Операционная система **Linux**:
SCHEDULER

Предисловие

На этом занятии мы рассмотрим:

- зачем нужен планировщик процессов;
- какие типы многозадачности есть;
- какие проблемы есть у планировщиков;
- планировщики, использованные в Linux;
- как работать с планировщиком.

План занятия

1. Предисловие
2. Появление планировщика
3. Типы многозадачности
4. Процессорное время как ресурс
5. Планировщики Linux
6. Итоги
7. Домашнее задание

Планировщик задач

Планировщик (шедулер, англ. scheduler) — часть операционной системы, ответственная за распределение ресурсов компьютера между рабочими юнитами (процессами, потоками, пользователями и т. д.).

На данном занятии мы рассмотрим планировщик задач, отвечающий **за ресурсы процессора.**

Появление планировщика

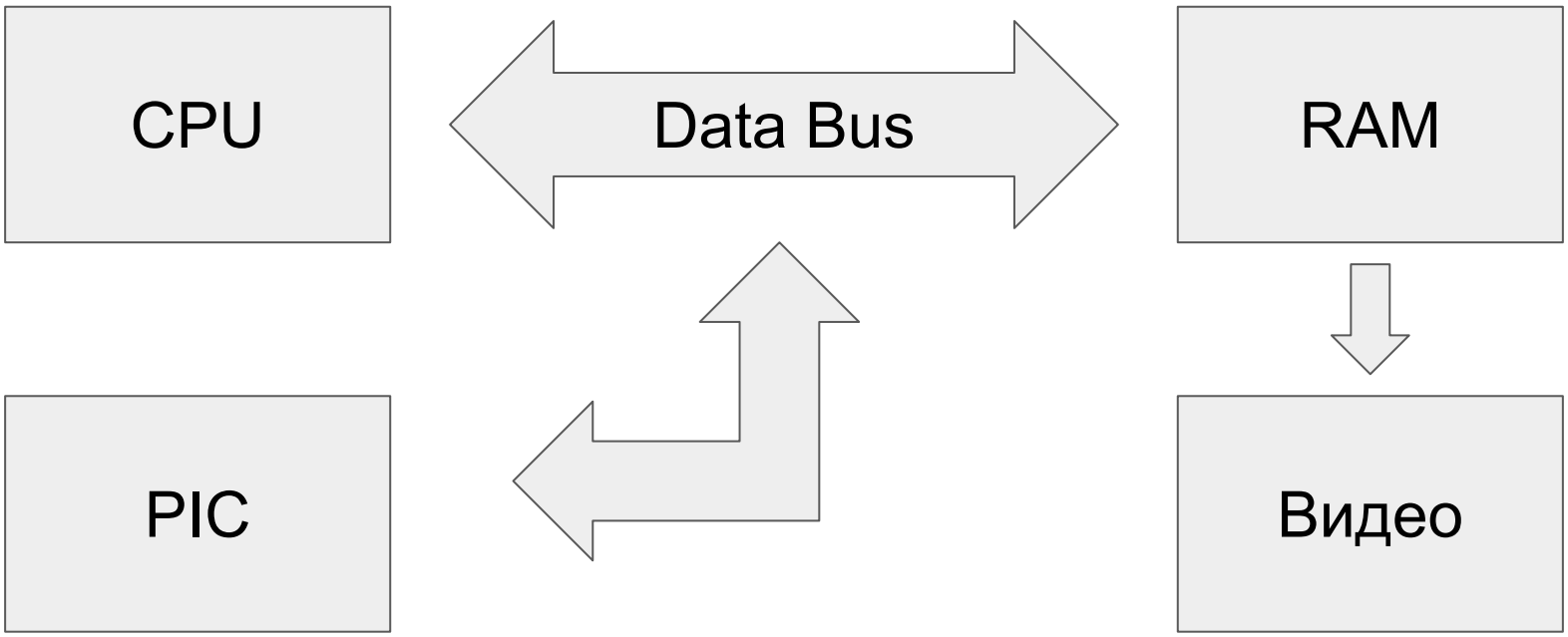
Схема работы компьютера

- **процессор** — сильно ограничен по памяти, но умеет совершать операции над этими данными;
- **память** — условно считаем, что способна вместить достаточно данных для выполнения текущих процессов;
- **шина данных** — последовательно передаёт данные между процессором и памятью.

