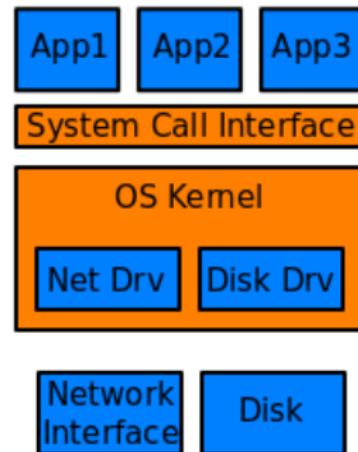


Операционные Системы

Введение

January 5, 2019

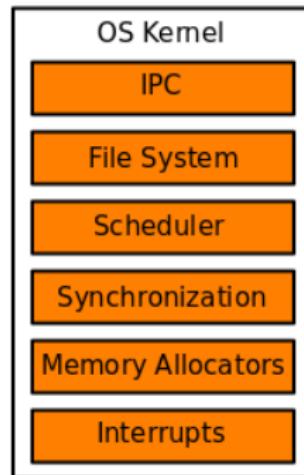
Что такое ОС?



Сервисы ОС

- ▶ Доступ к устройствам
 - ▶ например, мышь или клавиатура
 - ▶ часто через файловый интерфейс
- ▶ Файловая система
 - ▶ файлы и каталоги
- ▶ Создание и взаимодействие процессов
 - ▶ сигналы, каналы и другие
 - ▶ сетевое взаимодействие

Компоненты ОС



Операционные Системы

Язык ассемблера

January 5, 2019

Язык ассемблера для x86

- ▶ Язык ассемблера концептуально прост:
 - ▶ минимум синтаксических правил;
 - ▶ много различных инструкций (зависит от архитектуры).
- ▶ Есть много диалектов:
 - ▶ будем использовать GNU, а. к. а. AT&T.

Классы инструкций

- ▶ Инструкции копирования:
 - ▶ из памяти в регистр и назад;
 - ▶ из регистра в регистр;
 - ▶ реже из памяти в память.
- ▶ Арифметические инструкции.
- ▶ Инструкции перехода:
 - ▶ условного перехода и безусловного.
- ▶ Прочие инструкции.

Регистры

- ▶ Регистры - очень быстрые именованные ячейки памяти.
- ▶ Регистры специального назначения
 - ▶ указатель команд;
 - ▶ флаговый регистр;
 - ▶ и много много других.
- ▶ Регистры общего назначения.

Регистры x86

- ▶ Указатель команд - *RIP*.
- ▶ Флаговый регистр - *RFLAGS*.
- ▶ Регистры общего назначения:
 - ▶ указатель стека - *RSP*;
 - ▶ указатель "базы" - *RBP*;
 - ▶ *RAX*, *RBX*, *RCX*,
RDX, *RSI*, *RDI*,
R8 - *R15*.

Инструкция копирования MOV

- ▶ movq <src>, <dst>
 - ▶ movq %RAX, %RBX
 - ▶ movq (%RAX), %RAX
 - ▶ movq \$42, %RAX
 - ▶ movq 42, %RAX
 - ▶ movq \$value, %RAX
 - ▶ movq value, %RAX

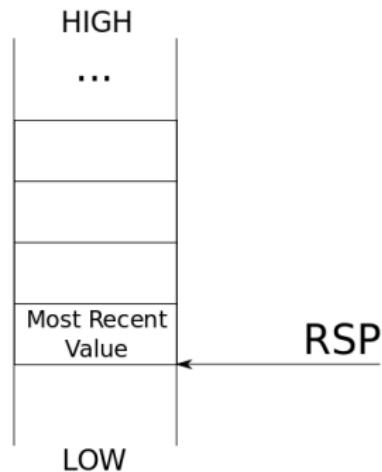
Простые арифметические инструкции

- ▶ addq <src>, <dst>
 - ▶ addq %RAX, %RBX
 - ▶ addq %RAX, value
 - ▶ addq \$42, %RAX
- ▶ sub <src>, <dst>
- ▶ incq <op>
 - ▶ incq %RAX
- ▶ decq <op>

Инструкции беззнакового умножения и деления

- ▶ `mulq <op>`:
 - ▶ $RAX = (< op > \times RAX) \bmod 2^{64}$
 - ▶ $RDX = (< op > \times RAX) / 2^{64}$
- ▶ `divq <op>`:
 - ▶ $RDX = (RDX \times 2^{64} + RAX) \bmod < op >$
 - ▶ $RAX = (RDX \times 2^{64} + RAX) / < op >$

Стек

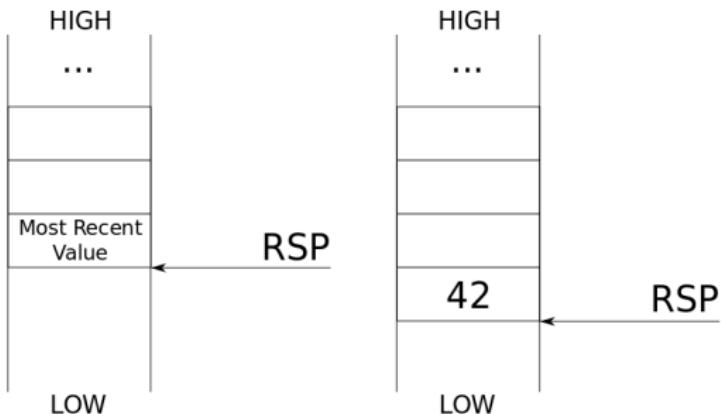


Инструкции работы со стеком

- ▶ `pushq <src>` - уменьшает *RSP* на 8 и сохраняет по полученному адресу *src*
 - ▶ `pushq $42`
 - ▶ `pushq %RAX`

Инструкции работы со стеком

PUSHQ \$42

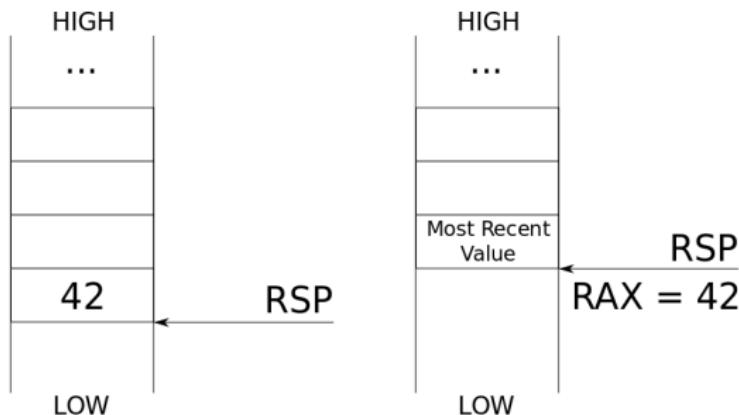


Инструкции работы со стеком

- ▶ `popq <dst>` - обратное действие к *pushq*
 - ▶ `popq %RAX`
- ▶ `movq (%RSP), %RAX`

Инструкции работы со стеком

POPQ %RAX



Метки и переменные

- Метка - просто имя для некоторого адреса:

```
1          .data
2 value:
3          .quad 42
4
5          .text
6 add42:
7          movq %rdi, %rax
8          addq value, %rax
9          retq
```

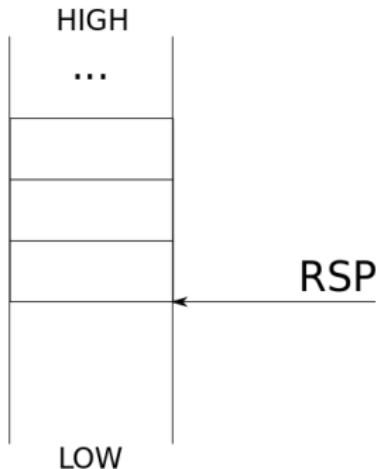
Инструкции безусловного перехода

- ▶ Инструкции безусловного перехода изменяют значение регистра *RIP*:
 - ▶ `jmp <label>`
 - ▶ `call <label>` - инструкция вызова функции
 - ▶ `retq` - инструкция возврата из функции

Функции

- ▶ Функция:
 - ▶ функцию можно вызвать;
 - ▶ функция возвращает управление *вызывавшему коду*.

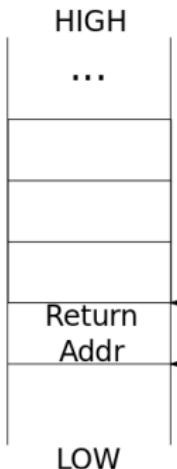
Вызов функции



main:

```
'movq $42, %rsi'  
'call add42'  
retq
```

Вызов функции



main:

```
    movq $42, %rsi
    call add42
    retq
```

Флаговый регистр RFLAGS

- ▶ Флаговый регистр хранит флаги!
- ▶ Флаги регистра RFLAGS:
 - ▶ ZF - результат операции 0;
 - ▶ CF - произошло беззнаковое переполнение;
 - ▶ OF - произошло знаковое переполнение.

Инструкции условного перехода

- ▶ `jcc <label>` - выполняет переход, если условие *cc* истинно.
 - ▶ `jz, je` - проверяет, что $ZF = 1$;
 - ▶ `jne, jnz` - $ZF = 0$;
 - ▶ `jg` - если "больше" (знаковый вариант);
 - ▶ `jge` - "больше или равно" (знаковый вариант);
 - ▶ `ja` - если "больше" (беззнаковый вариант);
 - ▶ `jae` - "больше или равно" (беззнаковый вариант).

Инструкции сравнения

- ▶ Арифметические инструкции изменяют *RFLAGS*, но также изменяют свои аргументы!
- ▶ Есть команды, которые выставляют флаги, но не изменяют свои аргументы:
 - ▶ *cmpq <src>, <dst>* - вычисляет разность *dst – src* и выставляет флаги;
 - ▶ т. е. *cmpq* работает как *subq*, но не изменяет *dst*.

Пример ветвления

```
1 max:  
2     movq %rdi, %rax  
3     cmpq %rsi, %rdi # rdi - rsi  
4     ja rdi_gt      # rdi - rsi > 0  
5     movq %rsi, %rax  
6 rdi_gt:  
7     ret
```

Соглашения

- ▶ ABI (Application Binary Interface) - набор соглашений
 - ▶ как в функцию передаются параметры;
 - ▶ как функция возвращает значения;
 - ▶ какие регистры функция должна сохранить, а какие может испортить;
 - ▶ и многое другое.

Различные ABI

- ▶ Разные компиляторы используют различные ABI:
 - ▶ например, Microsoft используют свой собственный ABI;
 - ▶ Unix-like системы, зачастую, используют System V ABI.
- ▶ Мы будем использовать System V ABI
 - ▶ скачайте ABI и найдите, как в функцию передаются параметры.

Операционные Системы

Прерывания

April 28, 2017

Прерывания

- ▶ Прерывание - это событие, которое заставляет процессор прервать текущую задачу и вызвать специальный обработчик
 - ▶ внешнее устройство требует внимания;
 - ▶ произошла ошибка при выполнении инструкции;
 - ▶ специальная инструкция.

Асинхронные прерывания

- ▶ Прерывания могут происходить асинхронно
 - ▶ т. е. код не готов к тому, что его прервут
 - ▶ т. е. обработчик прерывания ответственен за сохранение состояния прерванной задачи.

Обработчики прерываний

- ▶ Откуда берутся обработчики прерываний?
 - ▶ часть ядра ОС;
 - ▶ ОС сообщает процессору, какой обработчик вызывать в какой ситуации.

Вызов обработчика прерывания

SS	RSP + 40		
RSP	RSP + 32	SS	RSP + 32
RFLAGS	RSP + 24	RSP	RSP + 24
CS	RSP + 16	RFLAGS	RSP + 16
RIP	RSP + 8	CS	RSP + 8
Error Code	RSP + 0	RIP	RSP + 0

Error Code

- ▶ Некоторые прерывания соответствуют ошибочным ситуациям
 - ▶ для некоторых из них на стек сохраняется Error Code.
- ▶ Error Code *иногда* содержит полезную для обработки ошибки информацию
 - ▶ а иногда он просто содержит 0.

Завершение обработчика прерывания

- ▶ Обработчик прерывания *обычно* завершается инструкцией *iretq*
 - ▶ для прерываний, сохраняющих Error Code, его *необходимо* удалить со стека.

Тело обработчика прерывания

- ▶ В общем случае зависит от прерывания
 - ▶ например, прерывания от сетевой карты и от таймера требуют разной обработки;
- ▶ Общая часть - сохранение состояния прерванной задачи:
 - ▶ *RIP* и *RFLAGS* не достаточно;
 - ▶ как минимум, нужно сохранить регистры общего назначения.

Таблица дескрипторов прерываний

- ▶ IDT указывает, каким прерываниям какие обработчики соответствуют
 - ▶ специальный регистр *IDTR* хранит адрес этой таблицы;
 - ▶ инструкции *LIDT* и *SIDT* позволяют записать/прочитать регистр *IDTR*.

Дескриптор IDT

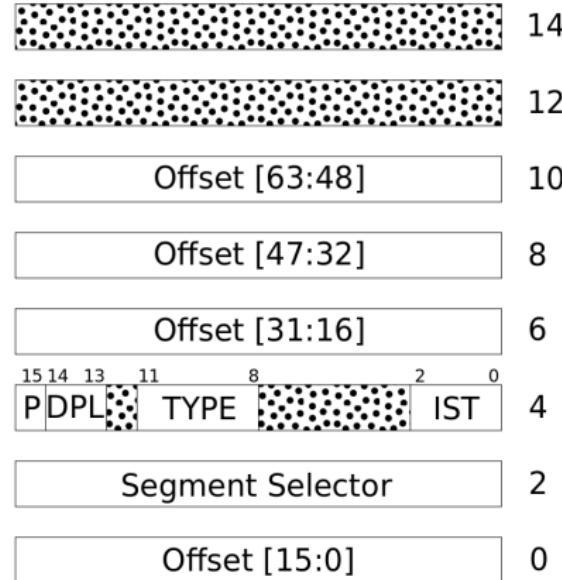


Таблица дескрипторов прерываний

- ▶ IDT может содержать максимум 256 записей
 - ▶ т. е. каждое ядро может обрабатывать 256 различных прерываний;
 - ▶ первые 32 из 256 зарезервированы под специальные нужды;
 - ▶ чему соответствуют оставшиеся 224?

Прерывания от внешних устройств

- ▶ Какое устройство какую запись в IDT использует?
 - ▶ может определяться настройкой устройства;
 - ▶ может определяться настройкой контроллера прерываний.

Контроллер прерываний

- ▶ Контроллер прерываний - посредник между устройствами и процессором
 - ▶ устройства сигналят контроллеру, контроллер сигналит процессору
 - ▶ задача контроллера - арбитраж (порядок обработки прерываний).
- ▶ Примеры контроллеров:
 - ▶ PIC (Programmable Interrupt Controller) (Intel 8259);
 - ▶ APIC (Advanced PIC)(Local APIC + IO APIC).

Запрет прерываний

- ▶ Зачем запрещать прерывания?
 - ▶ задача работает с данными, к которым обращается обработчик.
- ▶ Какие прерывания можно запрещать?
 - ▶ нельзя запрещать исключения (прерывания из-за ошибок).

Запрет прерываний

- ▶ Мы можем попросить устройство не генерировать прерывания
 - ▶ если мы знаем, какие прерывания могут привести к проблемам;
 - ▶ если устройство позволяет.
- ▶ Отключить прерывание на контроллере прерываний.

Запрет прерываний

- ▶ Отключить прерывание на процессоре
 - ▶ x86 регистр *RFLAGS* содержит флаг *IF*;
 - ▶ инструкция *cli* очищает флаг - запрещает прерывания;
 - ▶ инструкция *sti* устанавливает флаг.

Операционные Системы

Процесс загрузки

January 5, 2019

Первая инструкция

- ▶ Откуда берется первая инструкция?
 - ▶ x86 обращается по адресу *0xFFFFFFFF0*;
 - ▶ отвечает ему *материнская карта*.
- ▶ Какой код материнская карта отдает процессору?
 - ▶ BIOS (Basic Input/Output System) - наследство *IBM PC*;
 - ▶ UEFI (Unified Extensible Firmware Interface).

BIOS

- ▶ POST (Power-On Self-Test)
 - ▶ проверяет, что все на месте и "работает";
 - ▶ может выполнять начальную инициализацию устройств;
 - ▶ ищет загрузочное устройство (диск в ОС).

Загрузочное устройство

- ▶ BIOS ищет диск, с которого можно прочитать первые 512 байт
 - ▶ а. к. а. загрузочный сектор;
 - ▶ последние 2 байта сектора должны хранить числа $0x55$ и $0xAA$;
 - ▶ сектор загружается в память по физическому адресу $0x7c00$.
- ▶ BIOS передает управление по физическому адресу $0x7c00$
 - ▶ мы добрались до места, где мы можем на что-то повлиять.

Окружение

- ▶ Что нам известно о состоянии системы?
 - ▶ наш код начинается по физическому адресу $0x7c00$;
 - ▶ устройства как-то инициализированы и прерывания отключены;
 - ▶ процессор работает в *Real Mode*.

Real Mode

- ▶ Логический адрес состоит из двух частей:
 - ▶ 16-битного сегмента (*SEG*) и 16-битного смещения (*OFF*);
 - ▶ физический адрес получается по формуле $(SEG * 16 + OFF) \bmod 2^{20}$.
- ▶ Сегмент хранится в одном из специальных регистров:
 - ▶ *CS, DS, SS, ES, FS, GS.*

Real Mode

- ▶ Регистры общего назначения 16-битные:
 - ▶ *SP* - указатель стека;
 - ▶ *BP* - указатель "базы";
 - ▶ *AX, BX, CX, DX, SI, DI.*

Hello, World!

```
1      .code16
2      .text
3      .global  start
4 start:
5          ljmp 0x0, $real_start
6 real_start:
7          movw $0, %ax
8          movw %ax, %ds
9          movw %ax, %ss
10
11         movw $0x7c00, %sp
12         addw $0x0400, %sp
13         ...
14 loop:    jmp loop
```

Hello, World!

```
1      movw $0xB800, %ax
2      movw %ax, %es
3      movw $data, %si
4      movw $0, %di
5      movw size, %cx
6      call memcpy
7
8      ...
9
10     data:
11         .asciz "H\017e\017l\017o\017!\017"
12     size:
13         .short . - data
```

Hello, World!

```
1 memcpy:  
2         cmpw $0, %cx  
3         jz out  
4 next:  
5         movb (%si), %ah  
6         movb %ah, %es:(%di)  
7         incw %si  
8         incw %di  
9         decw %cx  
10        jnz next  
11 out:  
12         ret
```

Начальный загрузчик

- ▶ Как много кода можно поместить в первые 510 байт?
 - ▶ вряд ли туда поместится целая современная ОС;
 - ▶ задача этого кода прочитать с диска код, не поместившийся в первые 510 байт.
- ▶ Оставшийся код может быть кодом ОС,
 - ▶ а может быть кодом (вторичного) загрузчика;
 - ▶ например, GRUB.

