Ввод-Вывод

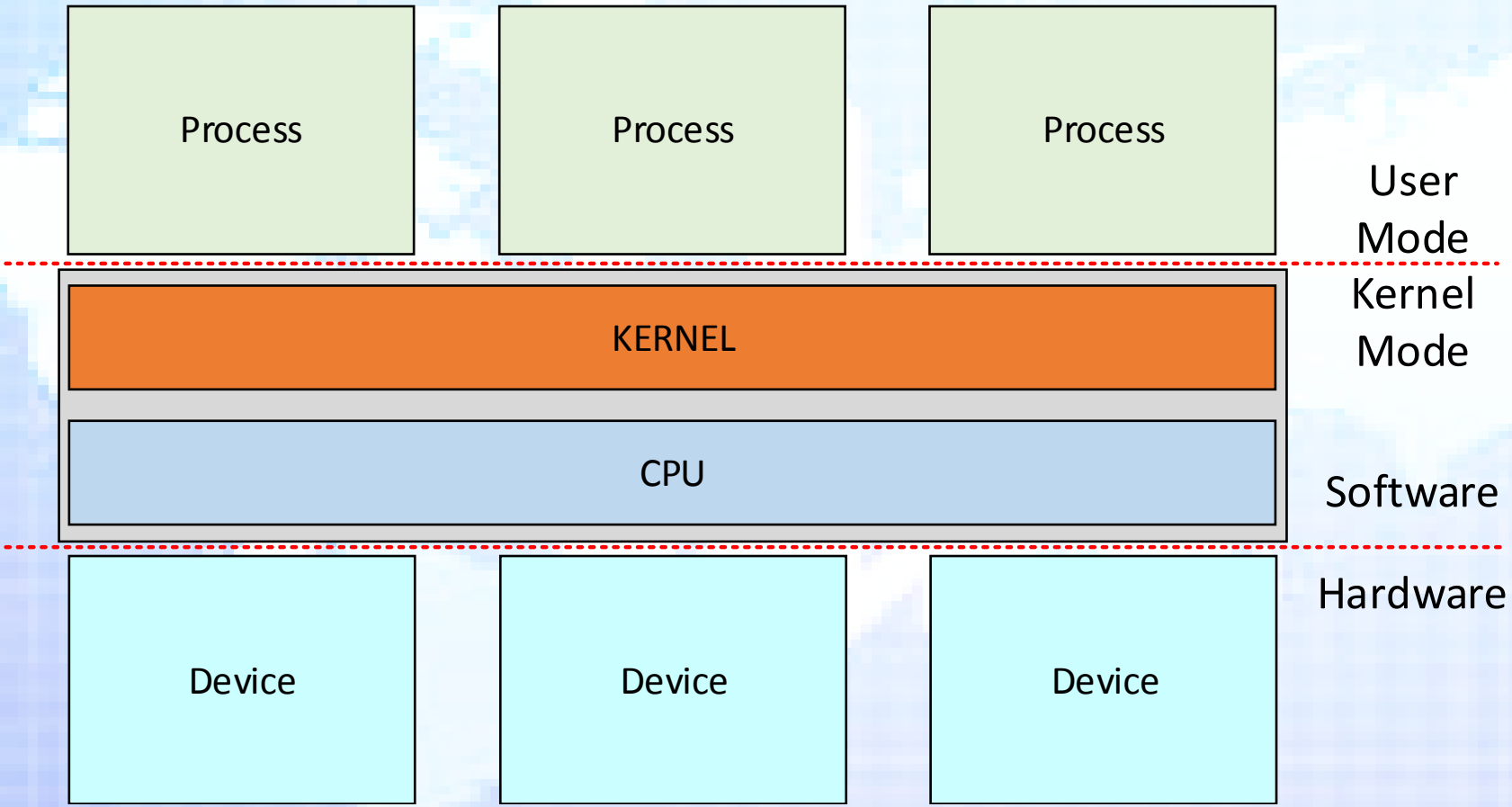
Евгений Иванович Клименков

osisp2019@gmail.com

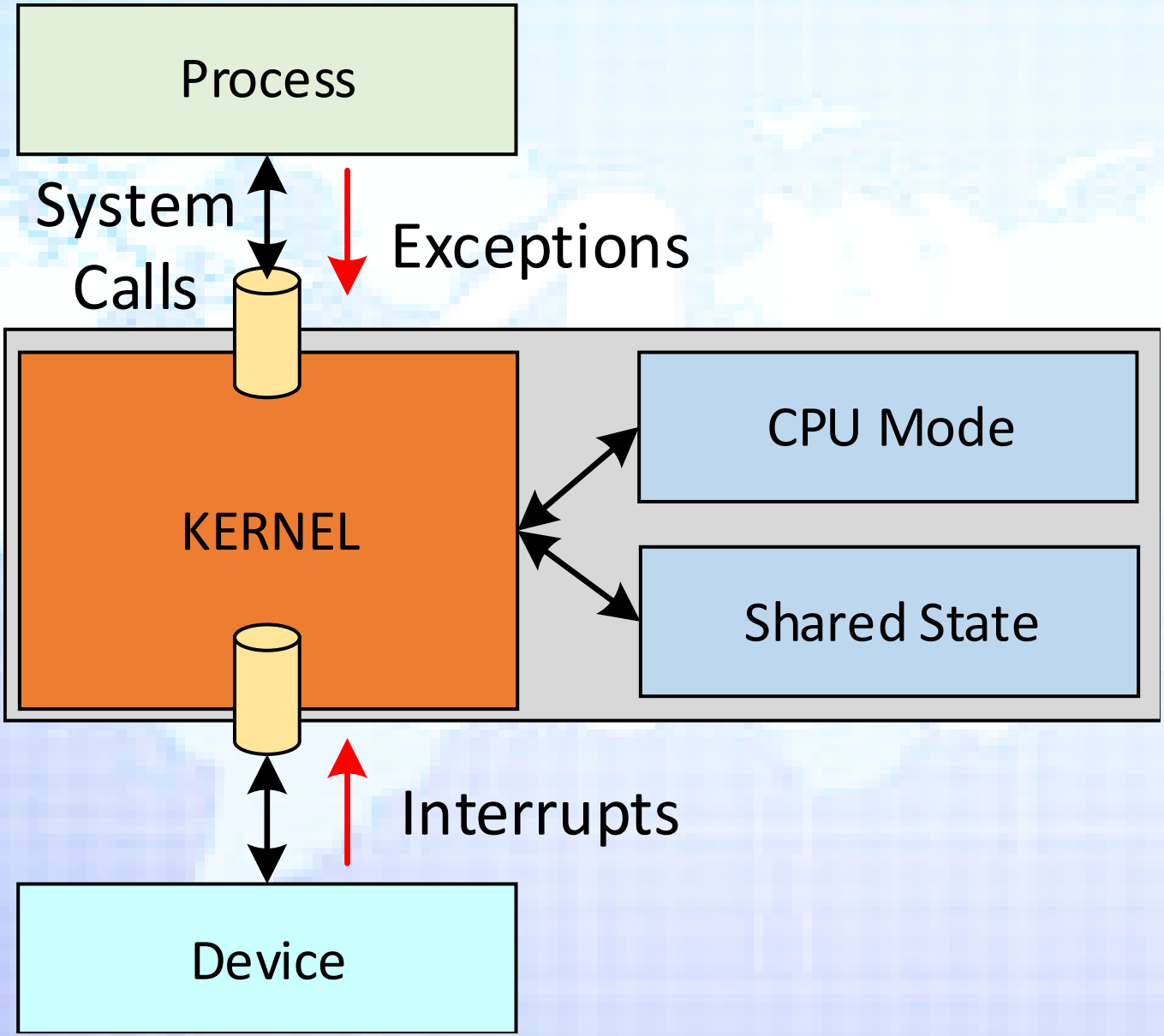
Белорусский Государственный Университет
Информатики и Радиоэлектроники

