Лекция 1

IPC

1.1 Общее

- Есть много разных **IPC** — Inter Process Communication
- Examples: **pipes in shell**, **sockets**, **System V shared memory**, **signals**, **mutexes**
- **IPC** дает какой-то способ взаимодействия
- **IPC** нужно выбирать с умом, зная требования к взаимодействию
- Демоны (**daemons**) — служебные процессы, которые долго живут и не перезагружаются (обслуживают что-то)

TODO Две картинки-схемы + классификация из презентации

1.2 Сигналы

1.2.1 Общее

- Сигналы характеризуются числом
- **$ kill -SIGSTOP [number of proccess]** — послать сигнал процессу
- Сигналы выставляются процессу

Примеры:

- **SIGSEGV** — segmentation violation
- **SIGBUS** — генерируется в связи с проблемами маппинга виртуальной памяти на диск TODO Is it true?
- **SIGINT** — interrupt (Ctrl + C)
- **SIGILL** — illegal instruction
- **SIGUSR1, SIGUSR2** — отдаются на использование программисту
- **SIGSTOP** — процесс перестает шедулироваться (грубо говоря замораживается)
- **SIGCONT** — процесс начинает шедулироваться TODO Is it true?
- **SIGTERM** — попросить процесс завершиться

SIGKILL и **SIGSTOP** — нельзя ни перехватить, ни игнорировать

1.2.2 Пример №1

Что произойдет?

signal1.cpp

```
#include <assert.h>

int main(int argc, const char *argv[]) {
    assert(0);
}
```

У сигнала может быть три разных поведения: игнорирование, дефолтное, свой обработчик

1.2.3 Пример №2

Как послать сигнал самому себе?

signal2.cpp

```
#include <signal.h>

int main(int argc, const char *argv[]) {
    raise(SIGNAL);
}
```

1.2.4 Пример №3

Пишем свой обработчик сигнала

signal3.cpp

```
#include <signal.h>
#include <unistd.h>

static void sigint_handler(int signo) {
    printf("SIGINT caught\n");
}

int main() {
    signal(SIGINT, sigint_handler);

    for(;;) {
        sleep(1);
    }
}
```

Нажимаем Ctrl + C, ловим сигнал

1.2.5 Пример №4

Сигнал обрабатывается по границе выполняемой инструкции

signal4.cpp

```
#include <signal.h>
#include <unistd.h>

static void sigint_handler(int signo) {
    printf("SIGINT caught\n");
}

int main() {
    signal(SIGINT, sigint_handler);

    for(;;) {
        sleep(10000);
        printf("hm\n");
    }
}
```

Генерация сигнала прерывает функцию *sleep()*

1.2.6 Пример №5

signal5.cpp

```
#include <signal.h>
#include <unistd.h>

static unsigned counter = 0;

static void sigint_handler(int signo) {
    counter += 100500;
    printf("SIGINT caught\n");
}

static void non_reentrant_func() {
    for (;;) {
        const unsigned prev_counter = counter;
        ++counter;
        assert(prev_counter + 1 == counter);
    }
}

int main() {
    signal(SIGINT, sigint_handler);
    non_reentrant_func();
}
```

Мы не имеем права звать из обработчика сигналов нереентерабельные функции (*malloc()*, *printf()*, ...)

1.2.7 Пример №6

signal6.cpp

```
#include <signal.h>
#include <unistd.h>

static sig_atomic_t sigint_flag;

static void sigint_handler(int signo) {
    sigint_flag = 1;
}

int main() {
    signal(SIGINT, sigint_handler);

    for(;;) {
        if (sigint_flag) {
            printf("SIGINT caught\n");
            sigint_flag = 0;
        }
        sleep(1);
    }
}
```

sig_atomic_t — define для TODO ?