Операционные Системы

Физическая память

January 5, 2019

Физическая и логическая память

- ▶ Логическая память - как видит память программа.
- ▶ Физическая память - как процессор видит память
 - ▶ логические адреса отображаются на физические;
 - ▶ все, что соблюдает интерфейс, выглядит как физическая память для CPU.

Физическая память

- ▶ Не все физические адреса соответствуют памяти
 - ▶ видеобуфер и другие устройства;
 - ▶ "дыры".
- ▶ Физическое адресное пространство может быть устроено довольно сложно
 - ▶ без карты памяти не разобраться.

Карта физической памяти

- ▶ Откуда брать карту физической памяти?
 - ▶ из документации аппаратной платформы;
 - ▶ спросить BIOS/UEFI/etc:
 - ▶ BIOS: прерывание 0x15, команда 0xe820;
 - ▶ UEFI: функция GetMemoryMap;
 - ▶ спросить загрузчик
 - ▶ GRUB поддерживает спецификацию multiboot.

Операционные Системы

Логическая память

January 5, 2019

Логическое адресное пространство

- ▶ Зачем вообще нужно разделение на логическое и физическое адресное пространства?
 - ▶ абстракция - приложение не знает о структуре физической памяти;
 - ▶ изоляция и защита - каждое приложение имеет свое логическое адресное пространство.

Понятие процесса

- ▶ Процесс - контейнер для ресурсов ОС
 - ▶ ОС может и не поддерживать (XX-DOS);
 - ▶ ОС выделяет ресурсы, например, память, процессам;
 - ▶ код, исполняющийся в рамках процесса, используют его ресурсы.

Понятие процесса

- ▶ Процессы, по-умолчанию, изолированы друг от друга:
 - ▶ у каждого процесса свое логическое адресное пространство;
 - ▶ т. е. код в рамках одного процесса не может залезть в память другого процесса.

Логическое адресное пространство

- ▶ Как логические адреса отображаются на физические?
- ▶ Как логические адресные пространства защищены?
 - ▶ сегментация (важно для x86);
 - ▶ paging (страничная организация памяти).

Сегментация

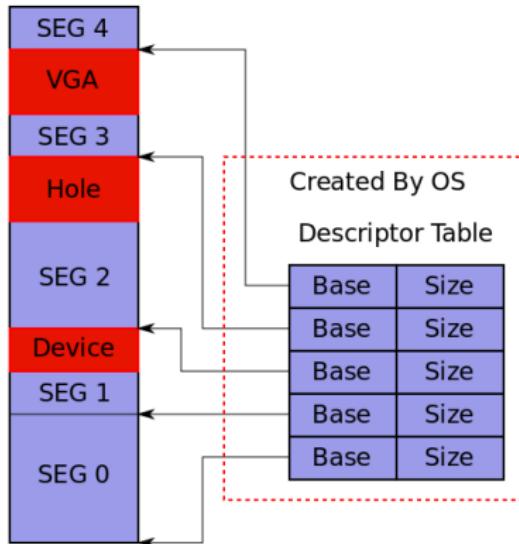
- ▶ Сегментация в Real Mode:
 - ▶ логический адрес - сегмент (*SEG*) и смещение (*OFF*);
 - ▶ *SEG* хранится в одном из сегментных регистров (*CS*, *SS*, *DS*, *ES*, *GS*, *FS*);
 - ▶ $A_{phys} = (SEG \times 16 + OFF) \bmod 2^{20}$.

Сегментация

- ▶ SEG - идентификатор сегмента физической памяти:
 - ▶ сегмент SEG начинается по физическому адресу $SEG \times 16$;
 - ▶ сегмент SEG имеет размер 2^{16} байт.
- ▶ А что если разрешить ОС изменять параметры сегмента?

Таблица дескрипторов сегментов

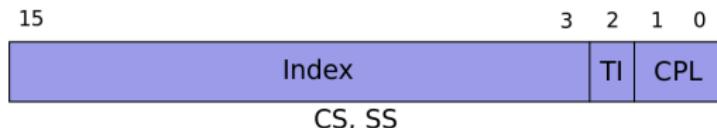
Physical Memory



Изоляция и защита с помощью сегментации

- ▶ Пусть ОС "выдает" каждому процессу свой дескриптор (*SEG*)
 - ▶ каждый дескриптор описывает свой участок физической памяти;
 - ▶ разные процессы пользуются разными дескрипторами (т. е. разные значения *SEG*);
 - ▶ непrivилегированному коду запрещено изменять сегментные регистры.

Селектор сегмента



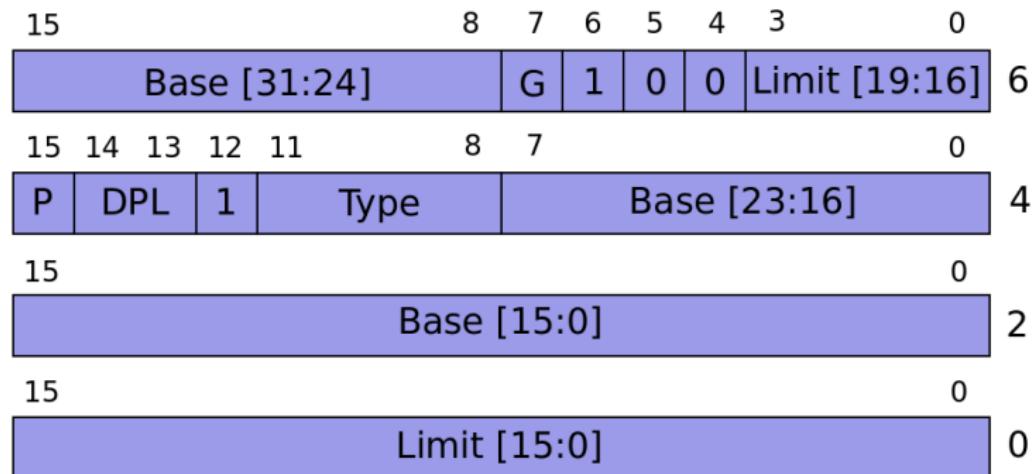
Уровни привилегий в x86

- ▶ В x86 выделяют 4 уровня привилегий:
 - ▶ ring0 - ring3;
 - ▶ ring0 - наивысший уровень привилегий (код ядра ОС);
 - ▶ ring3 - низший уровень привилегий (код пользовательских приложений).

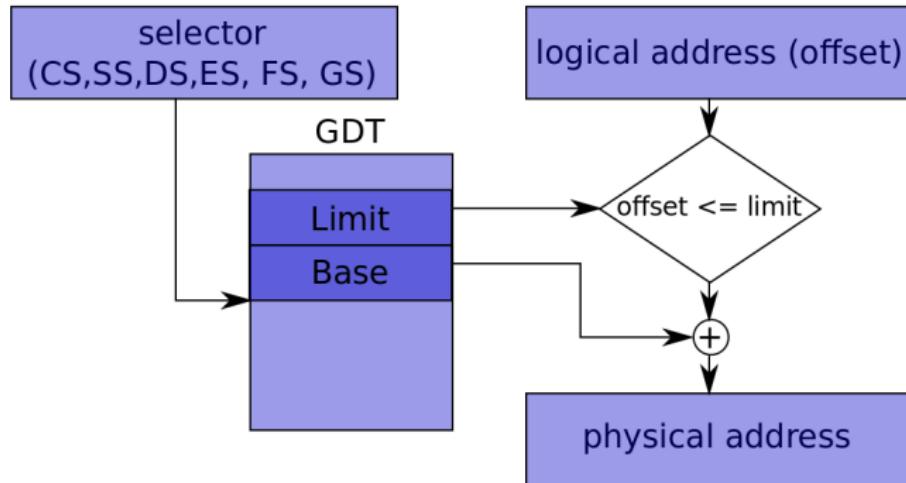
Global Descriptor Table

- ▶ В x86 таблицу дескрипторов называют GDT
 - ▶ адрес и размер GDT хранятся в специальном регистре GDTR;
 - ▶ писать и читать GDTR можно с помощью инструкций *LIDT* и *SIDT*;
 - ▶ писать в GDTR может только привилегированный код.

Дескриптор сегмента в Protected Mode



Преобразование в физический адрес



Сегментация в Long Mode

- ▶ Сегментация в Long Mode *практически* не используется
 - ▶ ES, DS, FS, GS обычно равны 0;
 - ▶ поля *Base* и *Limit* дескрипторов игнорируются;
 - ▶ CS и SS все еще хранят *CPL*.
- ▶ Вместо сегментации используется paging.

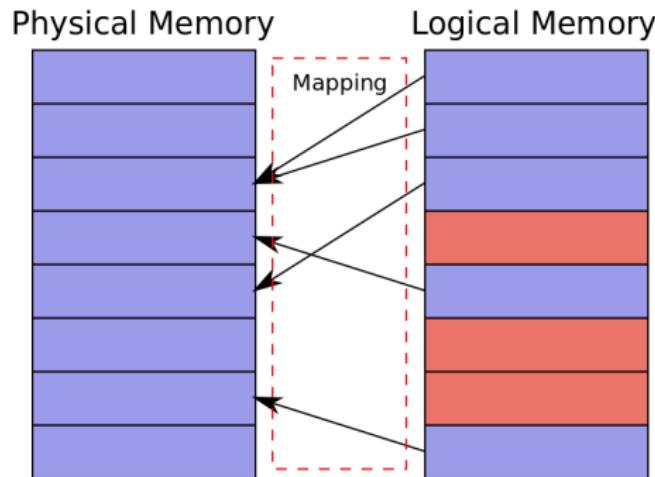
Paging

- ▶ Давайте просто использовать словарь
 - ▶ словарь хранит отображение логических адресов на физические;
 - ▶ ядро ОС создает свой словарь для каждого процесса.

Paging

- ▶ Как должен выглядеть словарь?
 - ▶ отображать каждый байт отдельно непрактично;
 - ▶ отображение происходит блоками фиксированного размера (страницами);
 - ▶ размер страницы определяется архитектурой (типичные размеры: 4Kb и 64Kb).

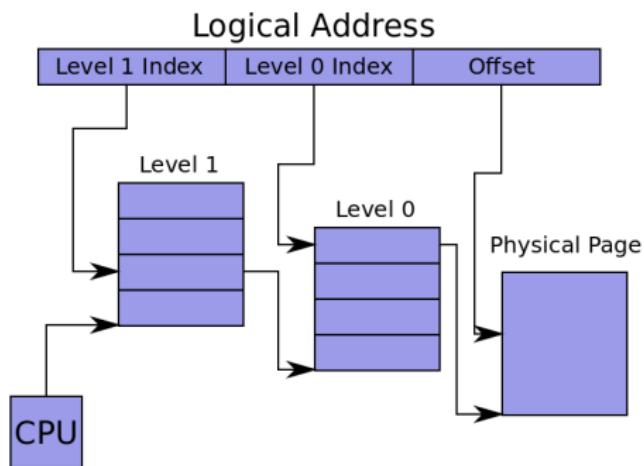
Paging



Paging

- ▶ Как должен выглядеть словарь?
 - ▶ не каждый процесс использует все логическое адресное пространство (даже в 32-битных системах и тем более в 64-битных);
 - ▶ не хочется хранить информацию для неиспользуемых страниц;
 - ▶ структура должна быть сравнительно простой.

Таблица страниц



Translation Lookaside Buffer

- ▶ А вы заметили проблему таблиц страниц?
 - ▶ мы хотим прочитать 1 байт по некоторому логическому адресу;
 - ▶ процессор должен прочитать записи в нескольких таблицах.

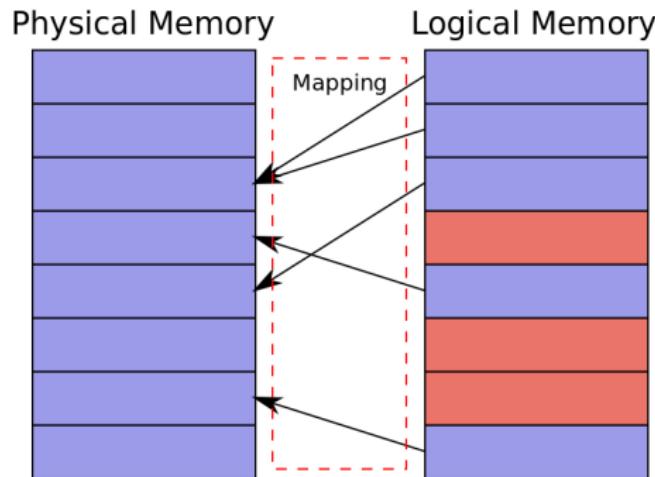
Translation Lookaside Buffer

- ▶ Процессор кэширует результаты трансляции в TLB:
 - ▶ TLB может значительно ускорить обращение к памяти;
 - ▶ если код не обращается каждый раз к новой странице.

Translation Lookaside Buffer

- ▶ Процессор не может отследить изменения в таблицах страниц:
 - ▶ TLB не прозрачен, т. е. необходимо явно "сбрасывать" записи;
 - ▶ об этом тоже должно заботиться ядро ОС.

Page Fault



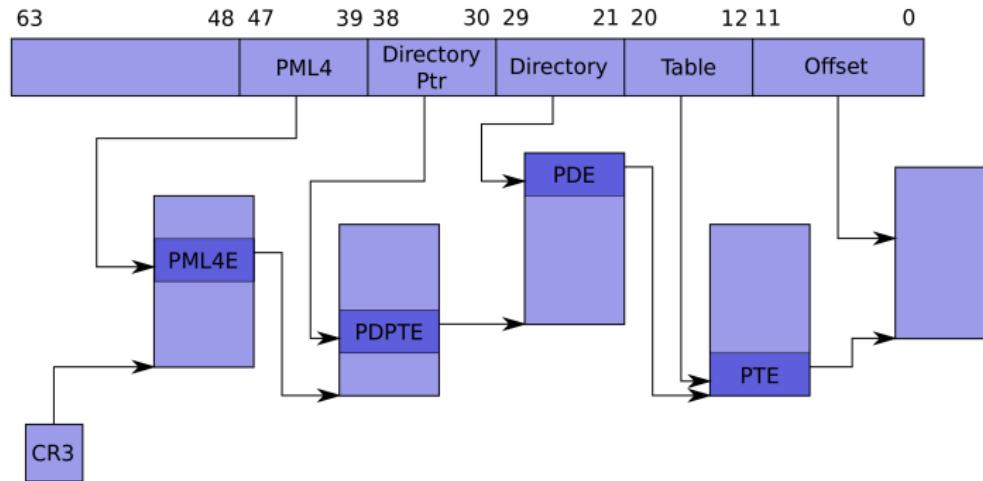
Page Fault

- ▶ Не все записи в таблицах страниц используются
 - ▶ что, если код обратится к логическому адресу, для которого нет отображения?
 - ▶ генерируется специальное исключение - Page Fault.

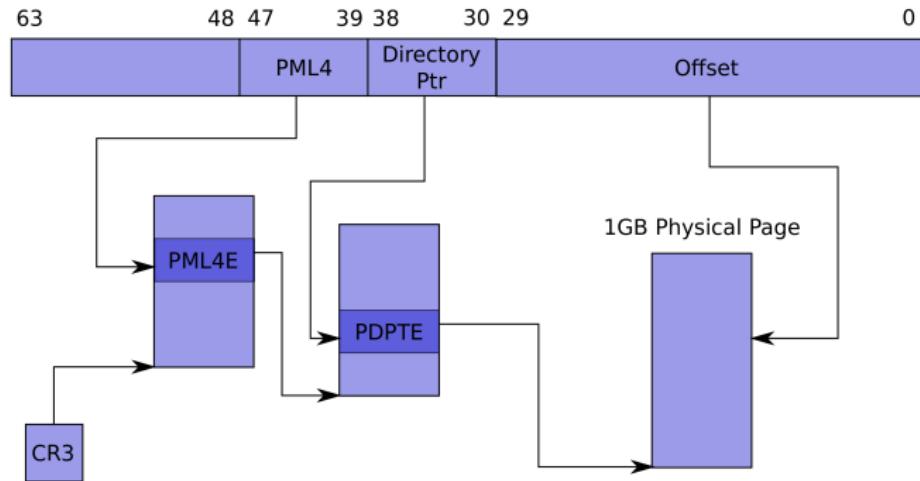
Защита памяти

- ▶ Paging также позволяет запретить некоторые действия с памятью:
 - ▶ мы уже видели запрет на обращение к памяти;
 - ▶ запись в какой-то участок логической памяти;
 - ▶ исполнение кода из какого-то участка памяти;
 - ▶ обращение непrivилегированного кода.

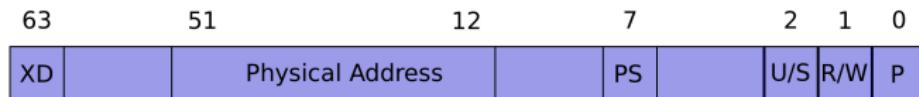
Paging в x86 Long Mode



Paging в x86 Long Mode



Paging в x86 Long Mode



- ▶ P - если 0, запись не используется;
- ▶ R/W - если 0, то запись запрещена;
- ▶ U/S - если 0, то запрещен непrivилегированный доступ;
- ▶ PS - если 1, это последний уровень;
- ▶ XD - если 1, то запрещено исполнение.