Проблемы прошлых систем

Если мы имеем «простой» вычислитель, то:

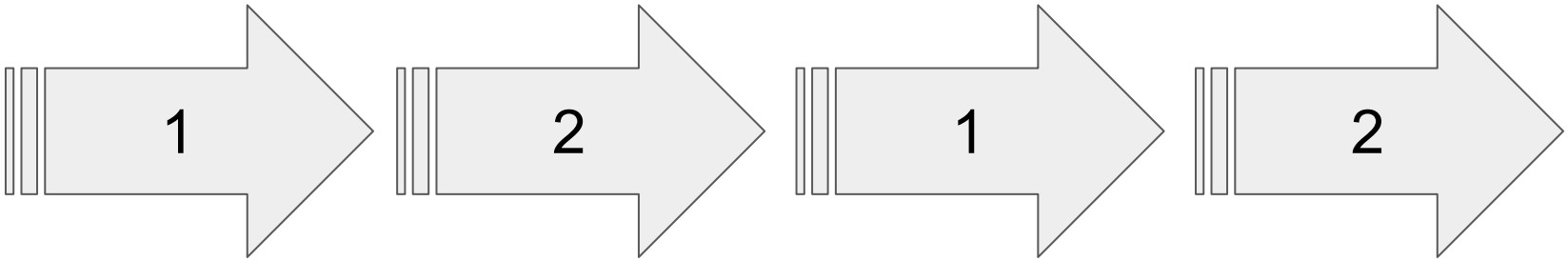
1. Проблема с обработкой внешних событий: событие ввода с клавиатуры, прихода пакета из сети, изменение изображения на мониторе и т.д.



Проблемы «прошлых» систем

Если мы имеем «простой» вычислитель, то:

1. Проблема с обработкой внешних событий: событие ввода с клавиатуры, прихода пакета из сети, изменение изображения на мониторе и т.д.
2. Проблемы с одновременной работой нескольких пользователей, нескольких задач.



Проблемы при работе нескольких процессов

- «одновременности» для массового пользователя у нас пока нет, но есть иллюзия, которую нужно поддерживать;
- нужна программа для управления программами;
- планировщик должен быть сравнительно «честным», но не всегда, не совсеми, а даже с ними непостоянно;
- планировщик должен быть «реальным».

Реальность планировщика

- невозможно узнать, когда завершится программа;

  В рамках Машины Тьюринга невозможно определить, когда закончится программа (дойдёт до какого-то момента).

- программа может зависнут, а процессор не знает о времени;
- используем аппаратные прерывания.

  Программно невозможно понять, что прошло какое-то время. Остаётся использовать аппаратные прерывания. Используем аппаратные прерывания от контроллера прерываний периферии, в частности — timer.

Пример мультипроцессинга в DOS

Всё управление у пользователя, так как технически выполняется только одна пользовательская задача, то сохранение состояния и переключение на другую задачу выполняется также ей:

- сохраняем текущее состояние в системный стек;
- long jump на следующую задачу.

Другой способ:

- сохраняем адрес возврата;
- передаём управление резидентной программе-планировщику.

Мультипроцессинг в DOS — регистрация

Регистрация программ, так как память общая:

- объявляем отрезок адресного пространства под список процессов;
- выделяем память под программу-планировщик;
- договариваемся о делении между программами адресного пространства.

В DOS мы работаем в реальном режиме процессора: каждая программа имеет доступ ко всей памяти.

➡ Определение не на уровне ОС, а на уровне человеческих договорённостей.

Мультипроцессинг в DOS — контекст

По «предложению» переключаем на другую программу.

Аппаратной поддержки переключения контекста нет, поэтому:

- нужен IP для возврата, т.к. мы его не храним;
- программа сохраняет IP, планировщик сохраняет регистры в системной очереди;
- планировщик берёт следующую программу из списка зарегистрированных, восстанавливает её регистры, long jump к сохранённому адресу исполнения.

Мультипроцессинг в DOS — проблемы

Если что-то пошло не так — работа всего компьютера нарушена.

Так как мы используем в основном договорённости, при этом не проверяем их программно/аппаратно:

- Если забыл сохранить IP перед переключением — планировщик использует данные, которые окажутся некорректными, управление перейдёт непонятно по какому адресу.
- Если программа обращается к системному стеку глубже, чем нужно — другая программа потеряет данные.