2019

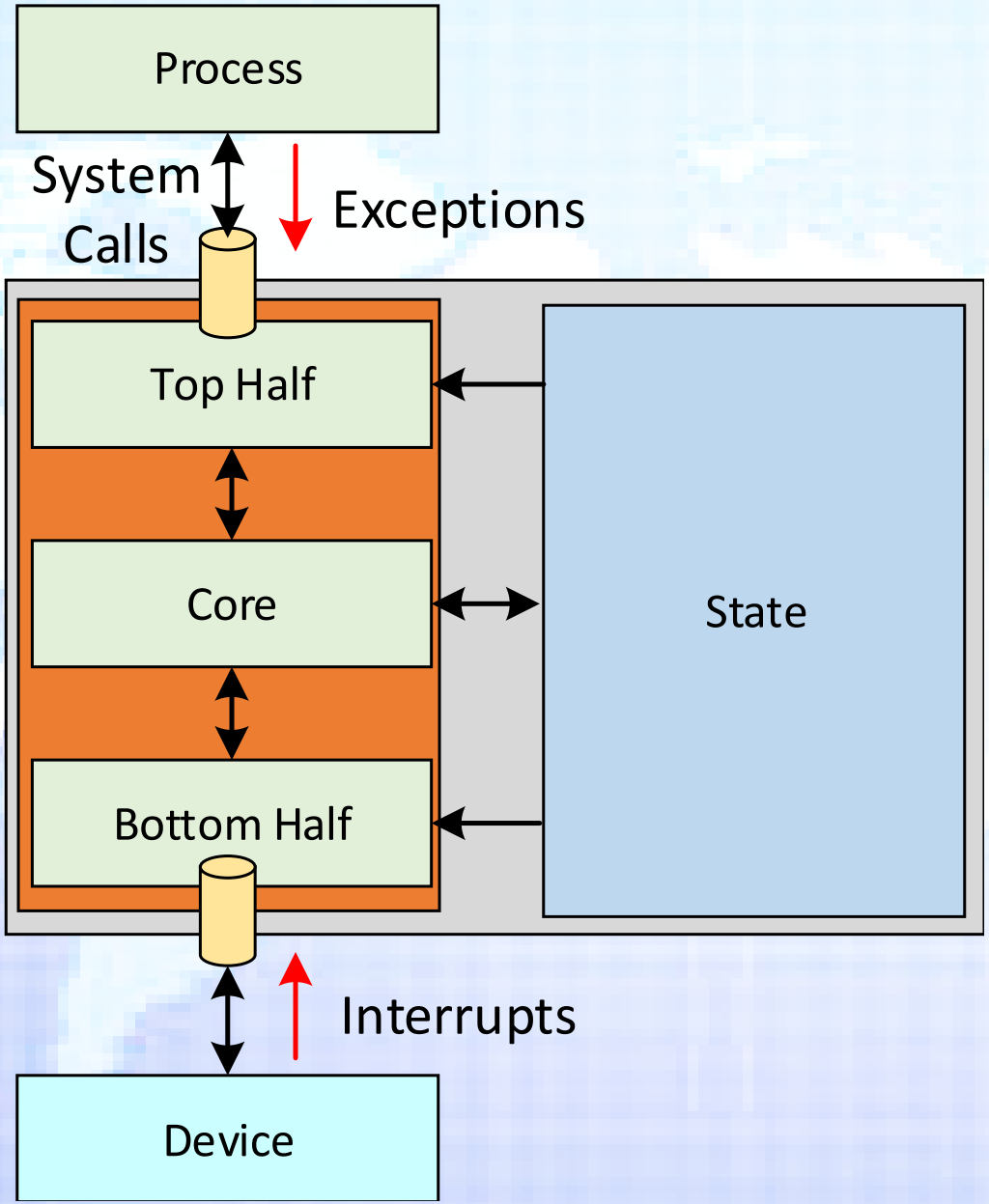
IO Model


Process
Process
Process
User Mode
KERNEL
Kernel Mode
CPU
Software
Hardware
Device
Device
Device

IO Model


Process
System Calls