Резюме

- ▶ Логическое и физическое адресное пространства:
 - ▶ программы используют логические адреса (указатели);
 - ▶ процессор использует физические адреса;
 - ▶ ОС определяет как логические адреса отображаются на физические.

Резюме

- ▶ Понятие процесса:
 - ▶ каждый процесс имеет свое логические адресное пространство;
 - ▶ процессы изолированы друг от друга.

Резюме

- ▶ Сегментация и страничная адресация памяти:
 - ▶ ОС использует эти аппаратные механизмы для организации изоляции процессов;
 - ▶ многие современные архитектуры с поддержкой защиты памяти используют paging (и очень немногие сегментацию).

Операционные Системы

Алгоритмы аллокации памяти

January 5, 2019

Аллокация памяти

- ▶ Рассмотрим простейшую постановку задачи:
 - ▶ есть непрерывный участок логической памяти;
 - ▶ функция аллокации: `void *alloc(int size);`
 - ▶ функция освобождения: `void free(void *free);`

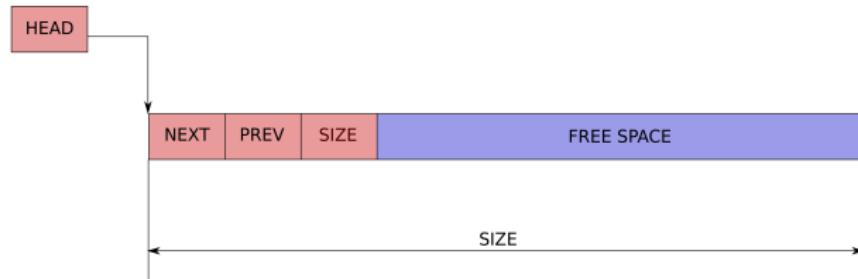
Выравнивание

- ▶ Некоторые процессоры требуют выравненных указателей
 - ▶ 2 байта - выравнивание 2 байта;
 - ▶ 4 байта - выравнивание 4 байта;
 - ▶ 8 байт - выравнивание 8 байт...

Простой подход к аллокации

- ▶ Создадим связный список свободных блоков
 - ▶ каждый узел списка описывает непрерывный свободный участок;
 - ▶ узлы списка хранятся в начале каждого свободного блока.

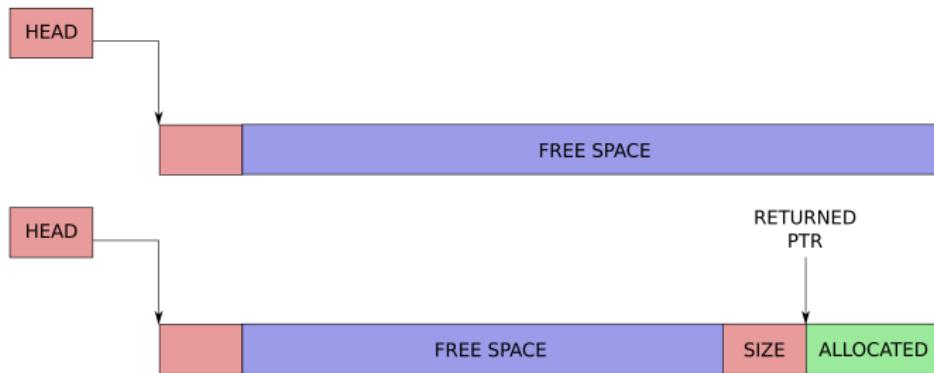
Связный список свободных блоков



Аллокация свободного блока

- ▶ Пройдемся по списку и найдем блок достаточного размера
 - ▶ если блок слишком большой, то отрезаем от него часть;
 - ▶ если подходящего блока не нашлось, то возвращаем ошибку.

Аллокация свободного блока



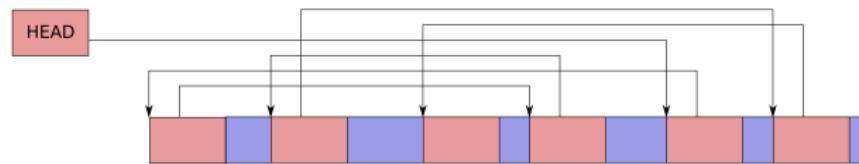
Освобождение занятого блока

- ▶ Чтобы освободить свободный блок, его нужно вернуть в список
 - ▶ например, можно добавить в список новый элемент.

Освобождение занятого блока



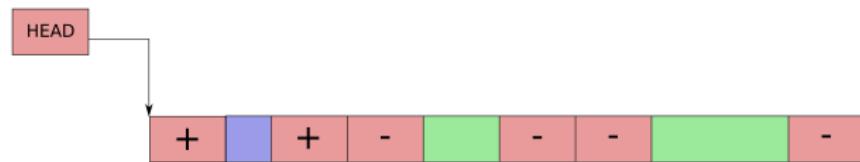
Фрагментация свободной памяти



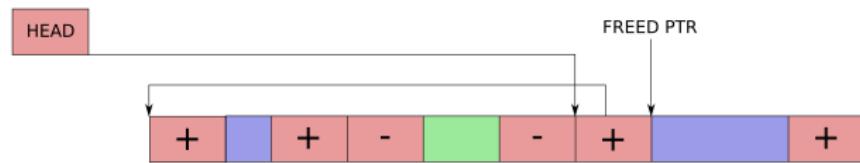
Боремся с фрагментацией

- ▶ Как избежать подобной фрагментации?
 - ▶ искать смежные блоки проходом по списку ($O(N)$);
 - ▶ поддерживать список упорядоченным ($O(N)$);
 - ▶ использовать упорядоченную структуру вместо списка ($O(\log N)$);
 - ▶ использовать граничные маркеры (Border Tags, $O(1)$).

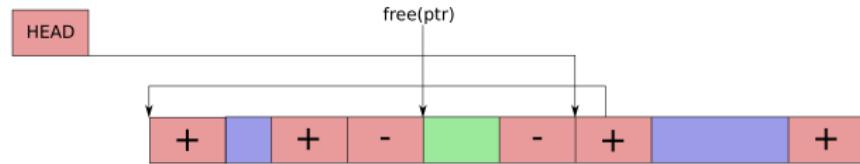
Границные маркеры



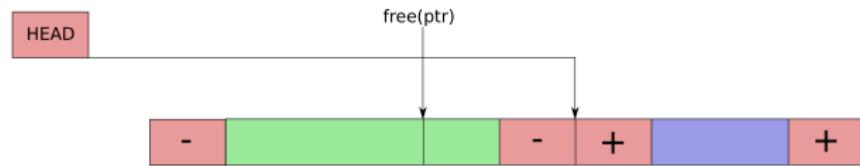
Границные маркеры



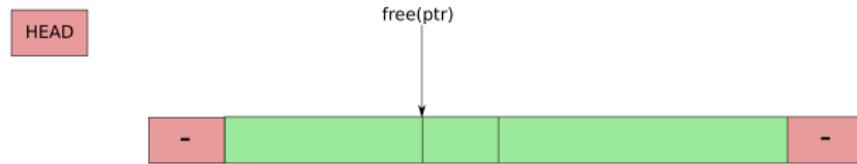
Границные маркеры



Границные маркеры



Границные маркеры



Границные маркеры



Buddy аллокатор

- ▶ Buddy аллокатор предназначен для аллокации больших участков памяти
 - ▶ buddy аллокаторalloцирует память блоками;
 - ▶ блок - 2^i последовательных страниц;
 - ▶ например, 1 страница, 2 страницы, 4 страницы и т. д.;
 - ▶ но не 3 страницы или 5 страниц.

Дескрипторы страниц

- ▶ Buddy аллокатор не будет работать с памятью напрямую
 - ▶ вместо страниц памяти будем использовать в алгоритме дескрипторы;
 - ▶ просто массив дескрипторов - по адресу страницы легко получить дескриптор и наоборот.

Дескрипторы страниц

- ▶ Что будет храниться в дескрипторах?
 - ▶ указатели, чтобы связать дескрипторы в двусвязный список;
 - ▶ признак свободности/занятости;
 - ▶ уровень (указание на размер блока).

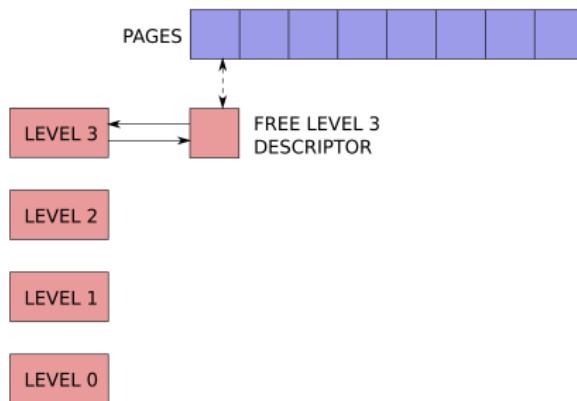
Списки свободных блоков

- ▶ Пусть у нас изначально есть 2^N последовательных свободных страниц:
 - ▶ заведем $N + 1$ изначально пустой двусвязный список;
 - ▶ i -ый список будет хранить свободные блоки размером 2^i страниц.

Начальное состояние

- ▶ Возьмем дескриптор 0-ой страницы
 - ▶ отметим дескриптор как свободный;
 - ▶ зададим в дескрипторе уровень $N - 1$;
 - ▶ добавим дескриптор в список $N - 1$.

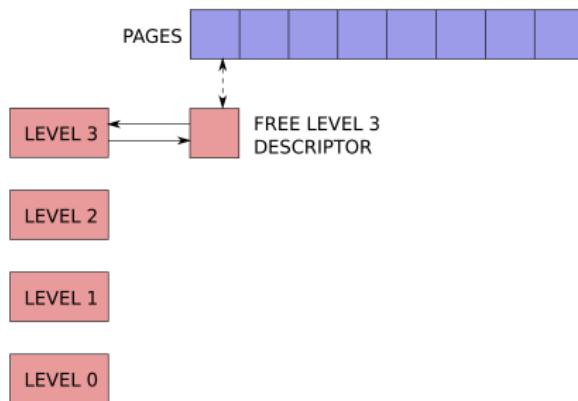
Инициализация



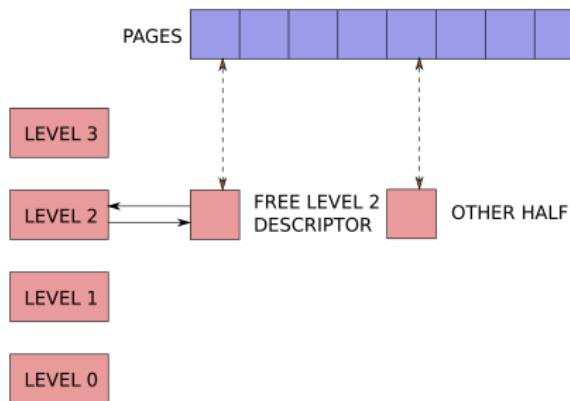
Аллокация

- ▶ Мы хотим аллоцировать 2^i страниц
 - ▶ найдем непустой список с блоками $\geq 2^i$;
 - ▶ берем один из блоков и делим его пополам, пока не останется 2^i .

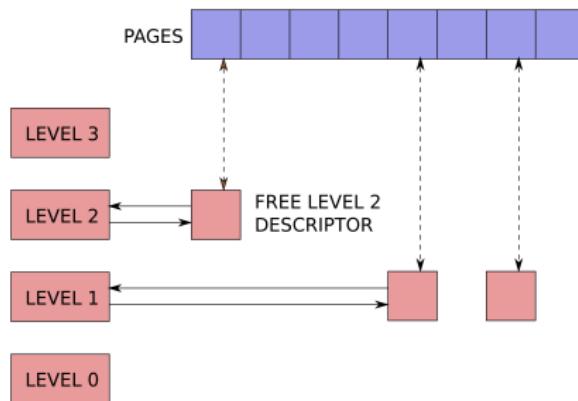
Аллокация



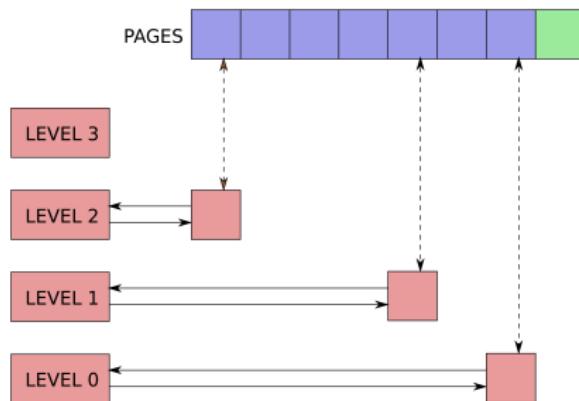
Аллокация



Аллокация



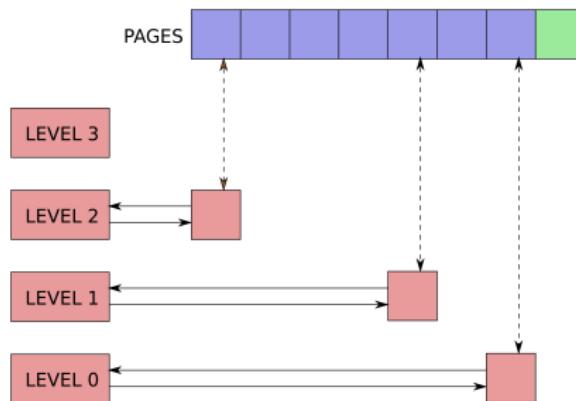
Аллокация



Освобождение

- ▶ Теперь мы хотим освободить блок размера 2^i
 - ▶ мы могли бы просто добавить дескриптор в список i , но это приведет к фрагментации;
 - ▶ мы должны попытаться объединить смежные блоки.

Buddies



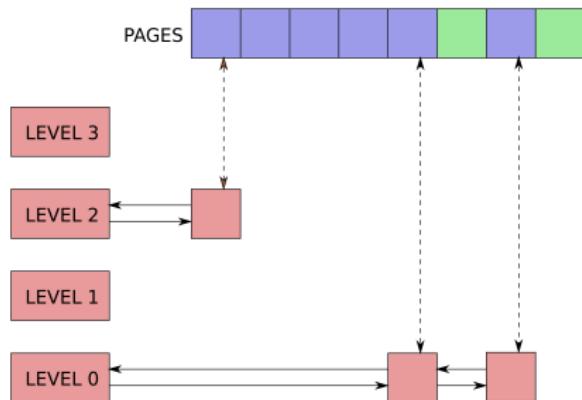
Buddies

- ▶ Как найти парный блок?
 - ▶ если мы осовобождаем блок размера 2^i с номером j ;
 - ▶ парный блок имеет номер $j \oplus 2^i$;
 - ▶ \oplus - исключающее побитовое ИЛИ.

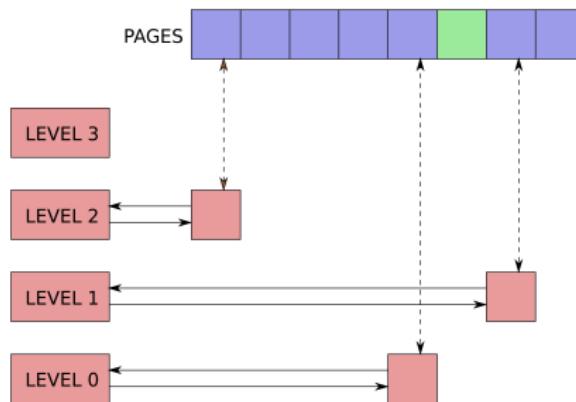
Освобождение

- ▶ Когда можно объединять парные блоки?
 - ▶ если оба блока свободны;
 - ▶ если оба блока имеют один размер (уровень в дескрипторе).

Освобождение



Buddies



Кеширующий аллокатор

- ▶ Создадим кеш блоков фиксированного размера
 - ▶ кеш будет аллоцировать/освобождать большие регионы памяти, используя другой аллокатор;
 - ▶ кеш "нарезает" большие регионы на блоки фиксированного размера.

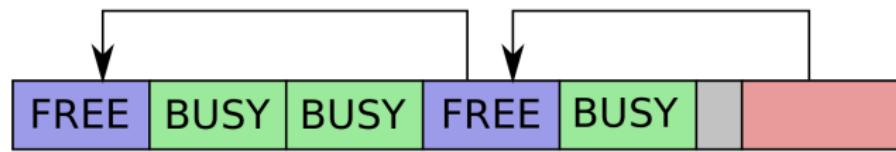
Кеширующий аллокатор

- ▶ Кеширующий аллокатор имеет ряд достоинств:
 - ▶ фиксированный размер блоков позволяет бороться с фрагментацией;
 - ▶ аллокация/освобождение могут работать за $O(1)$;
 - ▶ можно скомбинировать кэши разных размеров и построить универсальный аллокатор.

SLAB

- ▶ Кеширующий аллокатор, предложенный Джейфом Бонвиком и использованный в SunOS (Solaris):
 - ▶ slab - описывает большой регион памяти, который разбивается на маленькие блоки фиксированного размера;
 - ▶ все свободные блоки связываются в список;
 - ▶ количество элементов списка и указатель на первый элемент сохраняются в заголовке.

SLAB



Управление SLAB-ами

- ▶ SLAB аллокатор управляет slab-ами:
 - ▶ если нет slab-а со свободными объектами - аллоцируем новый;
 - ▶ если все объекты в slab-е свободны - можно освободить slab.

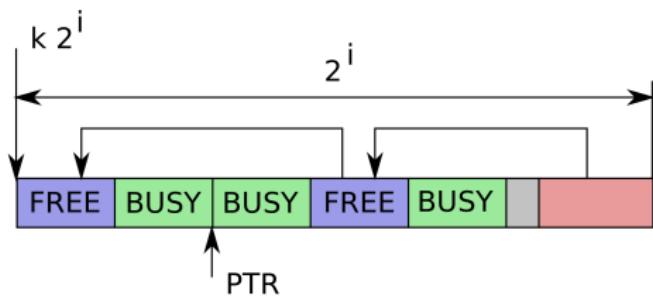
Управление SLAB-ами

- ▶ SLAB аллокатор поддерживает три списка slab-ов:
 - ▶ полностью свободные slab-ы;
 - ▶ частично занятые slab-ы;
 - ▶ полностью занятые slab-ы.