Любой отход от договорённостей или ошибка реализации приводят к отказу всего компьютера.

Типы многозадачности

Имеем N процессов, но 1 процессор

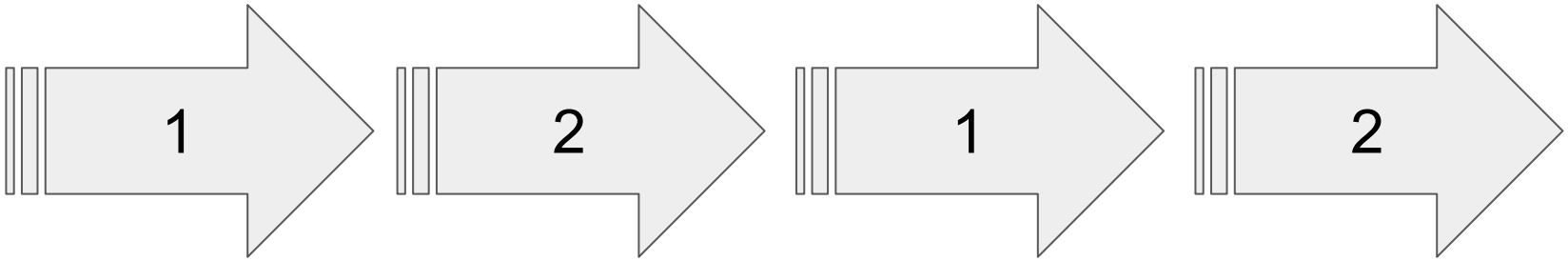
- нужно делить время работы на процессоре;

  Процент времени процесса = Приоритет ÷ Сумма приоритетов

- как справедливо поделить время на процессоре?

  Чтобы никто не забирал больше, чем надо, но получал достаточно.

- сама система распределения не должна быть дорогой.



Как распределяется время?

Существуют 2 основных подхода к обеспечению многозадачности:

- **Кооперативная многозадачность**;

  Процессы сами по доброй воле (вызовом специальной команды) отдают управление.

- **Вытесняющая многозадачность**.

  Планировщик (по таймеру) останавливает текущий процесс и передаёт управление другому.

Кооперативная многозадачность

- **простота;**

  Планировщик проще: просто список.

- **скорость;**

  Нет необходимости зря прерывать процесс (в моменте исполнения). Быстрее при IO-bound, разделяемых ресурсах.

- **низкая отзывчивость**;

  Потенциально получаем перерасход времени частью процессов.

- **зависания;**

  Если в одном из процессов ошибка/блокировка — висят все процессы.

Кооперативная многозадачность с ThreadSwitch

Как работает переключение исполняемой задачи с помощью функции ThreadSwitch():

- Текущая задача остаётся активной (готовой) и передаёт управление следующей готовой задаче.
- Для языка программирования данная функциональная возможность выглядит как обычная функция.

Примеры реализации ThreadSwitch

ThreadSwitch()

```
void ThreadSwitch () {
    old_thread = current_thread;
    add_to_queue_tail(current_thread);
    current_thread = get_from_queue_head();
    asm {
        move bx, old_thread
        push bp
        move ax, sp
        move thread_sp[bx], ax
        move bx, current_thread
        move ax, thread_sp[bx]
        pop bp
    }
    return;
}
```

Пример кооперативной многозадачности с async-await

В асинхронном программировании также используется **async-await:**

- async объявляет функцию сопрограммой;
- внутри сопрограммы возможна передача управления на другие сопрограммы с помощью await (почти как с ThreadSwitch);
- await-ить можно сопрограммы с целью получения их результата;

  Ожидать результата их исполнения: получение данных, отсылку данных, освобождения заблокированного ресурса.
- сопрограммы регистрируются на event-loop;
- event-loop запускается и поочерёдно запускает сопрограммы.

  Поочерёдно ждёт, когда очередная сопрограмма отпустит (await) управление, и передаёт следующей.

Примеры работы с async-await

async-await*

```
from asyncio import *

counter = 0

async def hello(word):
  global counter
  while counter < 10:
     print("from " + word)
     await sleep(1)
     counter += 1


loop = get_event_loop()
loop.run_until_complete(
  gather(hello("A"), hello("B"))
)
loop.close()
```

* на примере Python3

Вытесняющая многозадачность

- **прогнозируемая отзывчивость;**

  Ближе к реальности, т.к. переключение завязано на реальном таймере.