Exceptions
KERNEL
CPU Mode
Shared State
Interrupts
Device

IO Model

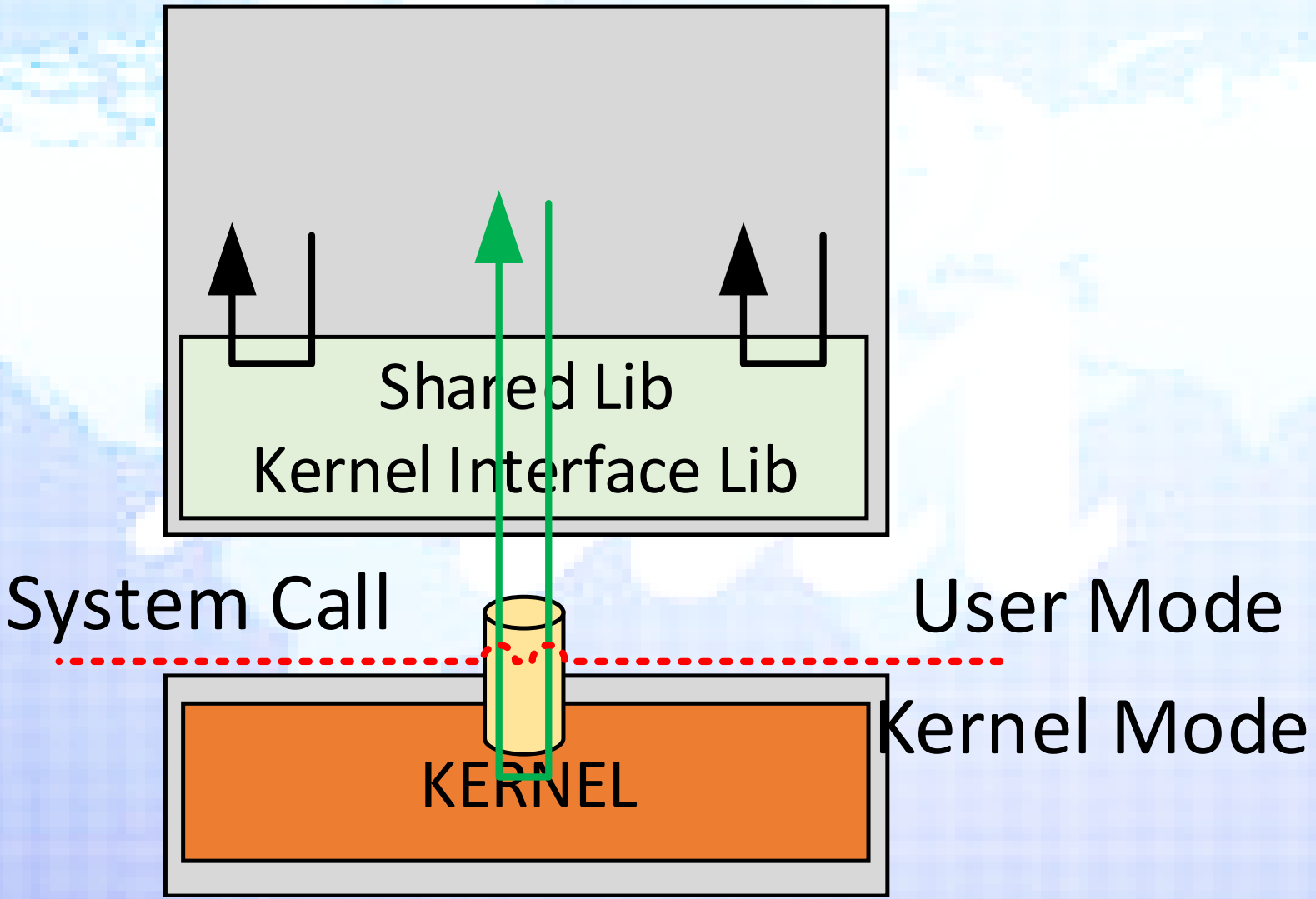

Process
System Calls
Exceptions
Top Half
Core
Bottom Half
State
Interrupts
Device

