API.



Ядро ОС - часть ОС, управляющая ресурсами ПК + обработка запросов от программ.
Пространство ядра - защищенная часть памяти, где лежит код ядра ОС и данные, остальное пространство является общим пространством пользователя. Переход между пространствами ядра и пользователя осущ-ся через системные вызовы.

Системные вызовы: механизм взаимодействия программ с ядром ОС для выполнения операций. Пользовательские программы могут осуществлять системные вызовы напрямую (через инструкцию syscall/int). *Виды системных вызовов:*
Работа с процессами; Работа с файлами; Управление памятью; и др

Файловая система;

Файловая система — это структура и методы, используемые операционной системой для управления файлами на носителе данных. Она определяет, как данные хранятся, организуются и извлекаются. **Файл** – именованный набор данных.

Структура файловой системы — **дерево**.
Корень дерева — каталог с именем «/».
с / - *абсолютный путь*, **без /** - *относительный путь*;

Дескриптор файла — это уникальный идентификатор, который операционная система назначает файлу или другому вводу-выводу ресурсу (например, сокетам или каналам) для управления доступом к нему. Для каждого процесса создается **таблица дескрипторов**. Основная цель - управление открытыми файлами в процессе. Дескриптор от open - индекс в этой таблице. Каждый открытый файл имеет запись в общей таблице. Записи в дескрипторах ссылается на записи в таблице файлов.

- **Open()** создает новую запись.
- **Close()** закрывает запись.

Функции.

- Для открытия файлов исп-ся **вызов open();**
- Чтение и запись в файл производятся вызовами **read** и **write**.

Основные флаги открытия: O_CREAT: создание файла, O_APPEND: добавление в конец, O_RDONLY - чтение, O_WRONLY - запись, O_RDWR - чтение и запись.

Дырка в файле - область без данных, которую можно заполнить записью.

Пример open() и close()

```
#include <iostream>
#include <fcntl.h>
#include <unistd.h>
```

```
// Открытие файла для чтения и записи, создание файла, если он не существует
int fd = open("example.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
if (fd == -1) {
   perror("Error opening file");
   return 1;
}

// Закрытие файла
if (close(fd) == -1) {
   perror("Error closing file");
   return 1;
}
```

Пример с write()

```
// Запись данных в файл
const char* text = "Hello, World!\n";
ssize_t bytes_written = write(fd, text, strlen(text));
if (bytes_written == -1) {
   perror("Error writing to file");
   close(fd);
   return 1;
}
```

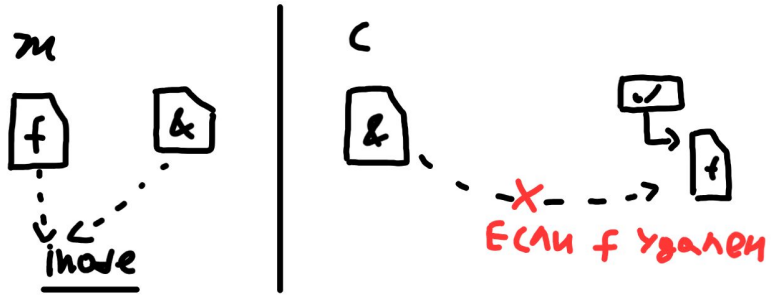
Пример с read()

```
// Чтение данных из файла
char buffer[128];
ssize_t bytes_read = read(fd, buffer, sizeof(buffer) - 1);
if (bytes_read == -1) {
   perror("Error reading file");
   close(fd);
   return 1;
}
```

Файловая система. Символьны/Жесткие ссылки.

Символьная ссылка (симлинк) — это специальный файл, который содержит путь к другому файлу или каталогу. Симлинки работают как указатели, перенаправляя доступ к целевому объекту.

Жесткая ссылка — это дополнительная ссылка на существующий файл, создающая новую запись в файловой системе, которая указывает на те же данные.



Символьная ссылка (symlink):

- тип файла, который содержит путь к другому файлу или директории.
- Может "сломаться", если файл удален.
- Отдельный файл.

Жесткая ссылка (link):

- альтернативное имя для существующего файла. (указывает на inode - узел с характеристиками файла)
- Все имена равноправны.
- Файл удаляется только после удаления всех ссылок.
- Нельзя создать циклы.