- **независимость процессов;**

  Если процесс завис, то остальные от этого меньше зависят.

- **схожая скорость число-дробилок;**

  В CPU-bound скорость не проседает.

- **дороже.**

  Излишние смены контекстов (context-switch).

Пример вытесняющей многозадачности

POSIX thread*

```
int count;

static void *hello(void* d) {
  while(count < ITERATION_LIMIT) {
     ++count;
     printf("Hello %d\n", d);
  }
  return NULL;
}

int main() {
  pthread_t thread[THREAD_COUNT];
  for (int i = 0; i < THREAD_COUNT; ++i)
     pthread_create(&thread[i], NULL, &hello, i);
  for (int i = 0; i < THREAD_COUNT; ++i)
     pthread_join(thread[i], NULL);
  return 0;
}
```

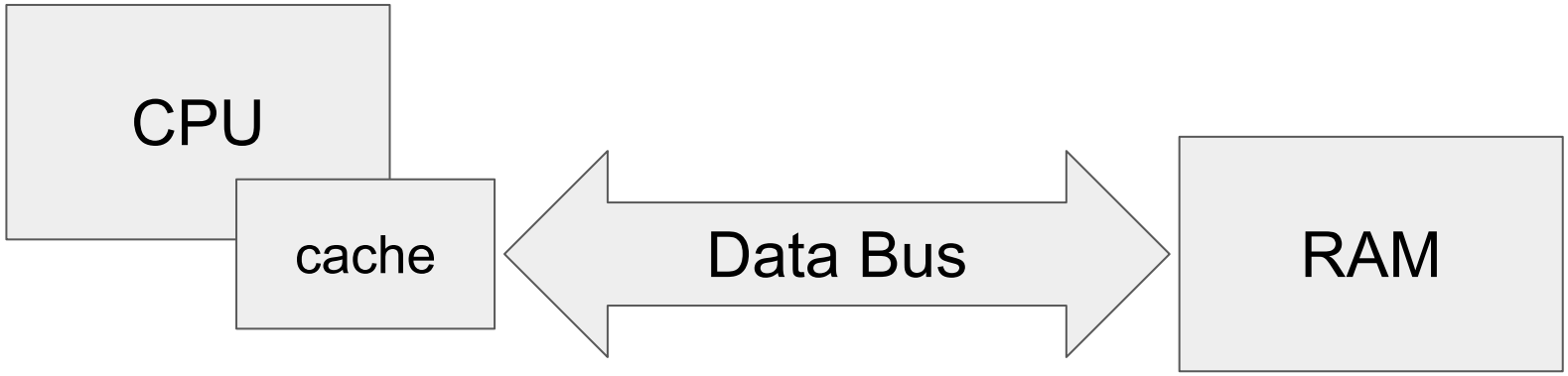
* На примере C

Процессорное время как ресурс

Дорогая смена контекста

Переключая процессы / треды, необходимо сохранять их контекст (стек, регистры и т.д.), менять дескрипторы тредов, прогревать кеш…

- cache
- kernel
- …



Пример замедления из-за кеша

```
#include "pthread.h"
#include "stdio.h"
#include "time.h"
#define THREAD_COUNT 8
#define ITERATION_LIMIT 1000000000LL

long long count = 0;

static void *hello(void* d) {
  printf("Start thread\n");
  while(count < ITERATION_LIMIT)
     ++count;
  return NULL;
}

int main() {
  pthread_t thread[THREAD_COUNT];
  for (int i = 0; i < THREAD_COUNT; ++i)
     pthread_create(&thread[i], NULL, &hello, i);
  for (int i = 0; i < THREAD_COUNT; ++i)
     pthread_join(thread[i], NULL);
  return 0;
}

// clang -lpthread example.c
```

Пример замедления из-за кеша

С помощью утилиты **taskset** можно задавать, на каких процессорах будет исполняться программа или менять по PID:

```
taskset [options] mask command [argument...]
taskset [options] -p [mask] pid
```

На одном процессоре против 8-ми:

```
$ time taskset 1 ./a.out
real    0m1,372s
user    0m1,368s
sys     0m0,005s

$ time ./a.out
real    0m5,007s
user    0m39,886s
sys     0m0,004s
```

Приоритетность процессов

Некоторые процессы могут выполняться без проблем в фоне, каким-то нужно выделить больше времени.

