Лекция 3

2 апреля 2025 г.

Содержание

1 Непосредственное программирование

В первые времена работы с компьютерами не существовало понятия операционной системы. Программы загружались напрямую, без возможности хранения кода внутри машины. Обычно для каждой задачи:

- вручную устанавливали программу (иногда даже с помощью рычажков);
- подавали данные с носителей;
- ожидали окончания исполнения;
- затем повторяли процесс с новой задачей.

2 Пакетное планирование

Когда компьютеры начали обслуживать несколько задач, возникла идея пакетного выполнения. Это означает:

- множество задач заранее загружаются в систему;
- задачи исполняются последовательно одна за другой;
- ресурсы компьютера используются эффективно, так как нет простоя.

Однако в каждый момент времени работает только одна задача, а переключения между ними нет.

3 Таймшеринг (разделение времени)

С появлением мощных машин (60-е года) стало очевидно, что человек стал узким местом — он не успевает готовить новые задачи. Возникла идея параллельной (логической) работы с несколькими задачами. В этом режиме:

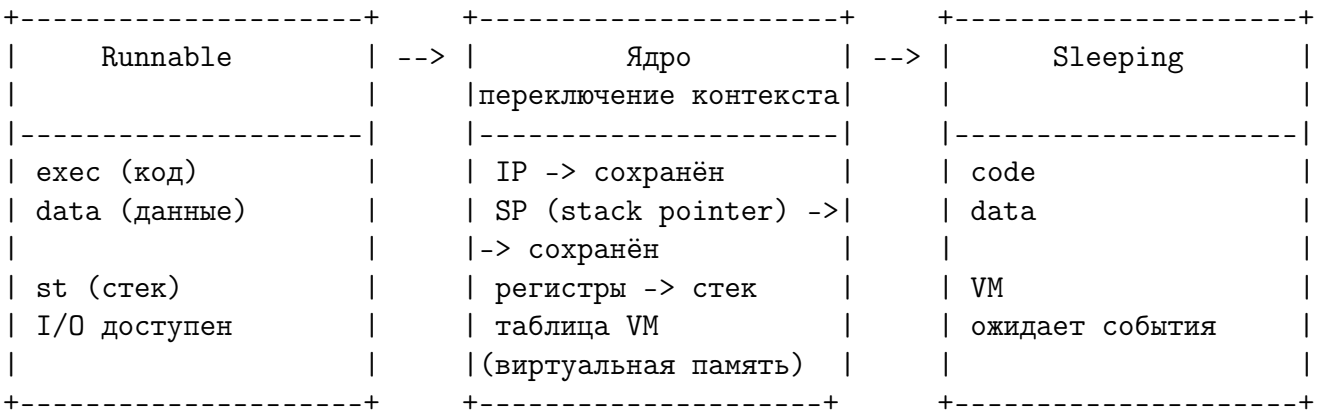
- каждому пользователю выделяется доля времени на CPU;
- создаётся иллюзия одновременного исполнения;
- появляется необходимость управления многими программами.

4 Планирование по времени и кванты

Чтобы несколько процессов могли работать одновременно, необходимо:

- разделить время на небольшие интервалы *(кванты)*;
- по истечении кванта — переключаться на другой процесс;
- сохранять состояние текущего процесса и загружать состояние следующего.

4.1 Структура процесса и переключение контекста



Пояснение: Слева изображён активный процесс, готовый к запуску (Runnable). При наступлении события (например, по таймеру), ядро сохраняет его состояние: инструкции, стек и регистры. Это и есть *переключение контекста*. Затем другой процесс, который находился в состоянии ожидания (Sleeping), может быть активирован. Его данные, код и стек уже существуют в памяти. Instruction Pointer (IP) указывает на код, который должен начать исполняться.

———————

Памятка для понимания::

- **Running** — процесс активно выполняется на CPU.
- **Runnable** — готов к исполнению, но не исполняется в данный момент.
- **Sleeping** — ожидает какого-либо события: ввода, таймера и т.д.