Типы входов в ядро

- Синхронные
  - Системные вызовы
  - Исключения
- Асинхронные
  - Прерывания

Системные вызовы

System Call


Shared Lib
Kernel Interface Lib
System Call
User Mode
Kernel Mode
KERNEL

System Call

Системные вызовы

- Могут рассматриваться как “специальные” функции
- Выполняются в превилигерованном режиме
- Имеют доступ к состоянию ядра
- После завершения возвращают управлению в точку программы из которой и был осуществлен вызов

Системные вызовы формируют интерфейс представляющий абстрактную машину/сервер которая реализуется ядром ОС.

- API - Прикладной программный интерфейс
- ABI - Прикладной бинарный интерфейс

System Call

Системные вызовы (API)

- Управление процессором
  - Многопроцессность
  - Многопоточность
- Управление памятью
  - Виртуальная память
- Управление вводом-выводом
  - Файловый ввод-вывод
  - Сетевые сокеты
  - Дополнительные устройства
- Решение тригометрических уравнений и в принципе любая блажь которая взбредет в голову разработчика ОС

Пример:

NAME

fork - create a new process

SYNOPSIS

```
#include <unistd.h>

pid_t fork(void);
```

DESCRIPTION

The *fork*() function shall create a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except as detailed below:

Пример: **NAME**

fork - create a new process **SYNOPSIS**

#include <unistd.h>
pid_t fork(void);

DESCRIPTION
The fork() function shall create a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except as detailed below:

ABI Пример:

NAME fork - create a new process

SYNOPSIS
IN: *EAX <- 0x34*
OUT: *EAX // according to description*

DESCRIPTION
The fork() function shall create a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except as detailed below:

System Call

- API описывает логический интерфейс ядра (сервисы предоставляемые ядром и их аргументы/возвратные значение)
- ABI = API + протокол передачи аргументов и возвратных значений на конкретном процессоре

- API is OS dependent
- ABI is OS and CPU dependent

API и ABI дают приложению описание той изолированной среды в которой функционирует процесс и о том как этот процесс может взаимодействовать с внешним миром.

API и ABI служат соглашением между разработчиками программ и разработчиками ОС.