Для управления приоритетами в Unix-like системах используется nice:

- от -20 до 19;
- чем меньше, тем приоритетнее;
- может отличаться от реального приоритета задачи.

  Лишь просьба к операционной системе.

Как использовать nice

- nice -n {приоритет} {программа} {аргументы};
- sudo nice -n — если приоритет меньше 0;
- renice -n {приоритет} PID — для уже запущенного процесса;
- renice -n -g -u — для запущенной группы процессов, для пользователя.
- ps ax -o pid,ni,cmd — чтобы посмотреть nice запущенных процессов.

Пример работы nice

```
#!/bin/bash

nice -n 19 ./a.out >> /dev/null && echo 19 &
nice -n 18 ./a.out >> /dev/null && echo 18 &

nice -n 11 ./a.out >> /dev/null && echo 11 &
nice -n 10 ./a.out >> /dev/null && echo 10 &

nice -n 1 ./a.out >> /dev/null && echo 1 &
nice -n 0 ./a.out >> /dev/null && echo 0 &
```

```
$ ./nice.sh
$ 1
0
10
11
18
19
```

Кванты времени

- планировщик выделяет время кусками (time slice);

  В современных системах (CFS) это значение можно поменять через sysctl kernel.sched_min_granularity_ns. Если увеличиваем его, то уменьшаем перерасход ресурсов на смену контекста, но теряем отзывчивость.

- можно отдать управление из программы;

  К примеру, pthread_yield / sched_yield

- настраиваются планировщики как раз через константы, описывающие размер квантов / допустимые задержки и т.д.

Планировщики Linux

Какие планировщики есть

За время жизни ОС GNU/Linux у неё были различные реализации планировщика:

- **O(n)** — первый, самый простой;
- **O(1)** — с версии 2.6, работающий быстрее при большем объёме задач;
- **CFS** — текущий (с 2.6.23). Также ускоренный, с дополнительными возможностями.
- **SCHED_DEADLINE** — альтернативный (с 3.14) планировщик реального времени.

Старый / линейный планировщик

- чем больше задач, тем линейно больше времени требуется на планирование, отсюда и название – O(n);
- проблемы работы с несколькими процессами;

- при выборе следующей задачи пробегаем по списку задач в поисках самой недоработавшей: её выбираем, исполняем, изменяем в дескрипторе время исполнения;
- доступ к списку задач блокировался 1-м lock-ом.

O(1) планировщик

- храним процессы в отсортированном списке;
- ещё один список для процессов, отработавших ресурс времени;
- если первый список закончился, то меняем списки местами;
- на каждый процессор по своей паре этих списков (избавились от общего spinlock-а);
- появилась проблема миграции процесса с одного процессора на другой.

CFS планировщик

- храним не процессы, а sched_entity — планировщик может рассчитывать время, например, для пользователей;
- все sched_entity лежат в бинарном сбалансированном дереве, отсортированном по оставшемуся времени исполнения и имеет сложность *O(logN)*;
- CFS планировщик более справедлив — при равном приоритете задачи будут получать более равные кванты, чем в *O(1)*;
- Работает сразу на все процессоры — нет проблемы с локальными spinlock-ами.

Планировщик SCHED_DEADLINE

- используется для систем реального времени;
- основная задача — выполнить задачу за **фиксированное реальное** время (не позже);
- использует алгоритм планирования по ближайшему сроку завершения:
  - планировщик ведёт список процессов, отсортированный по сроку завершения (deadline);
  - в работу берётся готовый процесс, имеющий самый близкий deadline;
  - при появлении нового процесса — пересортировка.

Планировщик ввода / вывода

- внешняя память — также разделяемый ресурс, его также нужно планировать;
- для разных типов диска (HDD, SSD) стоит использовать разные планировщики. Посмотреть можно в /sys/block/устройство/queue/scheduler;
- долгое время использовался CFQ. Однако, лучше посмотреть тесты. Для SSD часто лучше BFQ / deadline. А то и noop.
- ionice — аналог nice для планировщика ввода / вывода.

Итоги

Итоги

Сегодня мы:

- рассмотрели основные подходы и проблемы реализации многозадачных систем;
- получили представление об утилитах работы с планировщиком GNU/Linux;
- узнали особенности реализаций планировщиков Linux.