Цикл исполнения:

- системный таймер вызывает прерывание;
- состояние CPU сохраняется в стек ядра;
- операционная система может:
  - либо продолжить текущий процесс;
  - либо переключиться на другой.

5 Программный и прямой доступ к памяти

PIO (Programmed I/O):

- CPU сам опрашивает устройство;
- возможны прерывания от устройства для сигнализации готовности.

DMA (Direct Memory Access):

- устройство само инициирует запись в память;

- CPU не участвует в передаче данных;
- после завершения передачи — генерируется прерывание CPU.

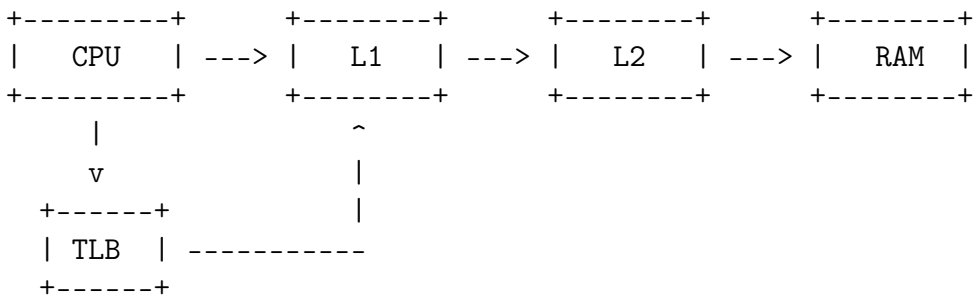
Сравнение PIO и DMA:

Метод	Кто управляет	Особенности
PIO	CPU	Постоянный опрос, высокая загрузка CPU
DMA	Устройство	CPU отдыхает, используется прерывание

6 Кэш и отображение памяти

Когда процесс обращается к данным, используется система виртуальной памяти. Это требует трансляции виртуальных адресов в физические. Для ускорения работы существует иерархия кэшей и TLB.

6.1 Кэш и виртуальная/физическая память



Объяснение:

- CPU выполняет команды и обращается к памяти через виртуальные адреса.
- TLB (Translation Lookaside Buffer) — кэш таблицы страниц, хранящий трансляции из виртуальных адресов в физические.
- Если трансляция найдена в TLB, физический адрес сразу поступает в кэш.
- Если нет — процессор обращается к таблице страниц, и TLB обновляется.
- L1 — кэш первого уровня. Может использовать виртуальные или физические адреса (в зависимости от архитектуры).
- L2 — общий кэш для нескольких ядер, чаще использует физические адреса.
- RAM — последняя точка доступа при промахе в кэше.

Проблема переключения контекста (Context Switch):

- При смене процесса адресное пространство меняется.
- Трансляции из старого процесса больше не актуальны.

- TLB необходимо *инвалидировать* — очистить, чтобы не использовать старые отображения.

Аргументы в пользу использования виртуальных адресов в кэше:

- быстрее — можно искать данные сразу после получения виртуального адреса, не дожидаясь трансляции;
- применимо в L1, где важна минимальная задержка.

Аргументы в пользу физического кэширования:

- не нужно инвалидировать кэш при context switch;
- используется в L2 и выше, где больше данных и важна универсальность.

Типичная архитектура:

- L1 — виртуальный или физический (зависит от реализации);
- L2, L3 — физические адреса;
- TLB — отдельный кэш, работает *до* обращения к кэшу данных.

7 Монолитное и микроядро

Монолитное ядро:

- Вся логика ядра (драйверы, планировщик, файловая система) — в одном адресном пространстве.
- Если в одном компоненте ошибка (например, в драйвере файловой системы), может быть повреждена вся система.
- Отсутствуют уровни изоляции между подсистемами ядра.