Освобождение

- ▶ Чтобы освободить элемент, нужно найти slab, которому он принадлежит
 - ▶ мы можем сохранить указатель на slab рядом с аллоцированной памятью;
 - ▶ мы можем потребовать, чтобы размер и выравнивание slab-а были равны 2^i .

Освобождение



Операционные Системы

Потоки исполнения

January 5, 2019

Поток исполнения

- ▶ Поток исполнения - это код и его состояние (а. к. а. контекст)
 - ▶ код - набор инструкций в памяти, на который указывает регистр *RIP*;
 - ▶ контекст потока включает значения регистров и память.

Потоки и процессы

- ▶ Поток работает в контексте некоторого процесса
 - ▶ т. е. поток "живет" в логическом адресном пространстве процесса;
 - ▶ несколько потоков могут работать в рамках одного процесса;
 - ▶ процесс имеет как минимум один поток.

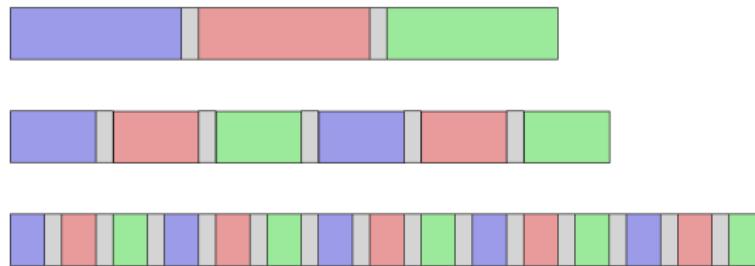
Стек потока

- ▶ Каждый поток исполнения имеет свой собственный стек
 - ▶ стек хранит адреса возвратов и локальные переменные;
 - ▶ для процессора стек - место в памяти, куда указывает *RSP*.

Многопоточность

- ▶ В системе могут *одновременно* работать несколько потоков исполнения
 - ▶ на нескольких ядрах процессора;
 - ▶ на одном ядре, создавая *иллюзию* одновременной работы.

Многопоточность



Переключение между потоками

- ▶ Для переключения между потоками необходимо:
 - ▶ сохранить контекст исполняемого потока;
 - ▶ восстановить контекст потока, на который мы переключаемся.

Пример переключения для x86

```
1      .text
2      switch_threads:
3          pushq %rbx
4          pushq %rbp
5          pushq %r12
6          pushq %r13
7          pushq %r14
8          pushq %r15
9          pushfq
10
11         movq %rsp, (%rdi)
12         movq %rsi, %rsp
13
14         popfq
15         popq %r15
16         popq %r14
17         popq %r13
18         popq %r12
19         popq %rbp
20         popq %rbx
21
22         retq
```

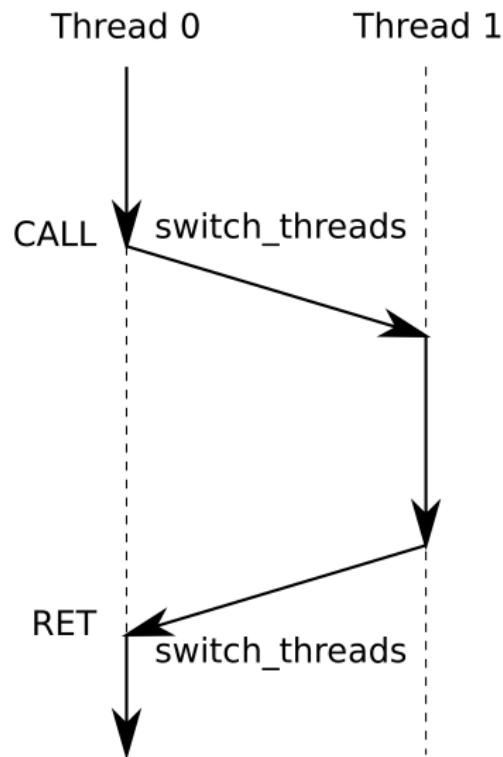
C API

- ▶ `void switch_threads(void **prev, void *next);`
- ▶ по завершении функции `*prev` будет указывать на сохраненный контекст;
- ▶ `next` указывает на сохраненный контекст потока, на который мы переключаемся.

switch_threads

```
1      .text
2  switch_threads:
3      /* save context on stack */
4      pushq %rbx
5      pushq %rbp
6      pushq %r12
7      pushq %r13
8      pushq %r14
9      pushq %r15
10     pushfq
11
12     /* rdi - the first argument */
13     movq %rsp, (%rdi)
14
15     /* rsi - the second argument */
16     movq %rsi, %rsp
17
18     /* restore from stack */
19     popfq
20     popq %r15
21     popq %r14
22     popq %r13
23     popq %r12
24     popq %rbp
25     popq %rbx
26
27     retq /* ! */
```

Переключение потоков



Создание нового потока

- ▶ Как создать новый поток и переключиться на него в первый раз?
 - ▶ нам нужно выделить место для хранения указателя на контекст;
 - ▶ нам нужно выделить место под стек нового потока;
 - ▶ нам нужно сохранить на стеке начальный контекст и сохранить указатель на него.

Начальный контекст

```
1 struct switch_frame {  
2     uint64_t rflags;  
3     uint64_t r15;  
4     uint64_t r14;  
5     uint64_t r13;  
6     uint64_t r12;  
7     uint64_t rbp;  
8     uint64_t rbx;  
9     uint64_t rip;  
10 } __attribute__((packed));
```

Кооперативная многозадачность

- ▶ Невытесняющая (кооперативная) многозадачность
 - ▶ поток должен сам вызвать функцию переключения;
 - ▶ что если в коде содержится ошибка?
 - ▶ или мы обращаемся к библиотеке, которая выполняет долгую операцию?

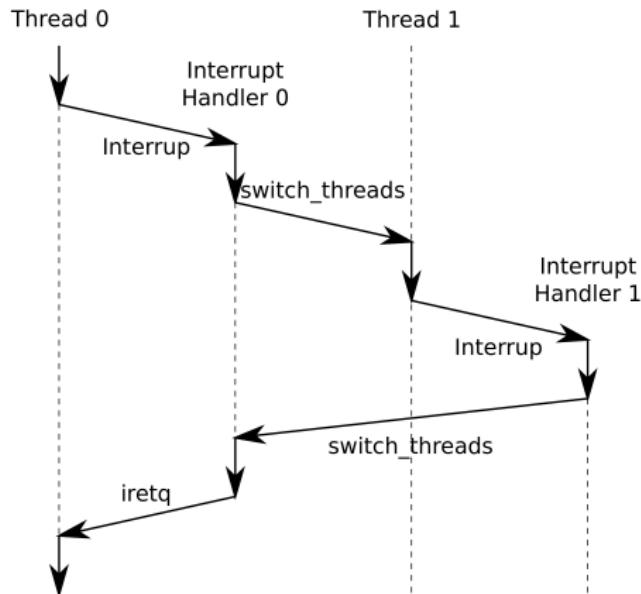
Вытесняющая многозадачность

- ▶ Вытесняющая (preemptive) многозадачность
 - ▶ поток снимается ОС с CPU "силой", по истечении кванта времени;
 - ▶ синхронизация потоков при этом усложняется;
 - ▶ как организовать вытесняющую многозадачность?

Сново о прерываниях

- ▶ Обработчик прерывания "прерывает" исполняемый код
 - ▶ но обработчик работает в контексте прерванного потока;
 - ▶ функцию переключения контекста можно вызвать от имени потока из обработчика прерываний.

Вытесняющая многозадачность



Таймер

- ▶ Таймер может генерировать прерывания с заданной периодичностью
 - ▶ Programmable Interval Timer (PIT, intel 8253) - IBM PC;
 - ▶ High Precision Event Timer (HPET);
 - ▶ Local APIC Timer.

Планирование потоков

- ▶ Планировщик (scheduler) - компонент ОС, который определяет
 - ▶ когда переключаться с потока;
 - ▶ на какой поток переключаться.

Простое планирование

- ▶ Рассмотрим простейшую задачу планирования
 - ▶ все задачи известны заранее;
 - ▶ про каждую задачу известно, сколько времени она займет;
 - ▶ задачи работают без переключений;
 - ▶ т. е. нам осталось только определить порядок.

Пропускная способность



A, 9с



B, 7с



C, 3с

Schedule



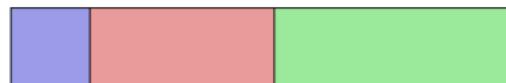
Пропускная способность

 A, 9с

 B, 7с

 C, 3с

Schedule



Среднее время ожидания

- ▶ Пусть все задачи принадлежат разным пользователям
 - ▶ пользователю важно, сколько ему нужно ждать завершения его задачи;
 - ▶ давайте в качестве метрики использовать среднее время ожидания.

Среднее время ожидания



A, 9с



B, 7с



C, 3с

Schedule



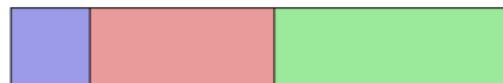
Среднее время ожидания

 A, 9с

 B, 7с

 C, 3с

Schedule



Динамическое создание задач

- ▶ Зачастую все задачи не известны заранее
 - ▶ задачи могут создаваться в произвольные моменты времени;
 - ▶ каждая вновь появившаяся задача может изменить решение планировщика.

IO операции

- ▶ Задачи могут давать команды устройствам или ждать каких-то событий:
 - ▶ запись/чтение на/с HDD (порядка нескольких мс);
 - ▶ ждать входящих соединений по сети;
 - ▶ ждать, пока пользователь нажмет на клавишу.

IO операции

- ▶ Пока задача ждет завершения IO операции, можно забрать у нее CPU
 - ▶ время ожидания может быть большим (1мс - очень много для CPU);
 - ▶ утилизация CPU - сколько времени CPU делал полезную работу.

Утилизация



Schedule



Утилизация



Schedule



Информация о задаче

- ▶ Зачастую время работы задачи и расписание ее IO не известны
 - ▶ задачи могут влиять друг на друга или зависеть от внешних обстоятельств;
 - ▶ мы можем оценивать эти параметры и классифицировать задачи.

IO-bounded и CPU-bounded

- ▶ IO-bounded задачи - много IO, но мало вычислений:
 - ▶ например, текстовый редактор;
 - ▶ вообще приложения, ожидающие ввода пользователя.
- ▶ CPU-bounded задачи - много вычислений, но мало IO:
 - ▶ например, научные вычисления;
 - ▶ компиляция программ.

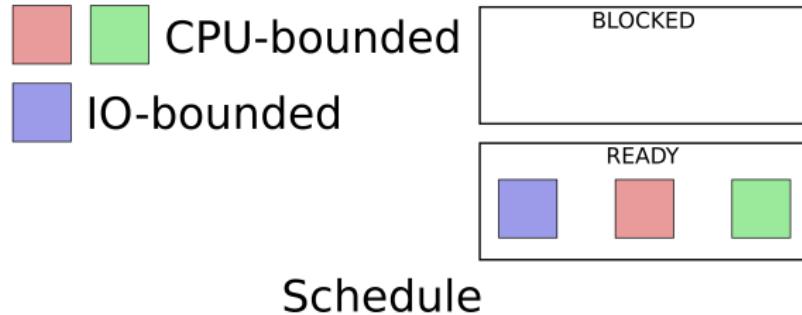
Round Robin

- ▶ Round Robin - выдаем потокам квант времени на CPU по очереди:
 - ▶ поток, отработавший свой квант, встает в конец очереди;
 - ▶ каждый новый поток встает в конец очереди;
 - ▶ потоки, дождавшиеся завершения IO, встают в конец очереди;
 - ▶ CPU отдается потоку в начале очереди.

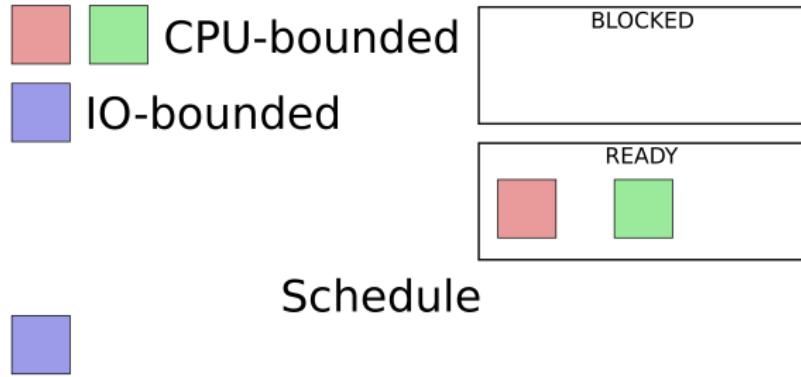
Достоинства Round Robin

- ▶ К списку достоинств Round Robin можно отнести:
 - ▶ подход очень прост;
 - ▶ время ожидания CPU ограничено - никто не голодает.