Файловая система. Удаление файла.

Для удаления файла исп-ся вызов unlink() *при этом все жесткие ссылки должны быть удалены, и программа закрыта*:

```
#include <iostream>
#include <unistd.h> // Для unlink()
#include <fcntl.h>  // Для open()
#include <cstring>  // Для strlen()
#include <cstdio>   // Для perror()

int main() {
   int fd = open("f.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
   if (fd == -1) {
       perror("Error opening file");
       return 1;
   }
   // Записываем данные в файл...
   // Закрываем файл
   if (close(fd) == -1) {
       perror("Error closing file");
       return 1;
   }
   // Удаляем файл
   if (unlink("f.txt") == -1) {
       perror("Error deleting file");
       return 1;
   }
   return 0;
}
```

UNIX. SH

В UNIX-системах, команды в командной оболочке (обычно это SH (Bourne Shell) или его вариации, такие как Bash) позволяют выполнять различные операции с файлами.

Команды/Функции:

- Для создания новой папки (директории) используется команда **mkdir**;
- Для перемещения по директориям используется команда **cd** (change directory);
- Для просмотра содержимого текущей директории используется команда **ls**;
- Для переименования файлов или папок используется команда **mv** (move);
- Для удаления файлов и папок используется команда **rm** (remove). Для удаления папок также используется опция -r (рекурсивное удаление) для удаления содержимого;
- Команда **cat** используется для вывода содержимого текстовых файлов в консоль;
- Команда **chmod** позволяет изменять права доступа к файлам и папкам: `chmod 755 file`
- Для поиска файлов в файловой системе используется команда find: `find /path/to/search -name "filename"`

SH команды;

- Команда pwd (print working directory) используется для вывода текущего рабочего каталога;
- Команда kill используется для отправки сигналов процессам, например, чтобы завершить процесс по его идентификатору (PID): `kill -9 PID`
- Для создания пустого файла вы можете использовать команду touch: `touch filename.txt`
- Для создания файла, заполненного нулями определенного размера, можно использовать команду dd:

  ```
  dd if=/dev/zero of=zeros_file bs=1024 count=1
  ```
- Чтобы создать файл размером 1.5 кбайт, можно использовать команду dd с параметрами bs (размер блока) и count (количество блоков): `dd if=/dev/zero of=1_5kb_file bs=1024 count=1.5`

- Для создания символьной ссылки на существующий файл используется команда ln -s:

  ```
  ln -s /path/to/existing_file symbolic_link
  ```
- Жесткие ссылки в UNIX создаются с помощью команды ln без опции -s:

  ```
  ln /path/to/existing_file hard_link
  ```

Пример файла: test.sh;

```
#!/bin/bash
# первая строка для интерпретатора
# Это комментарий
echo "Привет, мир!"
echo "Этот скрипт печатает текущее время:"
date
echo "Вы находитесь в директории:"
pwd
echo "Содержимое текущей директории:"
ls -l
```

Процессы.

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

Процесс — это выполняемая программа, включая текущее состояние выполнения. Каждый процесс имеет свое адресное пространство, ресурсы, дескрипторы файлов и другие атрибуты.

Процессы используются для изоляции и управления выполняемыми программами.

Они обеспечивают:

- Многозадачность: Одновременное выполнение нескольких программ.
- Безопасность и изоляцию: Процессы не могут напрямую обращаться к памяти других процессов, что повышает безопасность.
- Управление ресурсами: ОС может управлять ресурсами, выделяемыми каждому процессу, и приостанавливать или завершать процессы при необходимости.

Функция fork() используется для создания нового процесса, который называется "дочерним процессом". Дочерний процесс является копией родительского процесса, за исключением уникального идентификатора процесса (PID). PID=1 - родительский процесс, PID=0 потомок родителя;

Завершение процесса:

- waitpid() используется для ожидания завершения дочернего процесса и получения его статуса завершения.
- Макрос WIFEXITED(status) проверяет, завершился ли процесс нормально. Макрос WEXITSTATUS(status) возвращает код завершения процесса.
- одной из функций:
    a. exit, _exit, abort()