1.2.8 Пример №7

signal7.cpp

```
#include <signal.h>
#include <unistd.h>

static sig_atomic_t sigint_flag;

static void sigint_handler(int signo) {
    if (signo == SIGINT) {
        sigint_flag = 1;
        printf("SIGINT caught\n");
        raise(SIGUSR1);
        printf("SIGUSR1 sent\n");
    } else {
        assert(signo == SIGUSR1);
        printf("SIGUSR1 caught\n");
    }
```

```
}

int main() {
    signal(SIGINT, sig_handler);
    signal(SIGUSR1, sig_handler);

    for(;;) {
        if (sigint_flag) {
            printf("SIGINT caught\n");
            sigint_flag = 0;
        }
        sleep(1);
    }
}
```

Если сигнал возникнет в обработчике сигнала, то он обработается

1.2.9 Пример №8

TODO Пример с ассемблером

1.2.10 Дополнительно

- Можно взять обработчик сигнала для **SIGIGN**, и поставить его также на обработку какого-нибудь другого
- У интерфейса сигналов много проблем, поэтому появился *advanced* интерфейс **$ man sigaction**
  Можно доставать из него информацию о проблеме (например, для **SIGSEGV** — адрес памяти, которая защищена)
- TODO Гадание по **CR2** как в Матрице
- Если сигнал возник во время системного вызова, то он возвращается с кодом ошибки **EINTR**
  SA_RESTART — чтобы продолжить

1.3 Pipes

- Примитив **IPC**
- Данные на одном конце получаются ровно в том порядке, в котором они передаются с другого конца
- *pipe()* — системный вызов для создания
  $ man pipe
- *dup* — создание копии файлового дескриптора
  $ man dup
- **pipe** == byte stream buffer in kernel
- Гарантия атомарности упирается в константу (размер буфера)
- Механизм **IPC** не дает гарантий границ сообщений
- Если никто не пишет в **pipe**, то тот, кто читает из него — блокируется (поток исполнения)
- Есть неблокирующие файловые дескрипторы (вместо блокировки возвращают код ошибки)
- **Globbing** — подмножество регулярок (по маске получает что-то из файловой системы)

1.3.1 Buffer

- *stdout* — буферизированный, *stderr* — нет
- Буферизация — в данном случае, исключительно свойство библиотеки **libc** (системные вызовы не такие)

Пример №1

pipe1.cpp

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

int main() {
    for (int i = 0; i < 100500; ++i) {
        std::cout << 'X';
        usleep(1E5);
    }
}
```

Программа пишет, но ничего не выводится

Пример №2

pipe2.cpp

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

int main() {
    for (int i = 0; i < 100500; ++i) {
        std::cout << 'X' << sdt::endl;
        usleep(1E5);
    }
}
```

std::endl flushs output

Пример №3

pipe3.cpp

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

int main() {
    for (int i = 0; i < 100500; ++i) {
        std::cerr << 'X';
        usleep(1E5);
    }
}
```

Не буферизированный

1.3.2 Redirect

- Редиректы позволяют рассортировать вывод (**$ ./a.out 2>err**)
- **Shell** обрабатывает перенаправления последовательно
  - **$ ./a.out 1>out 2>&1** (В конце перенаправление обоих в файл)
  - **$ ./a.out 2>&1 1>out** (Сначала и первый, и второй указывают на терминал, позже первый указывает на файл, в конце *stderr* привязан к терминалу, а *stdout* к файлу)
- Еще бывают перенаправления ввода и вывода (**$ wc -l < /etc/passwd**)
- Вся информация — **$ man sh**
- **$ man 3 open**
- **$ cat /etc/passwd | tee /tmp/out** — одновременно выводим на экран и в файл

Пример

pipe4.cpp

```
#include <iostream>

int main() {
    for (int i = 0; i < 100500; ++i) {
        std::cout << 'X';
    }
    for (int i = 0; i < 100500; ++i) {
        std::cerr << 'Y';
    }
}
```

Выводы склеены

1.4 FIFO

Именованный **pipe** — **$ mkfifo**

1.5 SystemV

TODO

1.6 Sockets

TODO

1.7 Литература

- The Linux Programming Interface (практически все описано)
- Ссылка на презентацию от автора книги