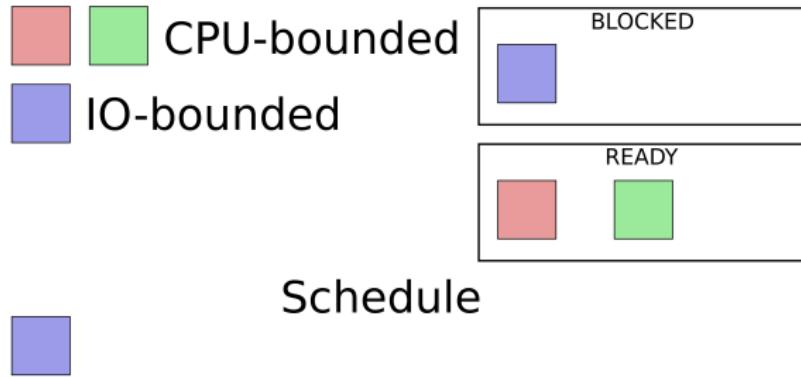
Round Robin



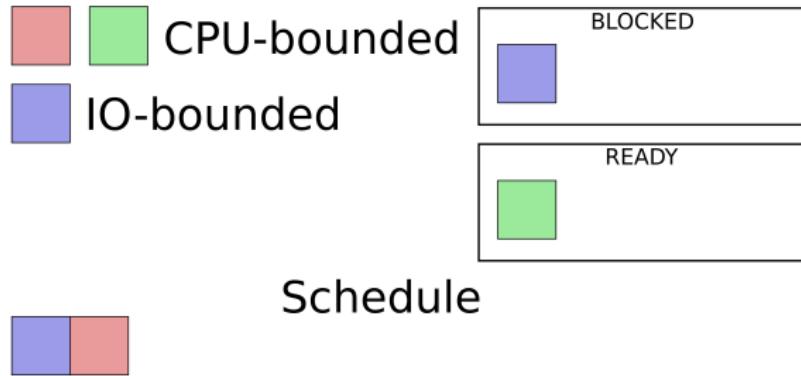
Round Robin



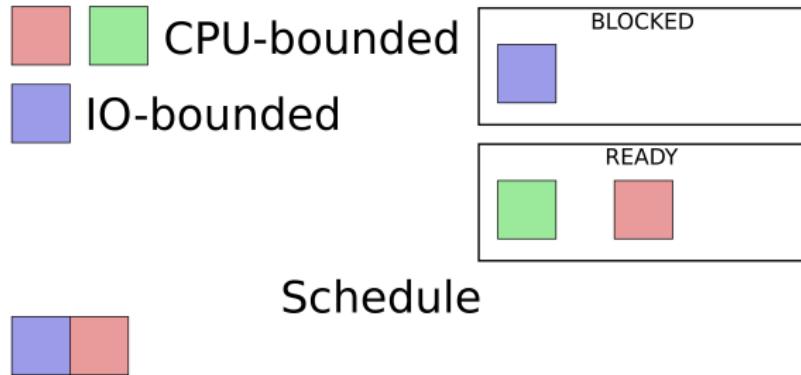
Round Robin



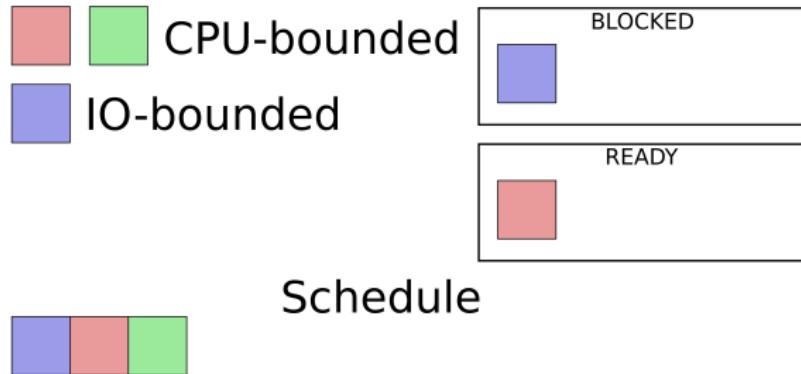
Round Robin



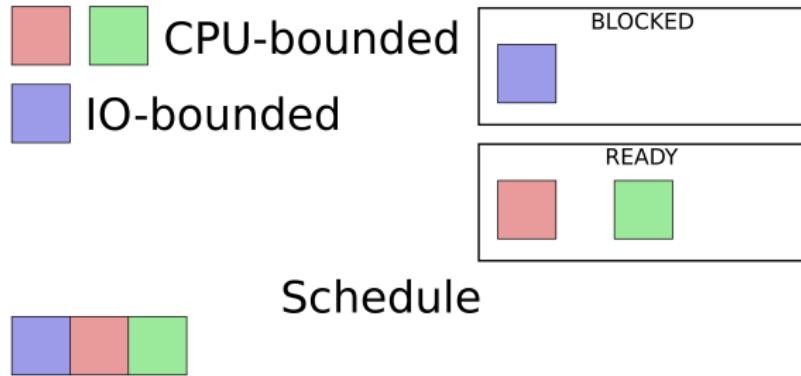
Round Robin



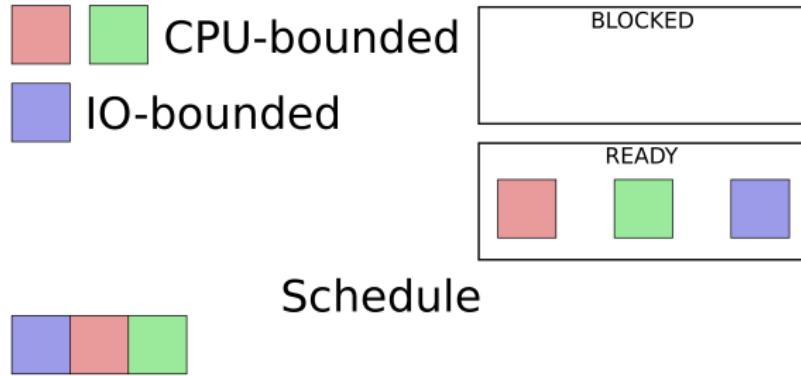
Round Robin



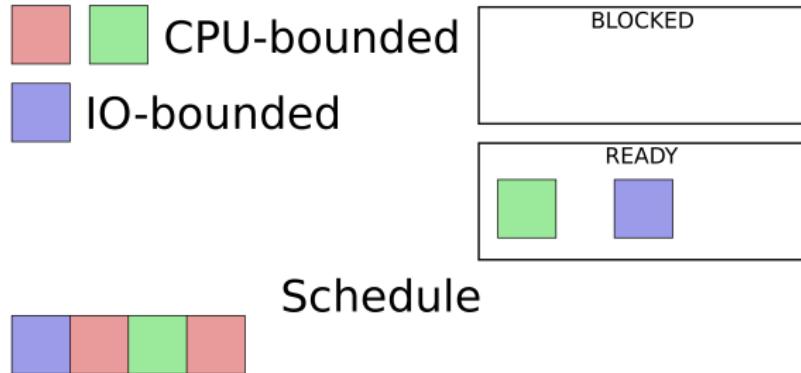
Round Robin



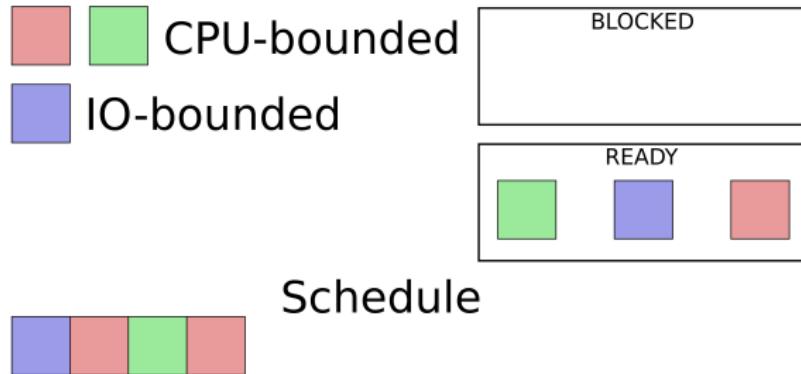
Round Robin



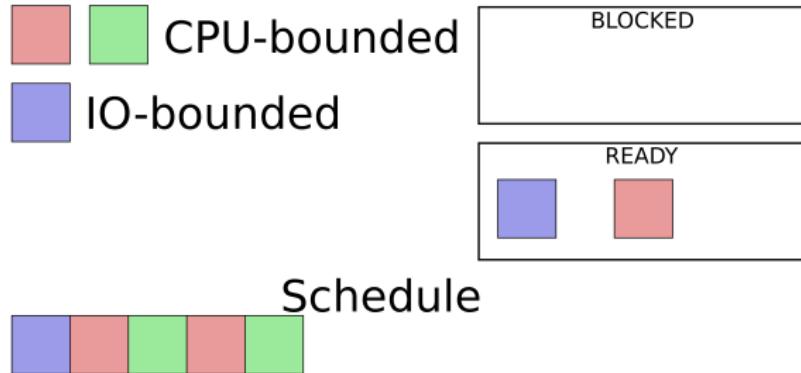
Round Robin



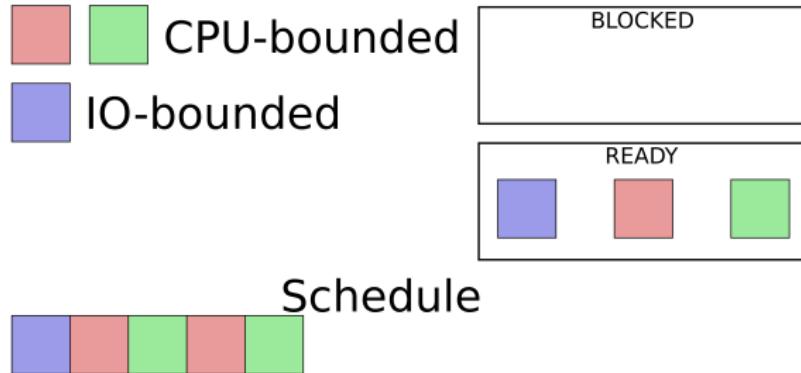
Round Robin



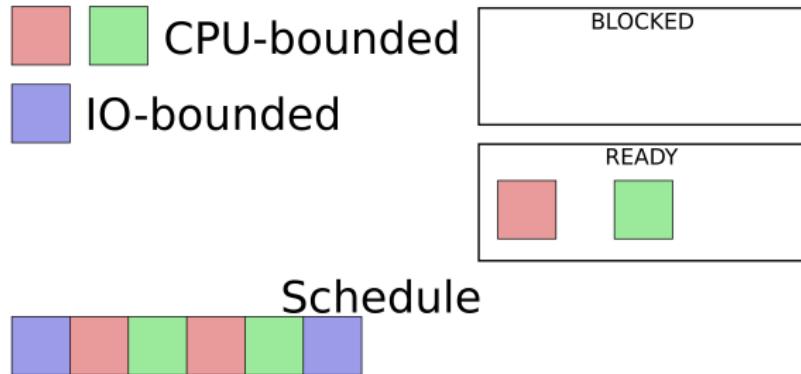
Round Robin



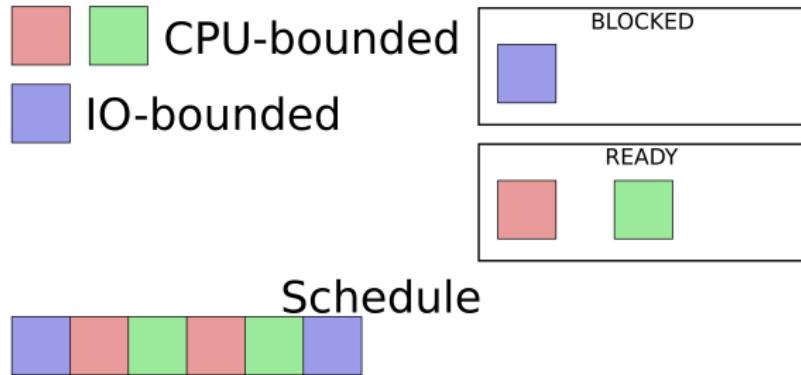
Round Robin



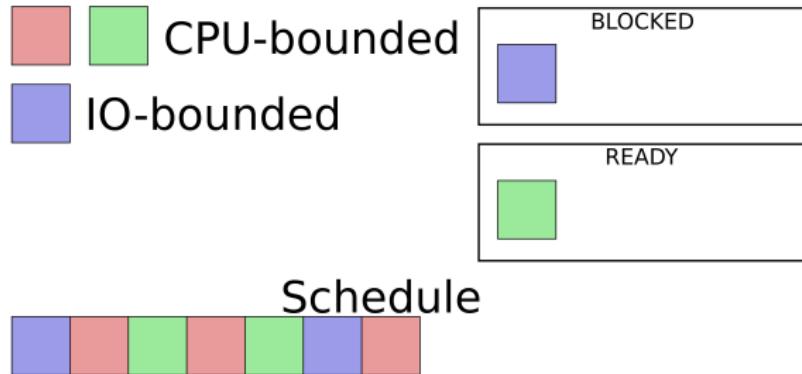
Round Robin



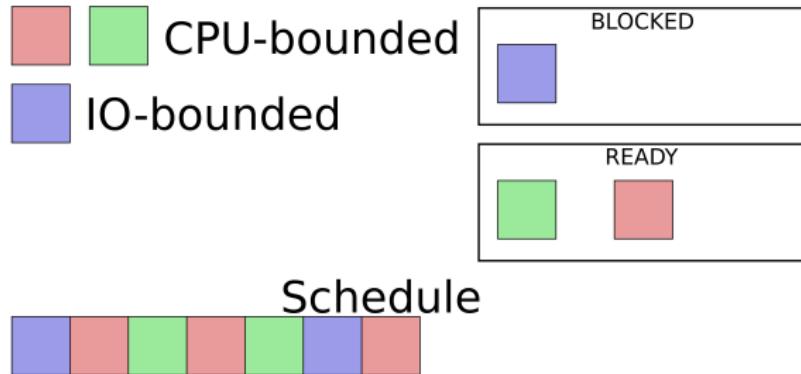
Round Robin



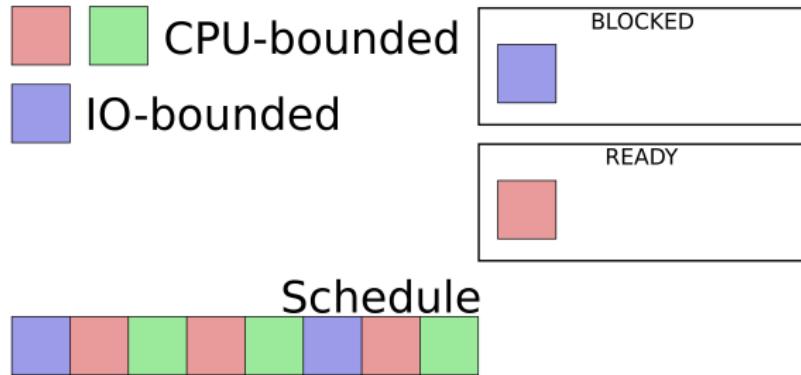
Round Robin



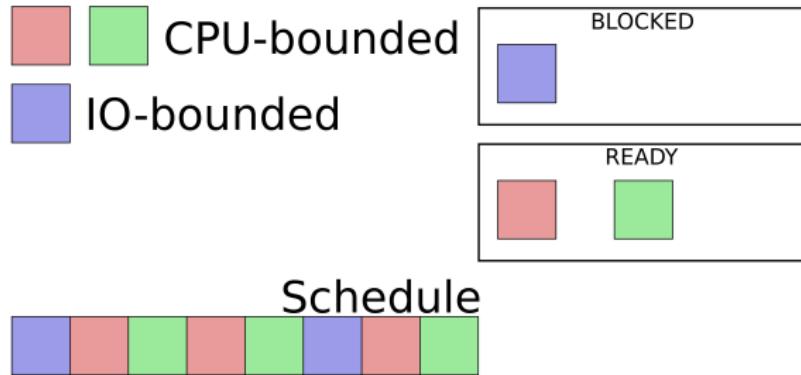
Round Robin



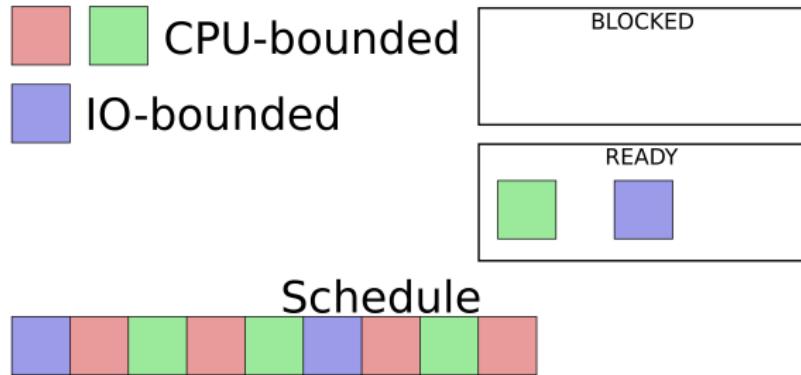
Round Robin



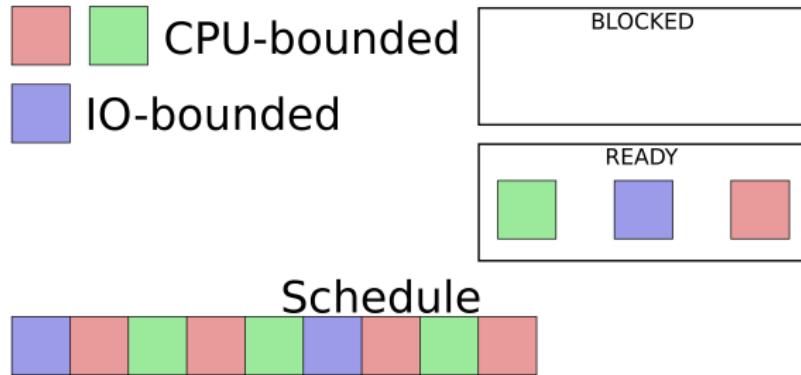
Round Robin



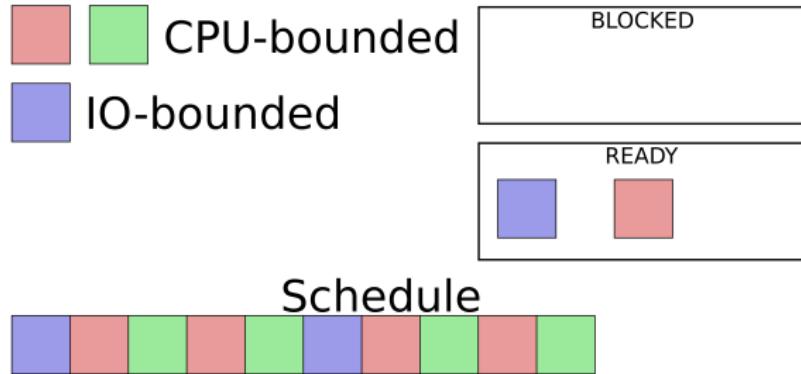
Round Robin



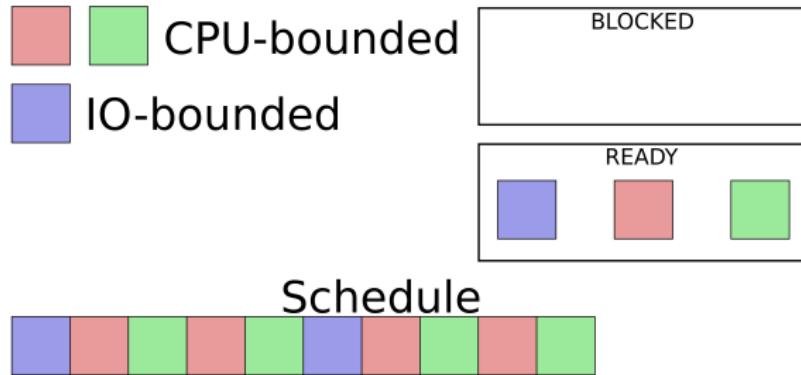
Round Robin



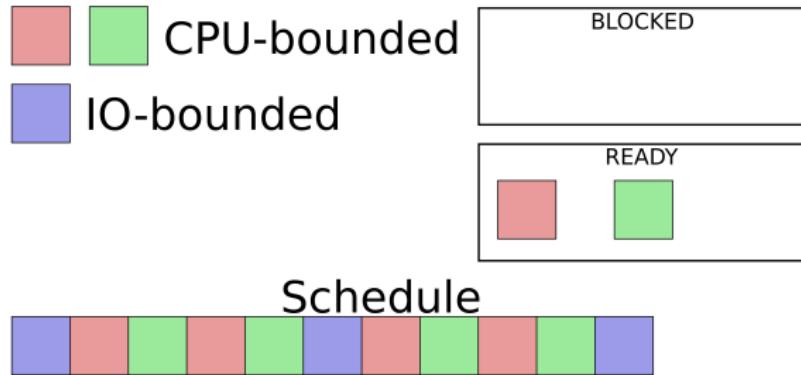
Round Robin



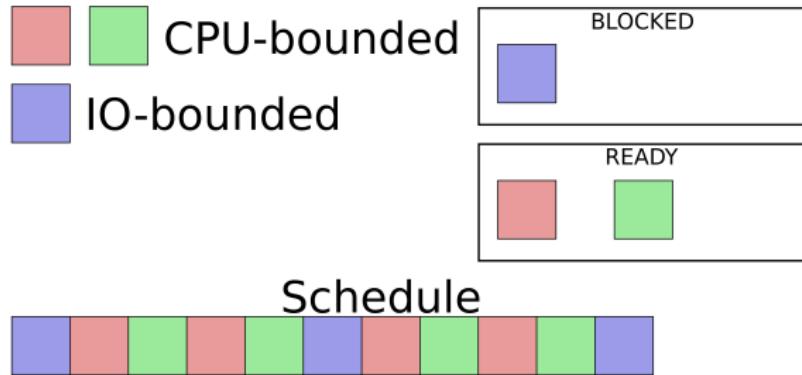
Round Robin



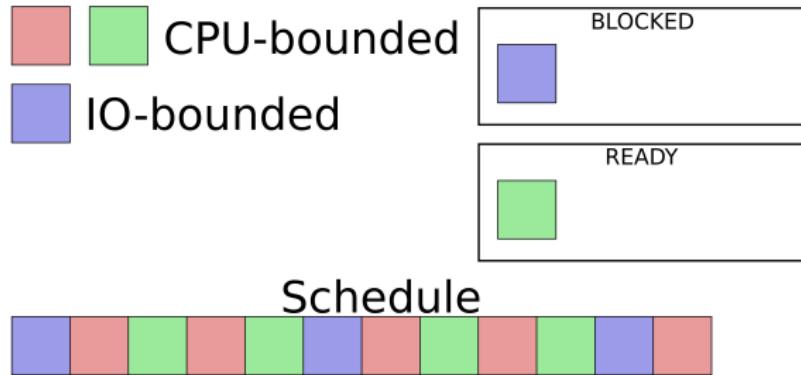
Round Robin



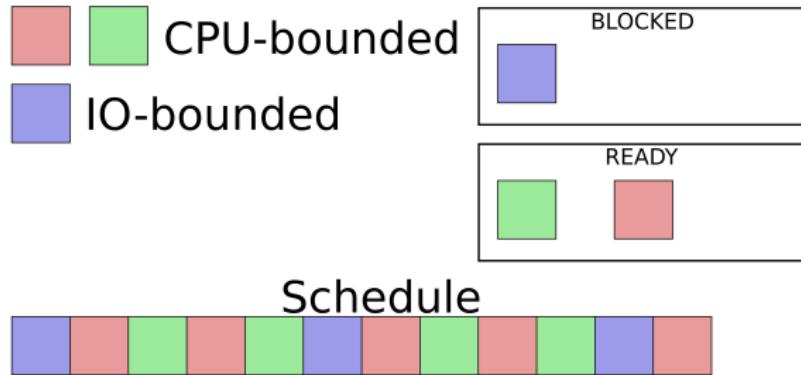
Round Robin



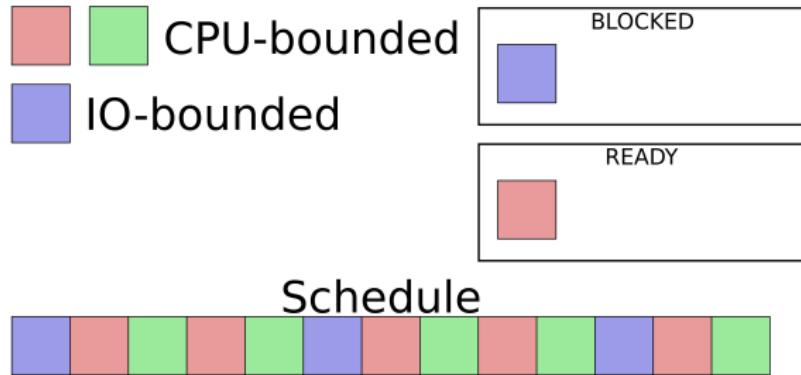
Round Robin



Round Robin



Round Robin



Выбор кванта времени

- ▶ Из каких соображений стоит выбирать квант времени?
 - ▶ чем больше квант
 - ▶ тем меньше доля времени на переключение;
 - ▶ тем больше время отклика;
 - ▶ чем меньше квант
 - ▶ тем больше доля времени на переключение;
 - ▶ тем меньше время отклика.