EXIT, _EXIT, ABORT

1. **void exit(int status);**
    a. **Описание:** Функция exit() используется для нормального завершения программы.
    b. **Поведение:**
        i. Выполняет очистку ресурсов, таких как закрытие открытых файлов и освобождение памяти.
        ii. Выполняет функции, зарегистрированные с помощью atexit().
        iii. Вызывает все деструкторы глобальных объектов C++.
        iv. Возвращает код завершения, указанный в аргументе status, родительскому процессу или вызывающей среде (например, оболочке).
2. **void _exit(int status);**
    a. **Описание:** Функция _exit() используется для немедленного завершения программы.
    b. **Поведение:**
        i. Не выполняет очистку ресурсов.
        ii. Не вызывает функции, зарегистрированные с помощью atexit().
        iii. Не вызывает деструкторы глобальных объектов C++.
        iv. Немедленно завершает процесс с указанным кодом завершения status.
3. **void abort();**
    a. **Описание:** Функция abort() используется для аварийного завершения программы.
    b. **Поведение:**
        i. Немедленно завершает программу и генерирует сигнал SIGABRT.
        ii. Создает дамп памяти (core dump), если это разрешено настройками системы, что может помочь в отладке.
        iii. Не выполняет функции очистки, зарегистрированные с помощью atexit().
        iv. Не вызывает деструкторы глобальных объектов C++.
        v. Возвращает код завершения, специфичный для аварийного завершения (обычно 134 на системах Unix).

getpid, getppid

- **getpid()** - функция, для получения идентификатора процесса PID. **Возвращаемое значение:** Функция возвращает идентификатор процесса, вызывающего эту функцию.
- **getppid()** - функция, для получения идентификатора процесса PID родителя. **Возвращаемое значение:** Функция возвращает идентификатор родительского процесса для текущего процесса.

```
pid_t pid = fork();  // Создание нового процесса

if (pid == -1) {
   return 1;
} else if (pid == 0) {
   // Дочерний процесс
   std::cout << "Child process: PID = "<< getpid() << ", Parent PID = " << getppid() << std::endl;
   _exit(0);  // Завершаем дочерний процесс
} else {
   // Родительский процесс
   std::cout << "Parent process: PID = "<< getpid() << ", Child PID = " << pid << std::endl;
   int status;
   waitpid(pid, &status, 0);  // Ожидаем завершения дочернего процесса
   if (WIFEXITED(status)) {
       std::cout << "Child process exited with status "<< WEXITSTATUS(status) << std::endl;
   }
}
```

Процесс сирота;

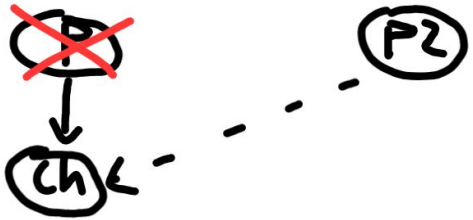
Процесс-сирота — это процесс, чей родительский процесс завершился или был убит до завершения самого процесса.

возникает:

- Когда родительский процесс завершает свою работу до завершения дочернего процесса, дочерний процесс становится сиротой.
- Операционная система (обычно Unix-подобные системы) автоматически назначает такие процессы-сироты какому-либо процессу-администратору, часто процессу с PID 1, обычно это init или systemd.

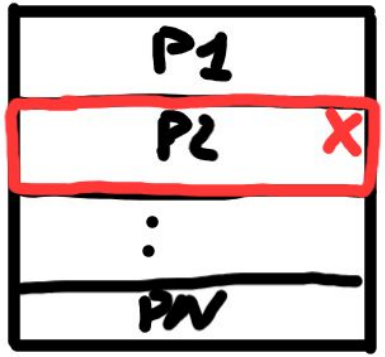
Поведение:

- После завершения родительского процесса, процесс-сирота становится под управлением init или systemd.
- init или systemd будет ожидать завершения процесса-сироты, чтобы правильно очистить его записи.



Процесс зомби;

Процесс-зомби — это процесс, который завершился, но его запись в таблице процессов еще не была удалена, так как родительский процесс еще не вызвал wait() для получения статуса завершения дочернего процесса.



возникает:

- Когда дочерний процесс завершает выполнение с помощью exit(), его запись остается в таблице процессов до тех пор, пока родительский процесс не вызовет wait() или waitpid().
- Эта запись сохраняется для хранения информации о завершении, которую родительский процесс может получить.