Микроядро:

- Только минимальная логика (планирование, изоляция, IPC) находится в самом ядре.
- Остальные части (драйверы, файловая система, сеть) вынесены в отдельные процессы, изолированы.
- Повышает безопасность: уязвимость в одном компоненте не даёт доступа к остальным.
- Минус — переключения между компонентами требуют дополнительных context switch и передач сообщений.

8 Учебное ядро YABLOKO (на лекции 10 минут посвещено, поэтому информации мало)

YABLOKO — это минималистичное учебное ядро, разработанное для обучения основам работы ОС, загрузке программ, управлению памятью и взаимодействию с устройствами.

8.1 Загрузка и поведение ядра

- Точка входа: функция `kmain()`.
- В функции `kmain` происходит настройка:
  - Выделение диапазона физической памяти с помощью `freerange`:

```
freerange(P2V(1<<20), P2V(2<<20)); // 1MB - 2MB
kvmalloc();
freerange(P2V(2<<20), P2V(PHYSTOP));
```

  - Настройка GDT, PIT, клавиатуры, IDT:

```
load_gdt();
init_keyboard();
init_pit();
uartinit();
load_idt();
sti();
```

  - Очистка экрана и печать приветствия:

```
vga_clear_screen();
printk("YABLOKO\n");
```

- В терминале можно запускать команды:

```
> run greet
Hello world!
Hello world!
* success

> run div0
Exception: Division By Zero
```

8.2 Настройка виртуальной памяти

- Имеется 128MB физической памяти.
- Виртуальное адресное пространство — 4GB (32-битная архитектура).
- Отображение: 128MB физической памяти отображается на виртуальные адреса начиная с KERNBASE (3GB).
- Это делает физическую память доступной ядру по смещению от KERNBASE.
- Отображение реализуется в функции:

```
void setupkvm(void) {
    pde_t *kvm = kalloc();
    memset(kvm, 0, PGSIZE);
```

```
    for (uintptr_t pa = 0; pa < PHYSTOP; pa += 4 << 20) {
        uintptr_t va = KERNBASE + pa;
        kvm[PDX(va)] = pa | PTE_P | PTE_W | PTE_PS;
    }
    return kvm;
}
```

- Таким образом ядро получает доступ ко всей физической памяти, не создавая дополнительные уровни таблиц страниц.

8.3 Загрузка пользовательской программы

- Чтение файла с диска по имени.
- Выделение физической памяти и отображение её в виртуальную память ниже 3GB:

```
allocuvm(pgdir, USER_BASE, USER_BASE + statbuf.size);
allocuvm(pgdir, USER_STACK_BASE - 2 * PGSIZE, USER_STACK_BASE);
```

- Чтение ELF-файла в память:

```
if (read_file(&statbuf, (void*)USER_BASE, 100 << 20) <= 0) {
    printk("file not found\n");
    return;
}
```

- Настройка регистров и стека:

```
registers_t *tf = &u->trapframe;
tf->eip = hdr->e_entry; // точка входа
...
tf->useresp = USER_STACK_BASE;
```

- Переключение в пользовательский процесс:

```
switch(&vm.kernel_thread, &u->context);
```

8.3.1 Схема отображения памяти

Виртуальное пространство (4GB):

```
+----------------------------+ 0xFFFFFFFF
|      Kernel space          |
|  (отображение всей RAM)    |
+----------------------------+ 0xC0000000 = KERNBASE
|      User stack            |
|  (8KB = 2 страницы)        |
+----------------------------+
```

```
|  Программа (ELF)           |
+----------------------------+ 0x00400000 = USER_BASE
```

где:
- KERNBASE = 0xC0000000 (начало kernel space)
- USER_BASE = 0x00400000 (начало пользовательской программы)
- USER_STACK_BASE = 0xE0000000

Учебное ядро YABLOKO реализует минимально необходимый функционал ОС: загрузка, отображение памяти, запуск и завершение пользовательских программ, базовые исключения и прерывания.