Выбор кванта времени



Планировщик Windows

- ▶ Потокам в Windows назначен приоритет
 - ▶ приоритет состоит из класса и приоритета внутри класса.
- ▶ На исполнение выбирается поток с наивысшим приоритетом
 - ▶ для потоков с равными приоритетами используется RR.

Priority Boost

- ▶ Чтобы избежать неограниченного голодания потоков, Windows иногда повышает им приоритет:
 - ▶ если поток отвечает за видимую часть UI;
 - ▶ при получении ввода или завершении операции IO;
 - ▶ для "случайно" выбранных потоков.

Планировщик Linux (один из)

- ▶ Completely Fair Scheduler (CFS) - честный планировщик:
 - ▶ для каждого потока поддерживается "виртуальное время";
 - ▶ "виртуальное время" увеличивается, когда поток работает;
 - ▶ CPU отдаем потоку с наименьшим "виртуальным временем".

Операционные Системы

Синхронизация потоков

January 5, 2019

Пример

```
1      .data
2 counter:
3      .int 0
4
5      .text
6 add_one:
7      movq counter, %rax
8      inc %rax
9      movq %rax, counter
0      retq
```

```
1  int counter;
2
3  int add_one(void)
4  {
5      return ++counter;
6 }
```

Вариант 1

```
1 movq counter, %rax      1
2 inc %rax                2
3 movq rax, counter       3
4                         4 movq counter, %rax
5                         5 inc %rax
6                         6 movq rax, counter
```

Вариант 2

```
1 movq counter, %rax  
2  
3 inc %rax  
4 movq rax, counter  
5  
6
```

```
1 movq counter, %rax  
2  
3  
4  
5 inc %rax  
6 movq rax, counter
```

Состояние гонки

- ▶ Состояние гонки - результат зависит от порядка выполнения инструкций
 - ▶ порядок зависит от слишком многих факторов;
 - ▶ решения планировщика, влияние других потоков, прерывания...
 - ▶ могут быть трудно воспроизводимы - не поддаются тестированию.

Критическая секция

- ▶ Критическая секция
 - ▶ участок кода, обращающийся к разделяемым несколькими потоками данным;
 - ▶ если не более, чем один поток может одновременно находиться в критической секции, то не будет состояния гонки.

Блокировка

- ▶ Блокировка (lock) - некоторый объект и пара методов для работы с ним
 - ▶ lock - метод захвата блокировки;
 - ▶ unlock - метод освобождения блокировки.

Свойство взаимного исключения

- ▶ Взаимное исключение (mutual exclusion)
 - ▶ потоки всегда вызывают lock и unlock парами (сначала lock, а потом unlock);
 - ▶ не более одного потока может одновременно находиться между lock-ом и unlock-ом.

Свойство взаимного исключения

```
1      struct lock l;
2      int counter;
3
4      int add_one(void)
5      {
6          int res;
7
8          lock(&l);
9          res = ++counter;
10         unlock(&l);
11     }
```

Свойство взаимного исключения

```
1 struct lock lock0;          1 struct lock lock1;
2 int counter0;              2 int counter1;
3                                         3
4 int add_one0(void)          4 int add_one1(void)
5 {                           5 {
6     int res;                6     int res;
7     lock(&lock0);          7
8     res = ++counter0;      8     lock(&lock1);
9     unlock(&lock0);       9     res = ++counter1;
0     return res;           10    unlock(&lock1);
1 }                           11    return res;
2 }                           12 }
```

Свойство живости

```
1      struct lock {
2  };
3
4      void lock(struct lock *unused)
5  {
6          (void) unused;
7          while (1);
8      }
9
10     void unlock(struct lock *unused)
11  {
12     (void) unused;
13 }
```

Свойство живости

- ▶ Свойство живости (deadlock freedom)
 - ▶ если один из потоков вызвал `lock`, то какой-то из потоков, вызвавших `lock`, захватит блокировку;
 - ▶ поток не ждет в `lock`, если он единственный пытается захватить блокировку;
 - ▶ если поток ждет, значит другому потоку повезло захватить блокировку.

На что нельзя полагаться?

- ▶ Скорость работы потоков:
 - ▶ мы не знаем, сколько времени потребуется потоку, чтобы выполнить какой-то код;
 - ▶ мы не можем полагать, что какой-то поток быстрее.

На что можно полагаться?

- ▶ Потоки работают корректно:
 - ▶ поток не находится между lock и unlock бесконечно;
 - ▶ поток не "падает", находясь между lock и unlock;
 - ▶ и так далее...

Атомарный Read/Write регистр

- ▶ Атомарный RW регистр - ячейка памяти и пара операций
 - ▶ write - "атомарно" записывает значение в регистр;
 - ▶ read - "атомарно" читает последнее записанное значение;
 - ▶ все операции (read/write) упорядочены.

Взаимное исключение для 2-х потоков

- ▶ Есть всего два потока
 - ▶ потоки имеют идентификаторы 0 и 1;
 - ▶ внутри потока мы можем узнать его идентификатор (пусть за это отвечает функция `threadId`).

Альтернатия

```
1      struct lock {
2          atomic_int last;
3      };
4
5      void lock_init(struct lock *lock)
6      {
7          atomic_store(&lock->last, 0);
8      }
9
10     void lock(struct lock *lock)
11     {
12         while (atomic_load(&lock->last) == threadId());
13     }
14
15     void unlock(struct lock *lock)
16     {
17         atomic_store(&lock->last, threadId());
18     }
```

Свойство взаимного исключения

- ▶ Для приведенного алгоритма взаимное исключение гарантируется
 - ▶ lock может вернуть управление только потоку с идентификатором, не равным `lock->last`;
 - ▶ только поток с `threadId() != lock->last` может изменить значение `lock->last`.

Свойство живости

- ▶ Пусть поток 1 вообще никогда не пытается захватить лок
 - ▶ если поток 0 вызовет `lock`, то он зависнет навсегда;
 - ▶ т. е. свойство живости не выполняется.

Флаги намерения

```
1     struct lock {
2         atomic_int flag [2];
3     };
4
5     void lock_init(struct lock *lock)
6     {
7         atomic_store(&lock->flag[0], 0);
8         atomic_store(&lock->flag[1], 0);
9     }
10
11    void lock(struct lock *lock)
12    {
13        const int me = threadId();
14        const int other = 1 - me;
15
16        atomic_store(&lock->flag[me], 1);
17        while (!atomic_load(&lock->flag[other]));
18    }
19
20    void unlock(struct lock *lock)
21    {
22        const int me = threadId();
23
24        atomic_store(&lock->flag[me], 0);
25    }
```

Корректность

- ▶ Гарантируется ли взаимное исключение?
- ▶ Гарантируется ли живость?

Алгоритм Петерсона для 2-х потоков

```
1      struct lock {
2          atomic_int last;
3          atomic_int flag[2];
4      };
5
6      void lock(struct lock *lock)
7      {
8          const int me = threadId();
9          const int other = 1 - me;
10
11         atomic_store(&lock->flag[me], 1);
12         atomic_store(&lock->last, me);
13
14         while (atomic_load(lock->flag[other])
15                 && atomic_load(&lock->last) == me);
16     }
17
18     void unlock(struct lock *lock)
19     {
20         const int me = threadId();
21
22         atomic_store(&lock->flag[me], 0);
23     }
```

Взаимное исключение

- ▶ Доказательство от противного - пусть два потока одновременно находятся в критической секции
 - ▶ оба потока записывали значение в атомарный регистр `last`;
 - ▶ один из них должен был быть первым, а другой последним;
 - ▶ для определенности пусть последним был поток 1.

Взаимное исключение

- ▶ Итак нам известно следующее:
 - ▶ $lock \rightarrow last == 1$ - последним туда записал поток 1;
 - ▶ $lock \rightarrow flag[0] = 1$ и $lock \rightarrow flag[1] == 1$.

Взаимное исключение

- ▶ Как в таких условиях поток 1 мог пройти мимо цикла в lock и войти в критическую секцию?
 - ▶ очевидно, никак.

Живость

- ▶ Пусть поток 0 пытается войти в критическую секцию, возможны две ситуации:
 - ▶ при проверке условия цикла $lock \rightarrow flag[1] == 0$;
 - ▶ при проверке условия цикла $lock \rightarrow flag[1] == 1$.

Живость

- ▶ В первом случае ($\text{lock} \rightarrow \text{flag}[1] == 0$)
 - ▶ поток 1 даже не пытался захватить блокировку;
 - ▶ условие цикла, очевидно, ложно, и поток 0 входит в критическую секцию

Живость

- ▶ Во втором случае ($lock \rightarrow flag[1] == 1$)
 - ▶ оба потока изъявили намерение войти в критическую секцию;
 - ▶ нужно показать, что хотя бы один из них рано или поздно войдет в критическую секцию (или уже там).

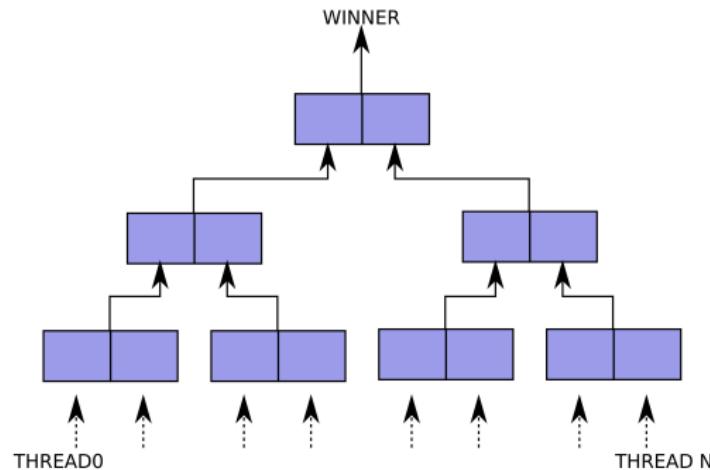
Живость

- ▶ Оба потока после записи в `lock->flag[x]` должны в какой-то момент записать в `lock->last`
 - ▶ не трудно увидеть, что если `lock->flag[0] == 1` и `lock->flag[1] == 1`,
 - ▶ то тот из них, кто сделал это первым, войдет в критическую секцию.

N потоков

- ▶ Реализовав взаимное исключение для 2-х потоков, мы можем реализовать взаимное исключение для любого числа потоков
 - ▶ организуем турнир для N потоков;
 - ▶ потоки конкурируют друг с другом на "выбывание".

N потоков



Алгоритм Петерсона для N потоков

```
1     struct lock_one {
2         atomic_int last;
3         atomic_int flag[N];
4     };
5
6     int flags_clear(const struct lock_one *lock)
7     {
8         const int me = threadId();
9
10        for (int i = 0; i != N; ++i) {
11            if (i != me && atomic_load(&lock->flag[i]))
12                return 0;
13        }
14        return 1;
15    }
16
17    void lock_one(struct lock_one *lock)
18    {
19        const int me = threadId();
20
21        atomic_store(&lock->flag[me], 1);
22        atomic_store(&lock->last, me);
23
24        while (!flags_clear(lock)
25               && atomic_load(&lock->last) == me);
26    }
27
28    void unlock_one(struct lock_one *lock)
29    {
30        const int me = threadId();
31
32        atomic_store(&lock->flag[me], 0);
33    }
```

Алгоритм Петерсона для N потоков

```
1     struct lock {
2         struct lock_one lock[N - 1];
3     };
4
5     void lock(struct lock *lock)
6     {
7         for (int i = 0; i != N - 1; ++i)
8             lock_one(&lock->lock[i]);
9     }
10
11    void unlock(struct lock *lock)
12    {
13        for (int i = N - 2; i >= 0; --i)
14            unlock_one(&lock->lock[i]);
15    }
```

Алгоритм Петерсона для N потоков

```
1  struct lock {
2      atomic_int level[N];
3      atomic_int last[N - 1];
4  };
5
6  void lock(struct lock *lock)
7  {
8      const int me = threadId();
9
10     for (int i = 0; i != N - 1; ++i) {
11         atomic_store(&lock->level[me], i + 1);
12         atomic_store(&lock->last[i], me);
13
14         while (!flags_clear(lock, i)
15               && atomic_load(&lock->last[i]) == me);
16     }
17 }
18
19 void unlock(struct lock *lock)
20 {
21     const int me = threadId();
22
23     atomic_store(&lock->level[me], 0);
24 }
```

Честность

- ▶ Не хочется, чтобы потоки голодали!
 - ▶ если поток захотел захватить блокировку, то когда-нибудь ему это удастся;
 - ▶ сравните с живостью - среди потоков, пытающихся захватить блокировку, одному это удастся.

Супер честность

- ▶ k-ограниченное ожидание:
 - ▶ после того как поток "изъявил" желание захватить блокировку (встал в очередь), не более k потоков могут пролезть вперед него без очереди.

Алгоритм Петерсона на примере 3 потоков

Nº	level[0]	level[1]	level[2]	last[0]	last[1]
0	0	0	0	0	0
1	1	0	0	0	0
2	1	1	0	1	0
3	1	1	1	2	0
4	2	1	1	2	0
5	0	1	1	2	0
6	1	1	1	0	0
7	1	1	2	0	2
8	1	1	0	0	2
9	1	1	1	2	2
10	2	1	1	2	0

Атомарный Read/Modify/Write регистр

- ▶ Атомарный RMW регистр позволяет за одну операцию
 - ▶ прочитать значение в регистре;
 - ▶ преобразовать некоторым образом прочитанное значение;
 - ▶ записать преобразованное значение назад.

Атомарный Read/Modify/Write регистр

```
1 int atomic_rmw(int *reg, int (*f)(int))
2 {
3     const int old = *reg;
4     const int new = f(old);
5
6     *reg = new;
7     return old;
8 }
```

Атомарный Read/Modify/Write регистр

- ▶ `atomic_exchange` - возвращает старое значение, записывает новое;
- ▶ `atomic_fetch_{add|sub|or|and|xor}` - выполняет арифметическое действие над атомарным регистром;
- ▶ `atomic_compare_exchange` - записывает новое значение, если старое значение равно заданному.

Реализация RMW регистра

- ▶ Архитектура может поддерживать RMW операции (x86 - одна из них)
 - ▶ xchg;
 - ▶ lock add, lock sub, lock or, lock and, lock xor;
 - ▶ lock cmpxchg.

Реализация RMW регистра

- ▶ Архитектура может поддерживать LL/SC (например, ARM):
 - ▶ LL (load-link, load-linked, load-locked) - загружает значение из памяти;
 - ▶ SC (store-conditional) - записывает новое значение в ячейку, но только если после LL эту ячейку никто не трогал;
 - ▶ LL/SC идут парами и работают вместе как одна RMW операция.

Взаимное исключение с использованием RWM регистра

```
1     #define LOCKED    1
2     #define UNLOCKED  0
3
4     struct lock {
5         atomic_int locked;
6     };
7
8     void lock(struct lock *lock)
9     {
10         while (atomic_exchange(&lock->locked, LOCKED) != UNLOCKED);
11     }
12
13    void unlock(struct lock *lock)
14    {
15        atomic_store(&lock->locked, UNLOCKED);
16    }
```

И снова про честность

- ▶ Что если блокировка находится под нагрузкой (high contention)?
 - ▶ т. е. блокировка практически всегда занята;
 - ▶ некоторый поток может получать CPU только тогда, когда блокировка занята;
 - ▶ такой поток будет голодать - блокировка не честная.