Поведение:

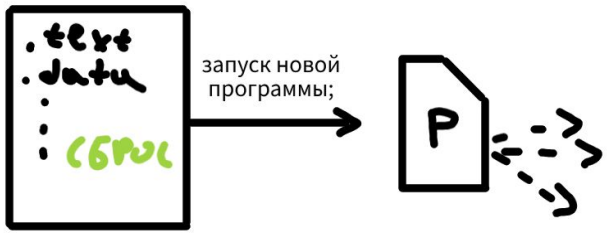
- Процессы-зомби занимают запись в таблице процессов, что может исчерпать ресурсы системы, если зомби-процессов становится слишком много.
- Процессы-зомби могут быть очищены только после вызова родительским процессом функции wait() или waitpid().

Процессы. Метод exec();

В операционных системах, основанных на Unix, метод exec() представляет собой семейство системных вызовов, которые используются для выполнения новой программы в текущем процессе. Эти вызовы заменяют текущий образ процесса (его код, данные, кучу и стек) новым образом, представленным исполняемым файлом.

Всего существует несколько различных функций exec(), каждая из которых выполняет новую программу в текущем процессе с разными способами передачи аргументов и переменных окружения:

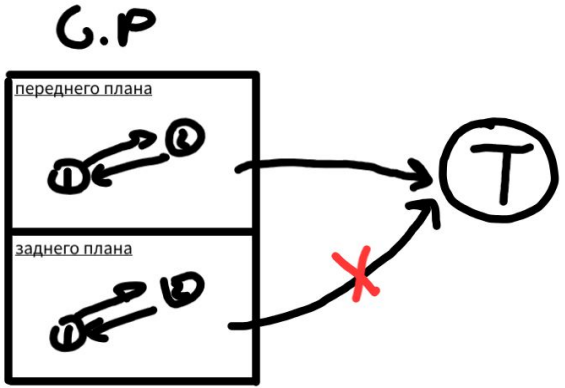
1. **execl() и execle()**
   - Они принимают список аргументов командной строки в виде отдельных параметров функции (аргументы передаются явно).
2. **execv() и execve()**
   - Они принимают список аргументов командной строки в виде массива строк (аргументы передаются через массив).
3. **execvp() и execvpe()**
   - Они аналогичны execv() и execve(), но в качестве первого аргумента принимают имя исполняемого файла без явного указания пути (поиск файла ведется согласно переменной окружения PATH).



Вызовы сбрасывают данные сегментов (.text, .data,..) и начинают новую программу.

Процессы. Группа процессов.

В операционных системах, таких как Unix и Linux, процессы могут быть объединены в группы, называемые группами процессов. Группа процессов является множеством процессов, связанных с общим идентификатором группы (PGID, Process Group ID). Группы процессов играют важную роль в управлении и коммуникации между процессами.



Основные аспекты групп процессов:

1. **Идентификатор группы процессов (PGID);**
2. **Создание группы процессов:** setpgid();
3. **Использование групп процессов:**
    - Группы процессов используются для организации и управления набором процессов.
    - Одним из распространенных применений групп процессов является управление процессами в рамках одного терминала (для управления сеансом ввода-вывода).
4. **Управление группами процессов:**
    - Функции управления группами процессов включают getpgid(), setpgid(), setpgrp() и getpgrp().
    - Например, функция setpgid(pid, pgid) устанавливает PGID процесса с идентификатором pid в значение pgid.

Сигналы.

Сигналы в операционных системах — это механизм асинхронной коммуникации между процессами или ядром операционной системы и процессами. Они используются для управления поведением процессов, например, для обработки ошибок, уведомлений о событиях или взаимодействия с пользователем.

Сигнал — это уведомление, которое отправляется процессу операционной системой или другим процессом.

Сигналы могут отправляться в ответ на события, такие как нажатие клавиши Ctrl+C, завершение дочернего процесса или другие асинхронные события.

Функции работы с сигналами:

- signal() — функция для установки обработчика сигнала.
- kill() — функция для отправки сигнала процессу или группе процессов.
- sigaction() — функция для установки детализированного обработчика сигнала с возможностью использования структуры struct sigaction.
- raise() — функция для генерации сигнала для текущего процесса.