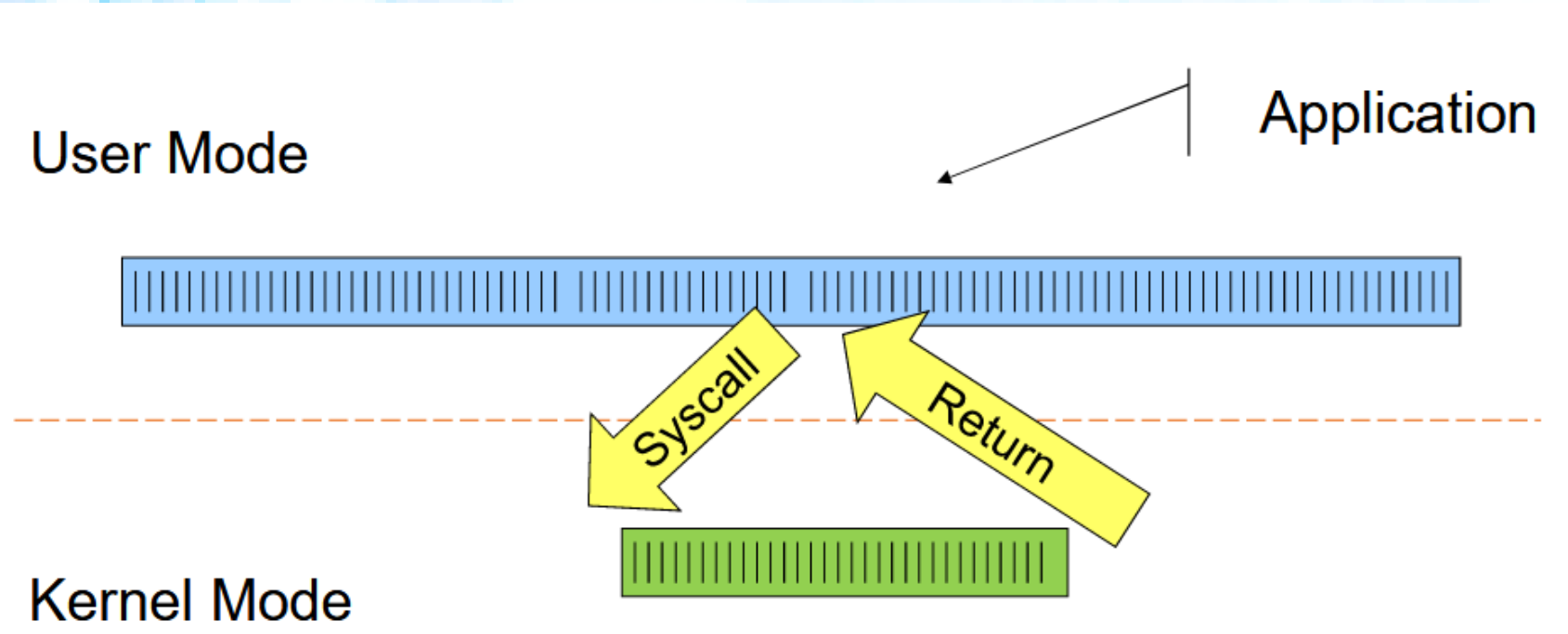
Вытеснение

Суть вытесняющего входа в ядро заключается в сохранении состояния процесса между точками входа и выхода из ядра. То есть для процесса сохраняется иллюзия того что ничего не произошло.
Состояние процесса

- Память
- Регистры процессора

Регистры процессора требуют специального обращения при вытесняющем входе в ядро, так как они используются и процессом и ядром.

Вытеснение



Механизмы системных вызовов на примере IA-32

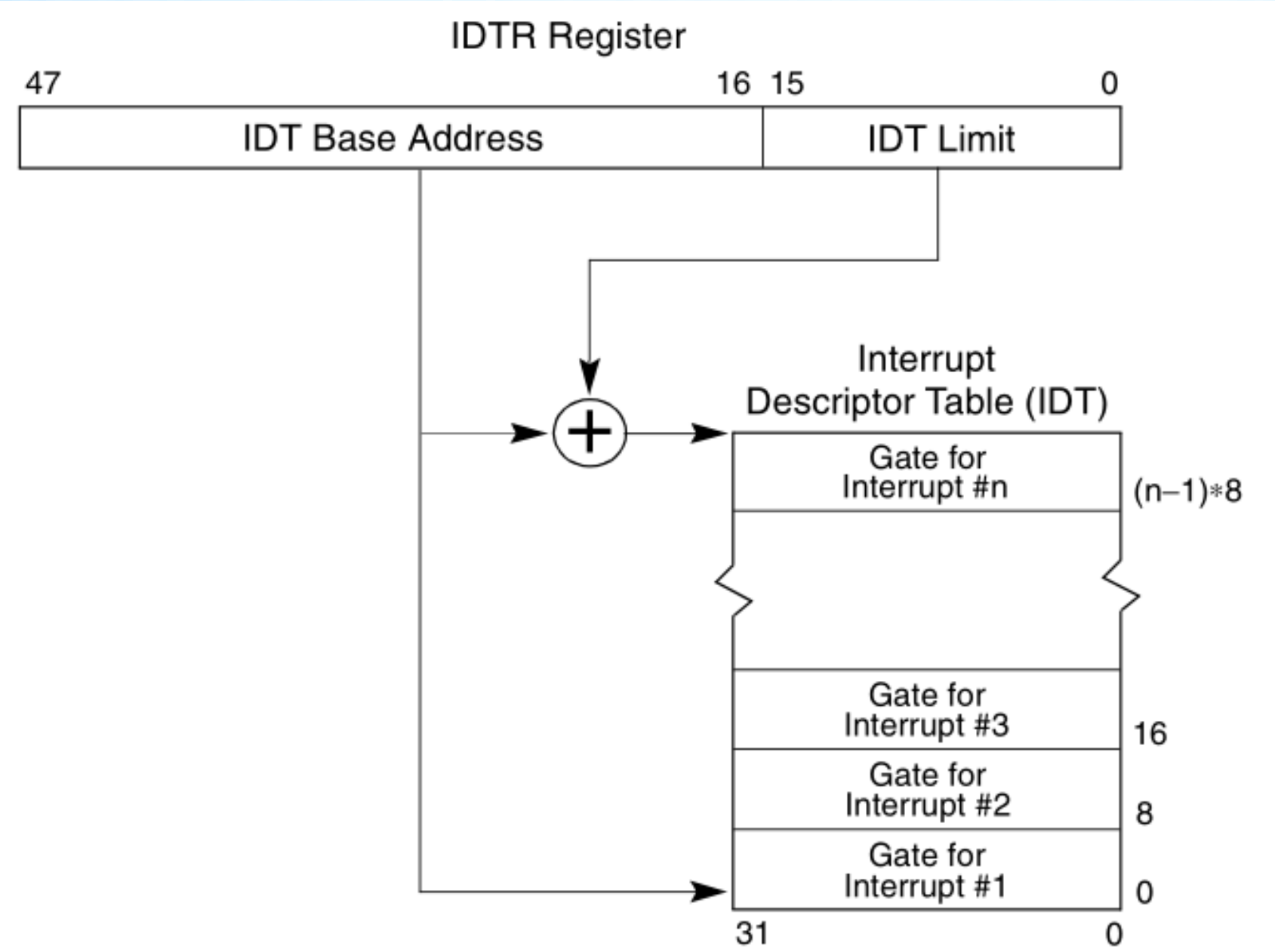
Механизмы:

- Int-based
- Syscall/Sysenter-based

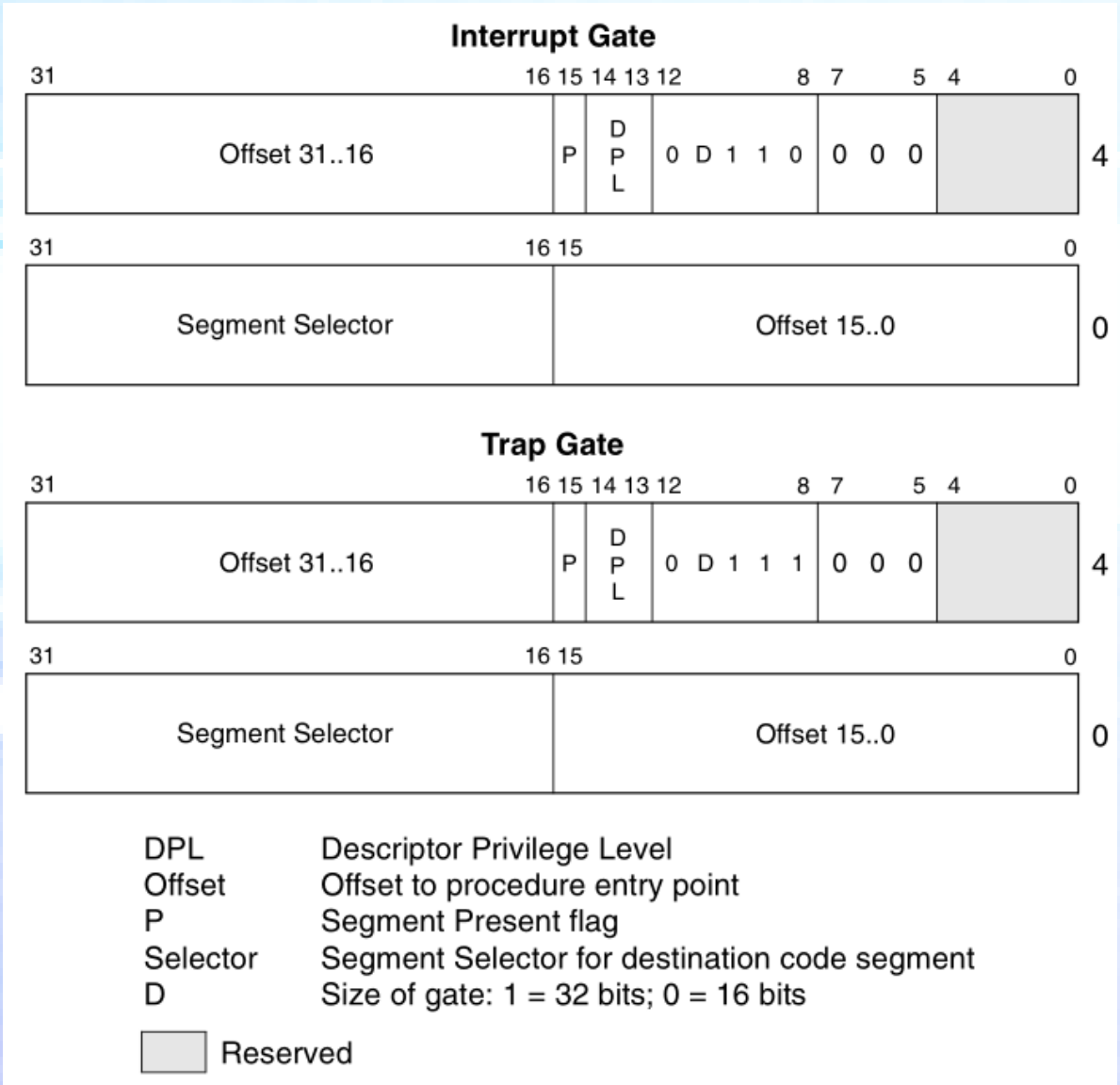
INT-Based

Специальная инструкция **INT n** позволяет генирировать так называемые программные прерывания. Например **INT 35** создает неявный вызов обработчика прерывания 35. Для возврата из системного вызова используется парная инструкция **IRET**