Ticket lock

```
1      struct lock {
2          atomic_uint ticket;
3          atomic_uint next;
4      };
5
6      void lock(struct lock *lock)
7      {
8          const unsigned ticket = atomic_fetch_add(&lock->ticket, 1);
9
10         while (atomic_load(&lock->next) != ticket);
11     }
12
13     void unlock(struct lock *lock)
14     {
15         atomic_fetch_add(&lock->next, 1);
16     }
```

И снова о прерываниях

- ▶ Пусть у нас есть устройство, которое получает данные из сети
 - ▶ устройство сигналит процессору - генерирует прерывание;
 - ▶ процессор вызывает обработчик прерывания - функцию ядра ОС;
 - ▶ обработчик прерывания должен забрать данные с устройства и положить их в буфер, из которого какой-то поток сможет их забрать.

И снова о прерываниях

- ▶ Что если к этому буферу могут обращаться из нескольких потоков?
 - ▶ мы должны защитить буфер блокировкой;
 - ▶ потоки и обработчики прерываний должны захватывать эту блокировку перед обращением к буферу;
 - ▶ что если обработчик прерывания устройства прервал поток, который захватил блокировку?

Deadlock

- ▶ Прерванный поток и обработчик прерывания ждут друг друга:
 - ▶ обработчик прерывания не может захватить блокировку, потому что ее держит прерванный поток;
 - ▶ пока обработчик прерывания не завершится, прерванный поток не получит управление и не сможет отпустить блокировку.

Мораль

- ▶ Если блокировка защищает данные, к которым обращается обработчик прерывания, то нужно выключать прерывания
 - ▶ если прерывания отключены, то deadlock между обработчиком прерывания и прерванным потоком не может возникнуть.

Однопроцессорные системы

- ▶ Представим систему с всего одним ядром/процессором
 - ▶ запретив прерывания и переключение потоков, мы получаем CPU в монопольное пользование;
 - ▶ все рассмотренные ранее алгоритмы просто не нужны.

Разделение на читателей и писателей

- ▶ Не все запросы к разделяемым данным одинаковы
 - ▶ есть запросы, которые модифицируют данные;
 - ▶ есть запросы, которые только читают данные.

Разделение на читателей и писателей

```
1      struct rwlock {
2          atomic_uint ticket;
3          atomic_uint write;
4          atomic_uint read;
5      };
6
7      void read_lock(struct rwlock *lock)
8      {
9          const unsigned ticket = atomic_fetch_add(&lock->ticket, 1);
10         while (atomic_load(&lock->read) != ticket);
11         atomic_store(&lock->read, ticket + 1);
12     }
13
14     void read_unlock(struct rwlock *lock)
15     {
16         atomic_fetch_add(&lock->write, 1);
17     }
18 }
```

Разделение на читателей и писателей

```
1      struct rwlock {
2          atomic_uint ticket;
3          atomic_uint write;
4          atomic_uint read;
5      };
6
7      void write_lock(struct rwlock *lock)
8      {
9          const unsigned ticket = atomic_fetch_add(&lock->ticket, 1);
10         while (atomic_load(&lock->write) != ticket);
11     }
12
13     void write_unlock(struct rwlock *lock)
14     {
15         atomic_fetch_add(&lock->read, 1);
16         atomic_fetch_add(&lock->write, 1);
17     }
18 }
```

Стратегии ожидания

- ▶ До сих пор функция `lock` всегда просто ждала в цикле
 - ▶ такая стратегия называется активным ожиданием;
 - ▶ блокировки, использующие активное ожидание, часто называются `spinlock`-ами;
 - ▶ они "крутятся" в цикле.

Активное ожидание

- ▶ Активное ожидание хорошо работает если:
 - ▶ потоки не держат блокировку очень долго;
 - ▶ блокировка не находится под сильной нагрузкой;
 - ▶ т. е. если активное ожидание длится недолго.

Альтернативы активному ожиданию

- ▶ Как можно ожидать не активно?
 - ▶ можно добровольно отдать CPU (переключиться на другой поток);
 - ▶ можно пометить поток как неактивный, чтобы планировщик не давал ему время на CPU, пока блокировка не будет отпущена.

Задача Producer-а и Consumer-а

- ▶ Рассмотрим следующую задачу:
 - ▶ Producer - поток/потоки, который генерирует данные;
 - ▶ Consumer - поток/потоки, который потребляет данные;
 - ▶ что если Producer и Consumer работают с разной скоростью?

Переменная состояния

- ▶ Переменная состояния (condition variable) - объект и несколько методов для работы с ним
 - ▶ wait - ожидает, пока кто-нибудь не просигналит;
 - ▶ notify_one - просигналить одному из ожидающих;
 - ▶ notify_all - просигналить всем ожидающим.

Переменная состояния

```
1 struct lock;
2 void lock(struct lock *lock);
3 void unlock(struct lock *lock);
4
5 struct condition;
6 void wait(struct condition *cv, struct lock *lock);
7 void notify_one(struct condition *cv);
8 void notify_all(struct condition *cv);
```

Producer

```
1  struct condition cv;
2  struct lock mtx;
3  int value;
4  bool valid_value;
5  bool done;
6
7  void produce(int x)
8  {
9      lock(&mtx);
10     while (valid_value)
11         wait(&cv, &mtx);
12     value = x;
13     valid_value = true;
14     notify_one(&cv);
15     unlock(&mtx);
16 }
17
18 void finish(void)
19 {
20     lock(&mtx);
21     done = true;
22     notify_all(&cv);
23     unlock(&mtx);
24 }
```

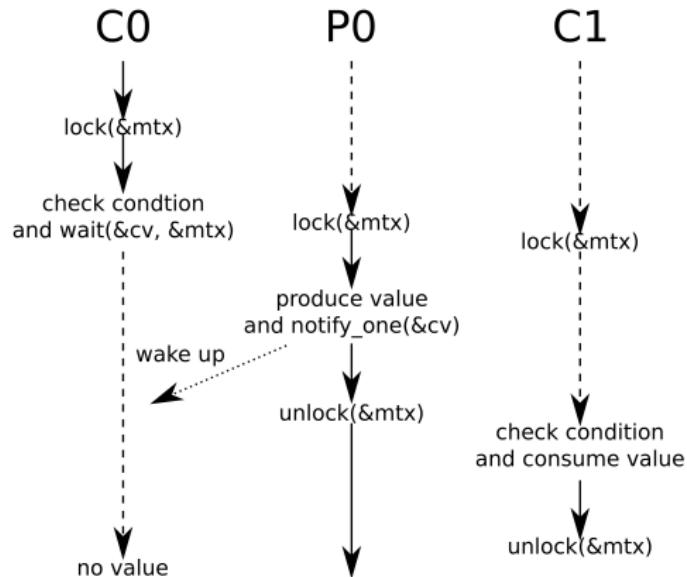
Consumer

```
1     int consume(int *x)
2 {
3     int ret = 0;
4     lock(&mtx);
5
6     while (!valid_value && !done)
7         wait(&cv, &mtx);
8
9     if (valid_value) {
10        *x = value;
11        valid_value = false;
12        notify_one(&cv);
13        ret = 1;
14    }
15    unlock(&mtx);
16    return ret;
17 }
```

Зачем нам lock?

```
1          1  /* lock(&mtx); */
2          2  while  (... && !done)
3  /* lock(&mtx); */
4  done = true;
5  notify_all(&cv);
6  /* unlock(&mtx); */
7          3
8          4
9          5
10         6
11         7      wait(&cv, &mtx);
12         8  ...
13         9  /* unlock(&mtx); */
```

Зачем нам цикл?



Зачем нам цикл?

- ▶ Spurious wakeups (ложные пробуждения) - ситуация, когда `wait` возвращает управление, даже если никто не сигналил
 - ▶ многие реализации переменной состояния подвержены:
 - ▶ C++;
 - ▶ Java;
 - ▶ POSIX Threads...

Deadlock

- ▶ Deadlock - ситуация, при которой потоки не могут работать, потому что ждут друг друга:
 - ▶ deadlock потоком исполнения и обработчиком прерывания;
 - ▶ поток А ждет, пока поток В что-то сделает (например, отпустит блокировку);
 - ▶ а поток В ничего не делает, потому что ждет, пока поток А что-то сделает (например, отпустит блокировку).

Пример

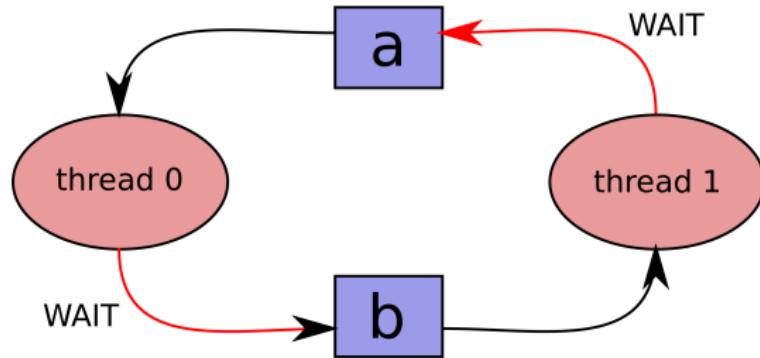
```
1 struct lock a;
2
3 void thread0(void)
4 {
5     lock(&a);
6     lock(&b);
7
8     /* do something */
9
10    unlock(&a);
11    unlock(&b);
12 }
```

```
1 struct lock b;
2
3 void thread1(void)
4 {
5     lock(&b);
6     lock(&a);
7
8     /* do something else
9      */
10    unlock(&a);
11    unlock(&b);
12 }
```

Пример

```
1      lock(&a);          1      lock(&b);
2      lock(&b);          2      lock(&b);
3      lock(&a);          3      lock(&a);
4
```

Wait-for گراف



Deadlock

- ▶ Как и с состоянием гонки, deadlock не поддается тестированию
 - ▶ появление зависит от многих факторов;
 - ▶ входные данные, решения планировщика, прерывания, производительность оборудования ...

Предотвращение deadlock-ов

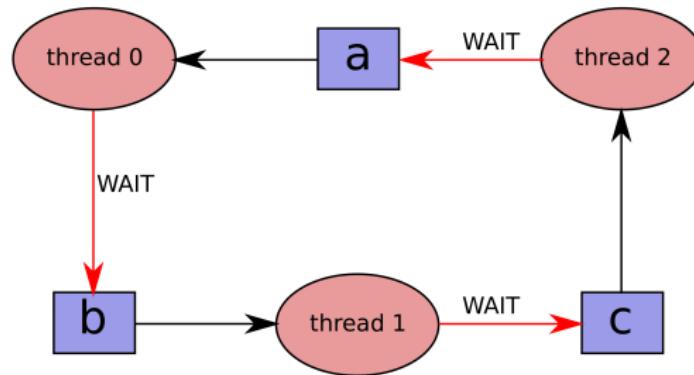
- ▶ Мы хотим избежать появления цикла в wait-for графе
 - ▶ простой случай - все блокировки известны заранее;
 - ▶ упорядочим все блокировки (например, по адресу);
 - ▶ захватываем блокировки только по порядку.

Пример

```
1 void thread0()
2 {
3     lock(&a);
4     lock(&b);
5     ...
6     unlock(&b);
7     unlock(&a);
8 }

1 void thread1()
2 {
3     lock(&b);
4     lock(&c);
5     ...
6     unlock(&c);
7     unlock(&b);
8 }
```

Пример



Пример

- ▶ Отсортируем блокировки a , b и c по алфавиту:
 - ▶ каждый поток должен захватывать блокировки только согласно порядку;
 - ▶ например, поток 2 хочет захватить блокировки c и a :
 - ▶ так как a в алфавитие раньше c , то сначала хватаем a ,
 - ▶ потом хватаем c .

Пример

```
1 void thread0()
2 {
3     lock(&a);
4     lock(&b);
5     ...
6     unlock(&b);
7     unlock(&a);
8 }

1 void thread1()
2 {
3     lock(&b);
4     lock(&c);
5     ...
6     unlock(&c);
7     unlock(&b);
8 }
```

Предотвращение deadlock-ов

- ▶ Сложный случай - все блокировки не известны заранее:
 - ▶ для этого случая придумано много различных вариантов;
 - ▶ мы рассмотрим подход, который называется Wait-Die.

Изменим интерфейс

```
1      struct wdlock_ctx {
2          unsigned long long timestamp;
3          struct wdlock *next;
4      };
5
6      struct wdlock {
7          ...
8      };
9
10     /* Grab unique "timestamp" */
11     void wdlock_ctx_init(struct wdlock_ctx *ctx);
12
13     /* This function may fail */
14     int wdlock_lock(struct wdlock *lock,
15                     struct wdlock_ctx *ctx);
16
17     /* Unlocks all of the locks */
18     void wdlock_unlock(struct wdlock_ctx *ctx);
```

Как использовать Wait-Die подход?

```
1      void thread(void)
2  {
3      struct wdlock_ctx ctx;
4
5      wdlock_ctx_init(&ctx);
6
7      while (1) {
8
9          if (!wdlock_lock(&lock1, &ctx)) {
10              wdlock_unlock(&ctx);
11              continue;
12          }
13
14          if (!wdlock_lock(&lock2, &ctx)) {
15              wdlock_unlock(&ctx);
16              continue;
17          }
18          ...
19      }
20      /* Acquired all required locks successfully,
21         can do something. */
22      wdlock_unlock(&ctx);
23 }
```

"Контекст"

- ▶ Wait-Die контекст состоит из:
 - ▶ списка захваченных блокировок;
 - ▶ уникального "timestamp".

"Контекст"

```
1  struct wdlock_ctx {
2      unsigned long long timestamp;
3      struct wdlock *next;
4  };
5
6
7  void wdlock_ctx_init(struct wdlock_ctx *ctx)
8  {
9      static atomic_ullong timestamp;
10
11     ctx->timestamp = atomic_fetch_add(&timestamp, 1) + 1;
12     ctx->next = NULL;
13 }
```

Магия timestamp

- ▶ timestamp позволяет избегать deadlock-ов
 - ▶ храним в каждой блокировке timestamp из wdlock_ctx, который использовали при захвате блокировки;
 - ▶ при попытке захватить блокировку возможно несколько вариантов:
 - ▶ если блокировка свободна, то пытаемся ее захватить - как обычно;
 - ▶ если блокировка занята, то нужно сравнить timestamp-ы.

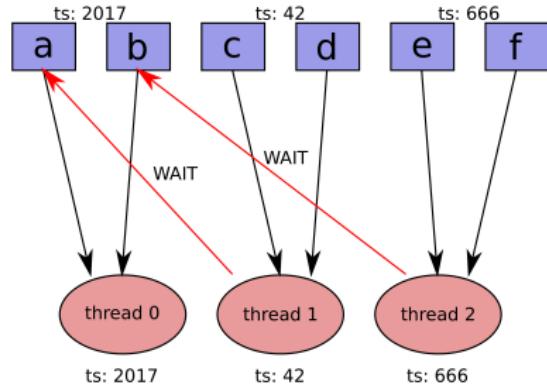
Магия timestamp

- ▶ Если блокировка захвачена, то нужно сравнить наш timestamp с сохраненным в блокировке:
 - ▶ если наш timestamp меньше, чем timestamp блокировки, то ждем;
 - ▶ в противном случае не ждем, а возвращаем признак неудачи (умираем).

Корректность

- ▶ Поток ждет на блокировке, если timestamp блокировки больше, чем timestamp потока
 - ▶ deadlock соответствует циклу в Wait-For графе;
 - ▶ при использовании Wait-Die timestampы блокировок на любом пути в графе строго возрастают;
 - ▶ следовательно, цикла в Wait-For графе быть не может.

Wait-Die γραφ



Операционные Системы

Исполняемые файлы

January 5, 2019

Первый процесс

- ▶ Ядро ОС настроило прерывания, аллокаторы, планировщик. Что дальше?
 - ▶ мы должны запустить первый процесс и первое приложение!
 - ▶ например, Linux проверяет файлы: `/sbin/init`, `/etc/init`, `/bin/init` и `/bin/sh`.

Исполняемые файлы

- ▶ Существует множество форматов исполняемых файлов:
 - ▶ ELF, a.out, PE, COM, Mach-O;
 - ▶ скрипты, начинающиеся с `#!` (sha-bang).

Заголовок исполняемого файла

- ▶ Заголовок:
 - ▶ magic number/string - позволяет быстро определить формат файла;
 - ▶ различного рода флаги и параметры:
 - ▶ версия формата исполняемого файла;
 - ▶ архитектура;
 - ▶ ссылки на другие части файла.

Заголовок ELF файла

```
1 struct elf64_hdr {
2     uint8_t e_ident[16];
3     uint16_t e_type;
4     uint16_t e_machine;
5     uint32_t e_version;
6     uint64_t e_entry;
7     uint64_t e_phoff;
8     uint64_t e_shoff;
9     uint32_t e_flags;
10    uint16_t e_ehsize;
11    uint16_t e_phentsize;
12    uint16_t e_phnum;
13    uint16_t e_shentsize;
14    uint16_t e_shnum;
15    uint16_t e_shstrndx;
16 } __attribute__((packed));
```

Точка входа

- ▶ У любой программы есть первая инструкция - точка входа:
 - ▶ формат исполняемого файла явно или не явно указывает адрес первой инструкции;
 - ▶ ОС после загрузки исполняемого файла передает управление первой инструкции;
 - ▶ обычно передача управления сопровождается понижением уровня привилегий кода (переходом в userspace).