Основные виды сигналов;

- **SIGINT** — сигнал прерывания (например, Ctrl+C), обычно приводит к завершению процесса.
- **SIGTERM** — сигнал завершения, используемый для просьбы о завершении процесса.
- **SIGKILL** — сигнал немедленного завершения, который не может быть перехвачен или игнорирован.
- **SIGSEGV** — сигнал нарушения сегментации, возникает при доступе к недопустимой области памяти.
- **SIGCHLD** — сигнал, отправляемый родительскому процессу при изменении статуса дочернего процесса (например, его завершении).

Блокировка сигналов;

В операционных системах сигналы могут быть **блокированы** или **игнорированы** процессами. Это позволяет контролировать, как процессы реагируют на асинхронные события, такие как нажатие клавиши Ctrl+C или завершение дочернего процесса. Блокировка сигналов позволяет временно отложить обработку сигналов в процессе. Это достигается установкой маски сигналов, которая определяет, какие сигналы будут блокированы и не могут быть доставлены процессу во время блокировки. В языке C/C++ для работы с маской сигналов используются функции из библиотеки signal.h.

Основные функции для работы с маской сигналов:

- **sigemptyset(sigset_t *set)**: Очищает множество сигналов (set), делая его пустым.
- **sigfillset(sigset_t *set)**: Заполняет множество сигналов (set) всеми возможными сигналами.
- **int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)**: Устанавливает маску сигналов для текущего процесса.

Для получения заблокированного сигнала без вызова обработчика можно использовать функцию sigwaitinfo(). Эта функция блокирует выполнение программы до тех пор, пока не будет получен один из сигналов, указанных в маске сигналов. Она возвращает информацию о полученном сигнале, что позволяет обработать его в приложении.

Пример-1. Создание процесса с уничтожением;

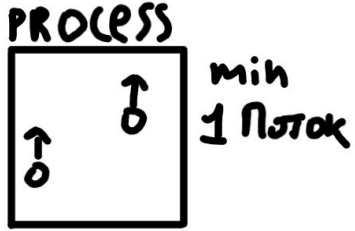
```
void foo() {
  // создание нового процесса
  pid_t pid = fork();


  if (pid < 0) {
      exit(1); // error
  } else if (pid == 0) {
      // блок дочернего процесса;
      // Отправляем сигнал SIGKILL родительскому процессу
      kill(getppid(), SIGKILL);
      exit(0);
  } else {
      // Род. процесс ожидает завершения дочернего процесса
      wait(NULL);
  }
}
```

Пример-2

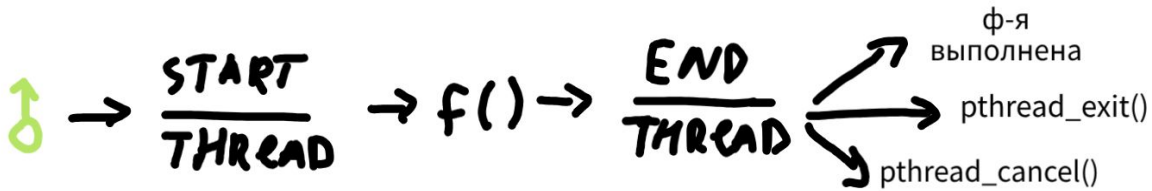
Написать функцию, которая создает новый процесс. Дочерний процесс спит 5 секунд, затем отправляет сигнал SIGUSR1 родительскому процессу, после чего завершает работу. Родительский процесс должен обработать сигнал и вывести сообщение.

```
// Обработчик сигнала
void signal_handler(int signum) {std::cout << "Received signal: " << signum <<
std::endl;}
void send_signal_to_parent () {
  pid_t pid = fork();
  if (pid < 0) {
      exit(1);
  } else if (pid == 0) {
      // child
      sleep(5);
      kill(getppid(), SIGUSR1);
      exit(0);
  } else {
      // parent
      signal(SIGUSR1, signal_handler);
      // Ожидаем завершения дочернего процесса
      wait(NULL);
  }
}
```

Потоки.



Поток — наименьшая единица выполнения в процессе.
Поток выполнения – код для выполнения. Процесс содержит минимум 1 поток выполнения, начинающий выполнение кода программы. Потоки создаются функцией **pthread_create**(). И запускаются сразу после создания.