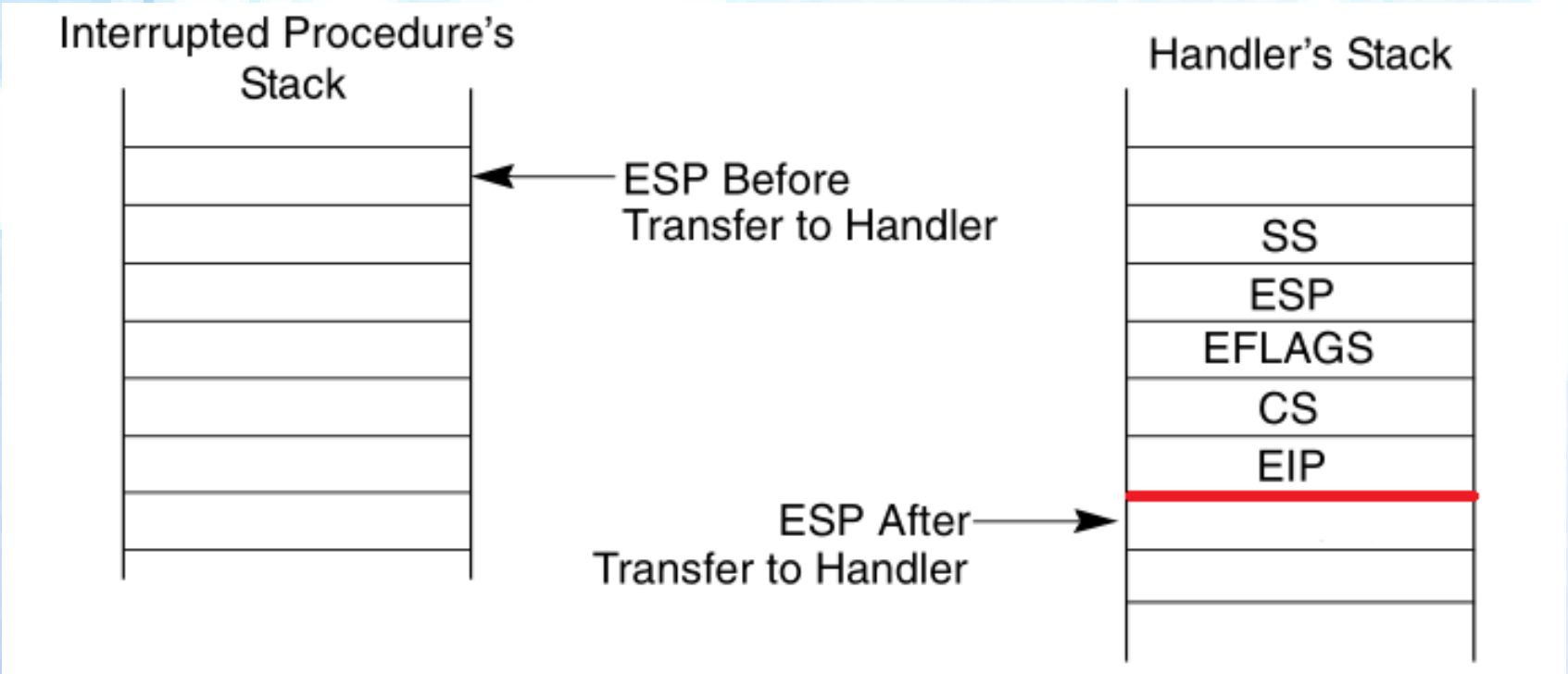
INT-Based


IDTR Register
47
16 15
0
IDT Base Address
IDT Limit
Interrupt
Descriptor Table (IDT)
Gate for
Interrupt #n
(n–1)*8
Gate for
Interrupt #3
16
Gate for
Interrupt #2
8
Gate for
Interrupt #1
0
31
0

INT-Based


Interrupt Gate
31
16 15 14 13 12
8 7
5 4
0
Offset 31..16
P
D
P
L
0 D 1 1 0
0 0 0
4
31
16 15
0
Segment Selector
Offset 15..0
0
Trap Gate
31
16 15 14 13 12
8 7
5 4
0
Offset 31..16
P
D
P
L
0 D 1 1 1
0 0 0
4
31
16 15
0
Segment Selector
Offset 15..0
0
DPL Descriptor Privilege Level
Offset Offset to procedure entry point
P Segment Present flag
Selector Segment Selector for destination code segment
D Size of gate: 1 = 32 bits; 0 = 16 bits
Reserved

INT-Based


Interrupted Procedure's Stack
Handler's Stack
ESP Before Transfer to Handler
SS
ESP
EFLAGS
CS
EIP
ESP After Transfer to Handler

INT-Based

```
IF ((vector_number « 3) + 7) is not within IDT limits
or selected IDT descriptor is not an interrupt-, trap-gate type
      THEN #GP(error_code(vector_number,1,EXT)); FI;
      (* idt operand to error_code set because vector is used *)
IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
      THEN #GP(error_code(vector_number,1,0)); FI;
      (* idt operand to error_code set because vector is used *)
      (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
IF gate not present
      THEN #NP(error_code(vector_number,1,EXT)); FI;
      (* idt operand to error_code set because vector is used *)
GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
```

INT-Based

```
TRAP-OR-INTERRUPT-GATE:
    Read new code-segment selector for trap or interrupt gate (IDT descriptor);
    IF new code-segment selector is NULL
        THEN #GP(EXT); FI; (* Error code contains NULL selector *)
    IF new code-segment selector is not within its descriptor table limits
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    Read descriptor referenced by new code-segment selector;
    IF descriptor does not indicate a code segment or new code-segment DPL > CPL
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF new code-segment descriptor is not present,
        THEN #NP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
```