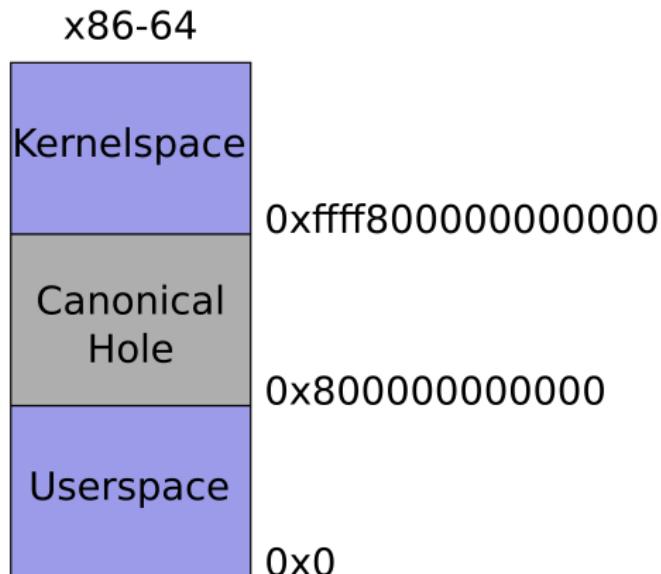
Заголовок ELF файла

```
1      struct elf64_hdr {
2          uint8_t e_ident[16];
3          uint16_t e_type;
4          uint16_t e_machine;
5          uint32_t e_version;
6
7          /* Logical address of the first instruction */
8          uint64_t e_entry;
9
10         uint64_t e_phoff;
11         uint64_t e_shoff;
12         uint32_t e_flags;
13         uint16_t e_ehsize;
14         uint16_t e_phentsize;
15         uint16_t e_phnum;
16         uint16_t e_shentsize;
17         uint16_t e_shnum;
18         uint16_t e_shstrndx;
19     } __attribute__((packed));
```

Описание адресного пространства

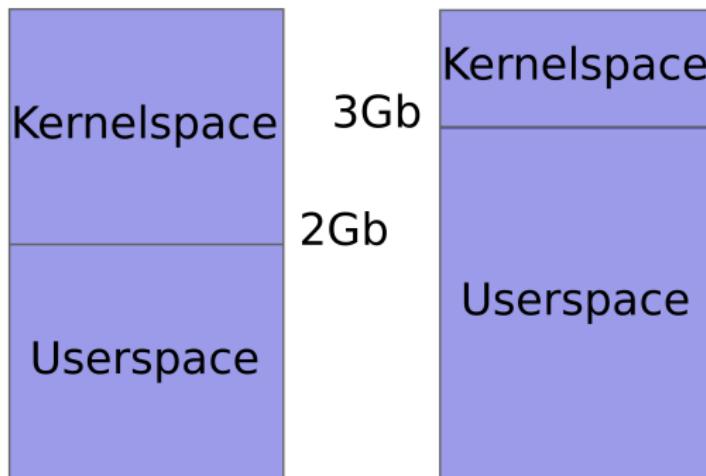
- ▶ Формат исполняемого файла описывает логическое адресное пространство:
 - ▶ какие участки логического адресного пространства нужны и для чего;
 - ▶ где в памяти процесса должны располагаться код и данные;
 - ▶ где в исполняемом файле хранятся код и данные.

Типичное адресное пространство

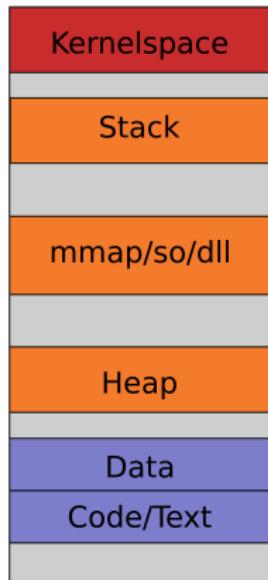


Типичное адресное пространство

x86-32



Типичное адресное пространство



Elf Program Headers

```
1  struct elf64_hdr {
2      uint8_t e_ident[16];
3      uint16_t e_type;
4      uint16_t e_machine;
5      uint32_t e_version;
6      uint64_t e_entry;
7
8      /* Offset of the program header table */
9      uint64_t e_phoff;
10
11     uint64_t e_shoff;
12     uint32_t e_flags;
13     uint16_t e_ehsize;
14
15     /* The size of a program header table entry */
16     uint16_t e_phentsize;
17     /* The number of entries in the program
18         header table */
19     uint16_t e_phnum;
20
21     uint16_t e_shentsize;
22     uint16_t e_shnum;
23     uint16_t e_shstrndx;
24 } __attribute__((packed));
```

Elf Program Headers

```
1  struct elf64_phdr {
2      /* There are different types of segments,
3         we need PT_LOAD == 1 */
4      uint32_t p_type;
5
6      /* Read/Write/Execute */
7      uint32_t p_flags;
8
9      /* Offset of the segment in the file */
10     uint64_t p_off;
11
12     /* The logical address of the segment in memory */
13     uint64_t p_vaddr;
14     uint64_t p_paddr;
15
16     /* The size of the file image of the segment */
17     uint64_t p_filesz;
18
19     /* The size of the memory image of the segment */
20     uint64_t p_memsz;
21
22     uint64_t p_align;
23 } __attribute__((packed));
```

Загрузка исполняемого файла

- ▶ Подготовить адресное пространство согласно описанию в файле
 - ▶ аллоцировать память и настроить таблицы страниц;
 - ▶ скопировать код и данные из файла в память;
 - ▶ возможно создать стек отдельно.
- ▶ "Прыгнуть" в userspace
 - ▶ передать управление точке входа, указанной в файле;
 - ▶ возможно, понизить уровень привилегий.

Библиотеки

- ▶ Виды библиотек:
 - ▶ статические - становятся частью исполняемого файла;
 - ▶ динамические - хранятся отдельно от исполняемого файла
 - ▶ загружаются при запуске приложения или по требованию.

Динамические библиотеки

- ▶ Особенность динамических библиотек - могут быть загружены по разным адресам
 - ▶ как код библиотеки обращается к своим коду и данным?
 - ▶ как код приложения обращается к библиотеке?
 - ▶ как код библиотек обращается к коду и данным других библиотек?

Компоновщик

- ▶ Компоновщик (linker, link editor) - программа, которая "связывает" бинарные файлы вместе и генерирует исполняемый файл
 - ▶ в момент компиляции адреса функций/переменных могут быть не известны;
 - ▶ компилятор просто оставляет "пустое место", а компоновщик записывает в него адрес.

Динамический компоновщик

- ▶ Адреса функций/переменных из динамических библиотек не известны
 - ▶ компилятор/статический компоновщик оставляют "пустые места";
 - ▶ динамический компоновщик должен записать в них адреса, после того как библиотека была загружена.

Загрузка ELF файла с динамическими библиотеками

- ▶ ELF файл загружается как обычно
 - ▶ ищем Program Header-ы с типом PT_LOAD и загружаем их в память.
- ▶ Смотрим в Program Header с типом PT_INTERP
 - ▶ там хранится имя файла динамического компоновщика;
 - ▶ загружаем его в память в дополнение к программе.
- ▶ Передаем управление динамическому компоновщику.

Поиск динамических библиотек

- ▶ Исполняемый файл должен хранить информацию о динамических библиотеках
 - ▶ например, ELF Program Header с типом PT_DYNAMIC указывает, где в файле хранится эта информация;
 - ▶ динамический компоновщик загружает все требуемые зависимости в память.

Редактирование связей

- ▶ Исполняемый файл и динамические библиотеки хранят список обращений к внешним сущностям
 - ▶ вызовы функций из других (и не только) библиотек;
 - ▶ обращения к переменным (и не только) из других библиотек;
 - ▶ динамический компоновщик находит адреса и записывает их в определенные места в памяти.

GOT

- ▶ ELF формат использует Global Offset Table (GOT)
 - ▶ код, обращающийся к переменной, знает относительный адрес GOT и номер записи в ней, соответствующей этой переменной;
 - ▶ компилятор генерирует код, который берет адрес из GOT;
 - ▶ динамический компоновщик записывает в GOT правильные адреса при загрузке.

PLT

- ▶ ELF формат также использует Procedure Linkage Table (PLT)
 - ▶ код обращающийся к функции знает относительный адрес PLT и номер "заглушки" в ней, соответствующей этой переменной;
 - ▶ компилятор генерирует код, который вызывает "заглушку" из PLT вместо реальной функции;
 - ▶ динамический компоновщик может изменять PLT, а может изменять GOT, к которой "заглушка" из PLT обращается.

Операционные Системы

Системные вызовы

January 5, 2019

Системные вызовы

- ▶ Системные вызовы - интерфейс между userspace и ядром ОС
 - ▶ пользовательский код не имеет достаточно привилегий, чтобы вызывать код ядра как обычные функции;
 - ▶ системный вызов сопровождается повышением привилегий;
 - ▶ возврат из системного вызова сопровождается понижением привилегий.

Реализация системных вызовов

- ▶ Как реализовать интерфейс системных вызовов?
 - ▶ способ, как обычно, зависит от архитектуры;
 - ▶ например, в x86 существуют инструкции `syscall` и `sysenter`;
 - ▶ но мы посмотрим на другой вариант (более старый).

И снова о прерываниях...

- ▶ Что происходит, если обработчик прерывания прерывает пользовательский код?
 - ▶ вызывается обработчик прерывания - код ядра;
 - ▶ обработчик прерывания выполняется уже в привилегированном режиме.

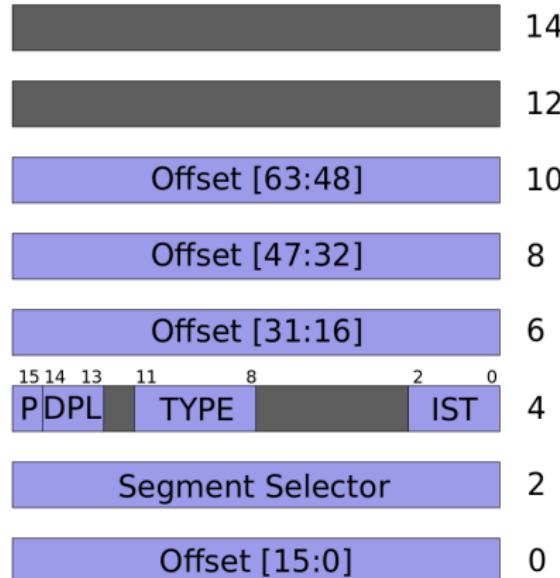
Программные прерывания

- ▶ Прерывания можно вызывать программно (и не только сделав ошибку)
 - ▶ например, в x86 для этого существует специальная инструкция `int`, номер прерывания - параметр инструкции;
 - ▶ выберем запись в IDT и будем использовать ее для системных вызовов.

Программные прерывания в x86

- ▶ С помощью инструкции `int` в x86 можно генерировать прерывание с любым номером
 - ▶ в том числе и соответствующие исключениям;
 - ▶ в том числе и соответствующие аппаратным прерываниям;
 - ▶ приложения могут натворить бед, если разрешить им генерировать прерывания как попало.

Дескриптор IDT, поле DPL



Дескриптор IDT, поле DPL

- ▶ DPL дескриптора системного вызова выставляем в 3
 - ▶ благодаря чему непrivилегированный код может генерировать это прерывание.
- ▶ DPL всех остальных дескрипторов выставляем в 0.

Стек обработчика прерывания

- ▶ При вызове обработчика на стек сохраняются адрес возврата и прочее
 - ▶ на какой стек все это будет сохранено?
 - ▶ не хочется использовать стек непrivилегированного кода
 - ▶ там может быть не достаточно места;
 - ▶ пользовательский код может делать со своим стеком все что угодно.

Отдельный стек для ядра

- ▶ Мы хотим использовать отдельный стек для ядра и отдельный для userspace
 - ▶ например, в Linux для каждого потока создается стек ядра, т. е. у каждого потока есть 2 стека;
 - ▶ при прерываниях и системных вызовах происходит переключение на стек ядра потока.

Task State Segment

- ▶ TSS (Task State Segment) - структура, которая хранит указатель стека, который будет загружен в RSP
 - ▶ ранее (в 32-битном режиме) могла быть использована для хранения состояния потока.

"Прыжок" в userspace

I/O Map Base
IST7 [63:32]
IST7 [31:0]
ISTi [63:32]
ISTi [31:0]
IST1 [63:32]
IST1 [31:0]
RSP0 [63:32]
RSP0 [31:0]
RSP1 [63:32]
RSP1 [31:0]
RSP0 [63:32]
RSP0 [31:0]

Task State Segment

- ▶ "Указание" на TSS хранится в специальном регистре TR
 - ▶ инструкция LTR записывает значение в TR, а инструкция STR читает;
 - ▶ для использования TSS необходимо завести специальный дескриптор в GDT
 - ▶ Base и Limit хранят логический адрес и размер TSS;
 - ▶ селектор дескриптора сохраняется в TR.

Task State Segment

- ▶ Простой вариант использования TSS:
 - ▶ создаем по TSS на каждое ядро процессора - один раз, при инициализации ядра ОС
 - ▶ при переключении потоков подменяем указатель стека в TSS.

Резюме

- ▶ Подготовить дескриптор IDT, который будет использоваться для системных вызовов.
- ▶ Создать TSS:
 - ▶ создать дескриптор, описывающий TSS, в GDT;
 - ▶ загрузить селектор, ссылающийся на дескриптор, в TR.
- ▶ Не забывать подменять указатель стека в TSS при переключении потоков.

"Прыжок" в userspace

- ▶ Как передать управление в userspace в первый раз?
 - ▶ инструкция iretq завершает обработчик прерывания и передает управление, возможно, понизив уровень привилегий;
 - ▶ инструкция iretq берет свои параметры со стека - подготовим стек и вызовем iretq.

"Прыжок" в userspace

SS	RSP + 32
RSP	RSP + 24
RFLAGS	RSP + 16
CS	RSP + 8
RIP	RSP + 0

RSP + 32
RSP + 24
RSP + 16
RSP + 8
RSP + 0

Операционные Системы

InterProcess Communication

January 5, 2019

Системные вызовы

- ▶ Какие бывают системные вызовы? Сервисы ОС:
 - ▶ работа с файловыми системами;
 - ▶ работа со временем (текущее время, нотификации и прочее)
 - ▶ создание, завершение и управление процессами;
 - ▶ взаимодействие с другими процессами.

Создание процессов

- ▶ Для создания процессов в Unix-like системах используется вызов `fork`:
 - ▶ новый процесс является почти точной копией родителя;
 - ▶ вызывает `fork` один поток, а возвращаются из `fork` уже два потока в двух разных процессах.

Создание процессов

- ▶ Что если в процессе несколько потоков, и один из них вызвал fork?
 - ▶ в новом процессе будет только один поток;
 - ▶ подумайте о блокировках в новом процессе.

Уничтожение процессов

- ▶ Уничтожение процессов состоит из двух частей:
 - ▶ один из потоков в процессе должен вызвать `exit`
 - ▶ `exit` принимает целочисленный код как аргумент - код возврата.
 - ▶ родительский процесс (родной или приемный) должен дождаться завершения процесса, используя `waitpid/wait`
 - ▶ `wait/waitpid` могут вернуть код возврата, переданный в `exit`.

Уничтожение процессов

- ▶ Что если в контексте процесса работают несколько потоков?
 - ▶ exit уничтожает процесс со *всеми* его потоками.
- ▶ Что если родитель был уничтожен раньше ребенка?
 - ▶ другой процесс становится родителем.
- ▶ Что если не вызвать waitpid/wait?
 - ▶ процесс будет находиться в состоянии Zombie, пока родитель не вызовет wait/waitpid.

Запуск исполняемых файлов

- ▶ Для запуска исполняемого файла используется один из вызовов exec*:
 - ▶ при вызове exec* ядро ОС "заменяет" старое адресное пространство процесса новым;
 - ▶ передает управление точке входа исполняемого файла (или динамического компоновщика).

Запуск исполняемых файлов

- ▶ Что если в процессе несколько потоков и один из них вызвал exec*?
 - ▶ все другие потоки будут уничтожены.

Файловые дескрипторы

- ▶ В Unix *все есть файл*:
 - ▶ некоторые ресурсы, предоставляемые ОС, имеют файловый интерфейс;
 - ▶ файловый интерфейс: read/write/close.

Файловые дескрипторы

- ▶ Файловый дескриптор - некоторый идентификатор ресурса
 - ▶ в Unix - это обычно просто целое число;
 - ▶ 0 - стандартный поток ввода,
 - ▶ 1 - стандартный поток вывода,
 - ▶ 2 - стандартный поток ошибок.

Файловые дескрипторы

- ▶ Способ получения дескриптора зависит от ресурса, которому он соответствует:
 - ▶ для обычных файлов можно использовать open;
 - ▶ для каналов (pipe-ов) используют pipe;
 - ▶ есть много других функций, возвращающих файловый дескриптор.

Дублирование дескрипторов

- ▶ Иногда вам может потребоваться управлять значением файлового дескриптора
 - ▶ например, чтобы перенаправлять стандартные потоки ввода/вывода в файл/из файла;
 - ▶ для этого можно использовать вызов dup2.

Виды IPC, которые мы не будем рассматривать

- ▶ Сигналы.
- ▶ Семафоры.
- ▶ Сокеты.

Виды IPC, которые мы будем рассматривать

- ▶ Каналы (неименованные каналы).
- ▶ FIFO (именованные каналы).
- ▶ Сегменты разделяемой памяти.
- ▶ Ptrace.

Каналы (Pipes)

- ▶ Канал - односторонний канал связи между процессами (или внутри процесса)
 - ▶ создается вызовом pipe - вызов возвращает два "файловых дескриптора";
 - ▶ через один можно писать, из другого можно читать;
 - ▶ работа с pipe происходит почти как с обычным файлом: read/write/close.

Именованные каналы (FIFO)

- ▶ Почти как pipe, но у FIFO есть имя
 - ▶ создается с помощью специальной функции mknod;
 - ▶ работа с FIFO происходит почти как с обычным файлом: open/read/write/close;
 - ▶ при попытке открыть FIFO поток по умолчанию блокируется, пока кто-нибудь не откроет "другой конец" FIFO.

Сегменты разделяемой памяти

- ▶ Сегмент разделяемой памяти - участок памяти, к которому можно обращаться из нескольких процессов
 - ▶ создать сегмент можно с помощью `shmget`;
 - ▶ получить указатель на сегмент памяти можно с помощью `shmat`;
 - ▶ "закрыть" сегмент можно с помощью `shmrdt`.

Process Trace

- ▶ ptrace не является IPC в привычном понимании:
 - ▶ позволяет одному процессу следить за другим;
 - ▶ позволяет одному процессу подсмотреть в память другого;
 - ▶ позволяет одному процессу изменить память другого.