Лекция 2

Повторение

- Задачи операционной системы
  - Предоставить общий интерфейс для работы с широким спектром различных внешних устройств
  - Распределять ресурсы компьютерной системы
- Процесс — программа в состоянии выполнения
- Процесс — основное «действующее лицо» (актор, субъект) в компьютерной системе

Надежность ОС

- Надежность ОС не должна зависить от надежности процессов в ОС, то есть ошибка в программе не должна приводить к ошибке работы компьютерной системы, блокировке других процессов или потере данных
- ОС должна изолировать процессы друг от друга
- Виртуализация — предоставление каждому процессу идеализированной (виртуальной) модели КС — способ изоляции

Память

- Память процессов должна быть изолирована друг от друга и от памяти ОС — механизм **защиты памяти**
- Обращение в чужую память считается ошибкой и приводит к принудительному завершению процесса
- **Виртуальная память** позволяет предоставить каждому процессу иллюзию, что он единственный размещен в памяти

Процессорное время

- Процесс не должен иметь возможность бесконтрольно использовать процессорное время
- Требуется внешний (по отношению к процессу) ограничитель на процессорное время — таймер и **прерывание по таймеру**
- Регулярные прерывания по таймеру полезны для ОС для разных «дел по хозяйству»

Контроль за ограничениями

- Ограничения на процесс не имею смысла, если процесс может их обходить — процесс не должен иметь право менять важные «настройки» процессора
- Инструкции процесса, позволяющие изменить настройки процессора, не должны быть доступны обычным процессам
- Требуются **привилегированные инструкции**
- (по крайней мере) два режима работы процессора: **пользовательский** (user mode) и **привилегированный** (kernel mode)

Прерывания

- Выполняющийся процесс не должен помешать операционной системе обработать событие от внешнего устройства
- В случае ошибки в работе процесса управление должно перехватываться операционной системой
- Требуются **прерывания**

Аппаратные требования

- Защита памяти
- Привилегированный режим работы процессора
- Прерывания
- Таймер

Ядро

- Ядро (kernel)
  - Главный компонент операционной системы
  - Загружается в память в момент старта операционной системы (загрузка компьютера)
  - Находится в оперативной памяти все время работы компьютера
  - Работает в привилегированном режиме работы процессора

Пользовательский процесс

- Работает в пользовательском режиме процессора (даже процессы пользователя Administrator или root!)
- Не имеет прямого доступа к аппаратуре (регистрам ввода-вывода, регистрам, отображенным в память и т. п.)
- Сам по себе процесс «слеп», «глух», «нем»: не может сделать ничего полезного

Системный вызов (syscall)

- Специальная инструкция процессора, переключающая выполнение на фиксированную точку входа в ядре в привилегированном режиме
- Запрос со стороны пользовательского процесса определенного сервиса ядра операционной системы.
- Linux 4.1.6 — 359 системных вызовов

Выполнение системного вызова

- Процесс подготавливает параметры (Linux x86: записывает на регистры процессора)
- Выполняется инструкция системного вызова (Linux x86: int 0x80)
- Процессор переключается в защищенный режим, управление передается в точку входа ядра (Linux x86: arch/x86/kernel/entry_32.S: system_call)
- Сохраняется текущий контекст выполнения (реализовано на ассемблере)
- По номеру системного вызова (Linux x86: eax, массив указателей ia32_sys_call_table) вызывается соответствующая функция ядра (например, sys_open в fs/open.c) — все реализации системных вызовов на Си

Выполнение системного вызова (2)

- Если нужно считать данные из памяти процесса, используется специальная функция ядра copy_from_user, для записи — copy_to_user
- Как правило, функции обработки системных вызовов возвращают значение типа int. Если оно отрицательно — значит системный вызов неудачен.

```
return -EPERM;
```

- Ассемблерный код в entry_32.S восстанавливает контекст выполнения с новым значением регистра eax — значение, которое было возвращено из обработчика системного вызова. Последняя инструкция — IRET.
- Процессор переключается обратно в пользовательский режим и продолжает выполнение программы с инструкции непосредственно после инструкции системного вызова.

Интерфейс POSIX

- Интерфейс POSIX определен в терминах языка Си
  - Где требуются строки (например, пути к файлам) должны передаваться char* строки (\0-terminated). Кодировка — байтовая (обычно UTF-8).
  - Где требуются буфера для ввода/вывода должны передаваться указатели на область памяти (void*) правильного размера

C++ → C

- Чтобы получить const char* из std::string используется метод c_str()

  ```
  std::string path;
  int fd = open(path.c_str(), O_RDONLY, 0);
  ```

- Чтобы получить адрес буфера из std::vector нужно взять адрес нулевого элемента:

  ```
  std::vector<char> buf(1024);
  rsz = read(fd, &buf[0], buf.size());
  ```

Интерфейс POSIX

- POSIX опирается на Си, необходим вспомогательный код, согласовывающий передачу параметров в стиле Си и передачу параметров в стиле соотв. ядра ОС
- Для Linux x86:
  - Скопировать параметры из стека в регистры
  - Если возвращенное из системного вызова значение отрицательное, скопировать его в errno
- Такие функции находятся в стандартной библиотеке и называются по имени системного вызова, они для удобства тоже называются «системными вызовами».

Интерфейс POSIX

- int open(const char *path, int flags, int mode);

```
open:
    pushl %ebp
    movl %esp, %ebp
    movl $__NR_open, %eax
    movl 8(%ebp), %ebx
    movl 12(%ebp), %ecx
    movl 16(%ebp), %edx
    int 0x80
    tstl %eax
    jge done
    movl %eax, __errno
    movl $-1, %eax
done:
    leave
    ret
```

Интерфейс POSIX

- POSIX определяет много Си функций и их семантику
- Эти функции находятся в стандартной библиотеке Си libc (/lib/libc.so.6 или /usr/lib/libc.a).
- Не все такие функции напрямую соответствуют системным вызовам в ядро.
  - creat может вызывать open
  - fork может вызывать clone
- В Linux детали различны для разных архитектур

POSIX обработка ошибок

- #include <errno.h>
- Если системный вызов завершается с ошибкой, как правило, возвращается -1, код ошибки сохраняется в «переменной» errno
- Коды ошибок находятся в errno.h, например, EPERM, EAGAIN, ENOENT
- Документация на системный вызов описывает какая ошибка возвращается в какой ситуации
- #include <string.h>
  char *strerror(int err);
- Позволяет получить текстовое сообщение об ошибке

Взаимодействие со средой

- Процесс завершается системным вызовом _exit(exitcode)
- Или возвращаемое значение return из main
- Значение в диапазоне [0;127] — код завершения процесса, он доступен процессу-родителю
- Код 0 — успешное завершение (/bin/true)
- Ненулевой код — ошибка (/bin/false)

Аргументы командной строки

- Функция main получает аргументы командной строки:

  int main(int argc, char *argv[])

- argv — массив указателей на строки Си

  ./prog foo 1 bar

  argv[0] → "./prog"; путь к программе
  argv[1] → "foo";
  argv[2] → "1";
  argv[3] → "bar";
  argv[4] → nullptr;

- Передаются на стеке процесса

Переменные окружения

- Именованные значения доступные процессу
- По умолчанию передаются неизменными порождаемым процессам

```
char *getenv(const char *name);
```

```
const char *h = getenv("HOME");
std::string path;
if (h) {
    path.assign(h);
    path += "/.myprops";
}
```