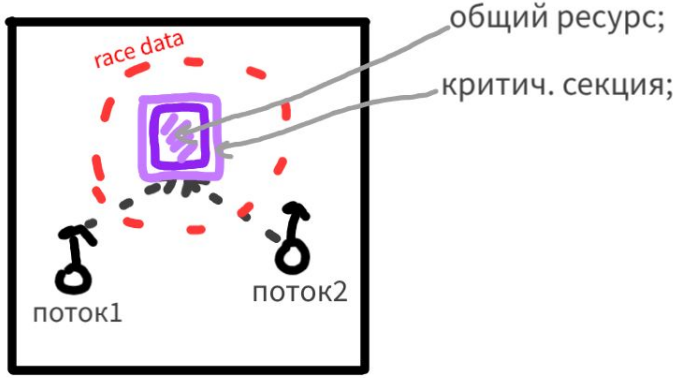


Поток завершен успешно в первых 2 случаях, поскольку у него будет возвращаемое значение. необработанное исключение в любом из потоков приведет к завершению всего процесса.

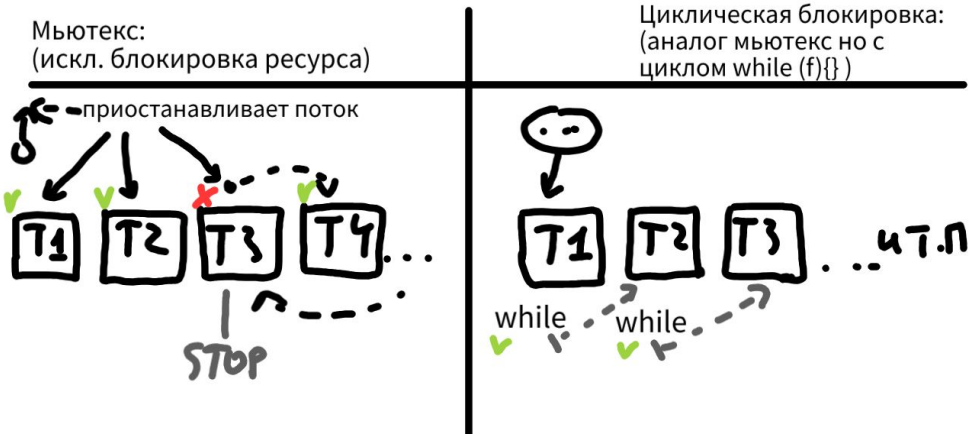
Ожидание завершения конкретного потока - **pthread_join**() - ф-я блокирует вызывающий поток до тех пор, пока целевой поток не завершиться;

Посылка сигналов между потоками 1го процесса выполняется ф-ми **pthread_kill**()/**pthread_sigqueue**().

Общий ресурс;



Общий ресурс — это ресурс, доступный нескольким потокам или сигналам одновременно. для предотвращения ошибок, таких как **гонки данных** (*конфликт одновременного исп-я*), исп-ся примитивы синхронизации. **Критические секции** - участок кода, с доступом к общему ресурсу. **Примитивы синхронизации** – объекты, исп-е для синхронизации. **Мьютекс** — исключительная блокировка ресурса. **Циклическая блокировка** — аналог мьютекса, но с while;



Пример на потоки;

Написать функцию, которая создает два потока. Первый поток принимает строку и выводит её, второй поток принимает целое число, засыпает на указанное количество секунд, затем завершается.

```
void* f1(void* arg) {
  char* message = static_cast<char*>(arg);
  std::cout << message << std::endl;
  pthread_exit(nullptr);
}


void* f2(void* arg) {
  int* seconds = static_cast<int*>(arg);
  sleep(*seconds);
  std::cout << "Slept for " << *seconds << " seconds" << std::endl;
  pthread_exit(nullptr);
}


void create_threads_with_args() {
  pthread_t thread1, thread2;
  const char* message = "text..";
  int sleepTime = 5;
  if (pthread_create(&thread1, nullptr, f1, (void*)message) != 0) return;
  if (pthread_create(&thread2, nullptr, f2, (void*)&sleepTime) != 0) return;
  if (pthread_join(thread1, nullptr) != 0) return;
  if (pthread_join(thread2, nullptr) != 0) return;
}
```