INT-Based

```
(* Identify stack-segment selector for new privilege level in current TSS *)
    IF current TSS is 32-bit
        THEN
            TSSstackAddress ← (new code-segment DPL « 3) + 4;
            IF (TSSstackAddress + 5) > current TSS limit
                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
            NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);
        ELSE    (* current TSS is 16-bit *)
            TSSstackAddress ← (new code-segment DPL « 2) + 2
            IF (TSSstackAddress + 3) > current TSS limit
                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 2);
            NewESP ← 2 bytes loaded from (TSS base + TSSstackAddress);
    FI;
    IF NewSS is NULL
        THEN #TS(EXT); FI;
    IF NewSS index is not within its descriptor-table limits
    or NewSS RPL ≠ new code-segment DPL
```

INT-Based

```
(* Identify stack-segment selector for new privilege level in current TSS *)
    TSSstackAddress ← (new code-segment DPL « 3) + 4;
    IF (TSSstackAddress + 5) > current TSS limit
        THEN #TS(error_code(current TSS selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
    NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);

    IF NewSS is NULL
        THEN #TS(EXT); FI;
    IF NewSS index is not within its descriptor-table limits
    or NewSS RPL ≠ new code-segment DPL
        THEN #TS(error_code(NewSS,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    Read new stack-segment descriptor for NewSS in GDT or LDT;
    IF new stack-segment DPL ≠ new code-segment DPL
    or new stack-segment Type does not indicate writable data segment
        THEN #TS(error_code(NewSS,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF NewSS is not present
        THEN #SS(error_code(NewSS,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
```

INT-Based

```
Push(far pointer to old stack);
(* Old SS and ESP, 3 words padded to 4 *)
Push(EFLAGS);
Push(far pointer to return instruction);
(* Old CS and EIP, 3 words padded to 4 *)

CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)

IF instruction pointer from IDT gate is not within new code-segment limits
      THEN #GP(EXT); FI; (* Error code contains NULL selector *)
ESP ← NewESP;
SS ← NewSS; (* Segment descriptor information also loaded *)
```

SYSENTER-Based

Специальная инструкция **SYSENTER** переключает процессор в превилигированный режим и передает управление в заранее заданную точку входа в ядро. Для возврата из системного вызова используется парная инструкция **SYSEXIT**

SYSENTER-Based

IA32_SYSENTER_CS (MSR address 174H) — The lower 16 bits of this MSR are the segment selector for the privilege level 0 code segment. This value is also used to determine the segment selector of the privilege level 0 stack segment (see the Operation section). This value cannot indicate a null selector.

IA32_SYSENTER_EIP (MSR address 176H) — The value of this MSR is loaded into RIP (thus, this value references the first instruction of the selected operating procedure or routine). In protected mode, only bits 31:0 are loaded.

IA32_SYSENTER_ESP (MSR address 175H) — The value of this MSR is loaded into RSP (thus, this value contains the stack pointer for the privilege level 0 stack). This value cannot represent a non-canonical address. In protected mode, only bits 31:0 are loaded.

SYSENTER-Based

```
ESP ← IA32_SYSENTER_ESP[31:0];
EIP ← IA32_SYSENTER_EIP[31:0];
CS.Selector ← IA32_SYSENTER_CS[15:0] AND FFFCH;
                                        (* Operating system provides CS; RPL forced to 0 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                            (* Flat segment *)
CS.Limit ← FFFFFH;                      (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                           (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 0;
CS.P ← 1;
CS.L ← 0;
CS.D ← 1;                       (* 32-bit code segment*)
CS.G ← 1;                               (* 4-KByte granularity *)
CPL ← 0;

SS.Selector ← CS.Selector + 8;          (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                            (* Flat segment *)
SS.Limit ← FFFFFH;                      (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                            (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 0;
SS.P ← 1;
SS.B ← 1;                               (* 32-bit stack segment*)
SS.G ← 1;                               (* 4-KByte granularity *)
```

INT-Based

Практически все сиситемы производят так называемое трамплинирование INT-based точек входа. INT_80:

```
push 0x80
jmp INT_HANDLER INT_81:

push 0x81
jmp INT_HANDLER
```

После трамплинирования в стек заталкивается фиктивный код ошибки в целях унификации состояния стека. Соответственно, при выходе из ситемного вызова он выталкивается обратно.

SYSENTER-Based

SYSENTER-Based системный вызов демаскирует прерывания. После чего эмулируется то же состояние стека которое производит процессор для INT-based входа в системный вызов. Точка возврата не фиксируется процессором автоматически, поэтому системная бблиотека должна явно зафиксировать точку возврата в регистрах. При возврате стек так же очищается обработчиком системного вызова.

Обработчик системного вызова

- Сохраняет значение всех регистров на стеке
- Устанавливает Direction Flag в предопределенное значение
- Убеждается в корректном значении регистров DS и ES
- Используя номер прерывания на стеке проиводит переход на специализированный обработчик системного вызова.
- Выделяет номер системного вызова и его корректность.
- Используя номер системного вызова как индекс производит вызов функции через таблицу обработчиков системных вызов передавая указатель на кадр системного вызова на стеке.
- Восстанавливает значение всех регистров из стека

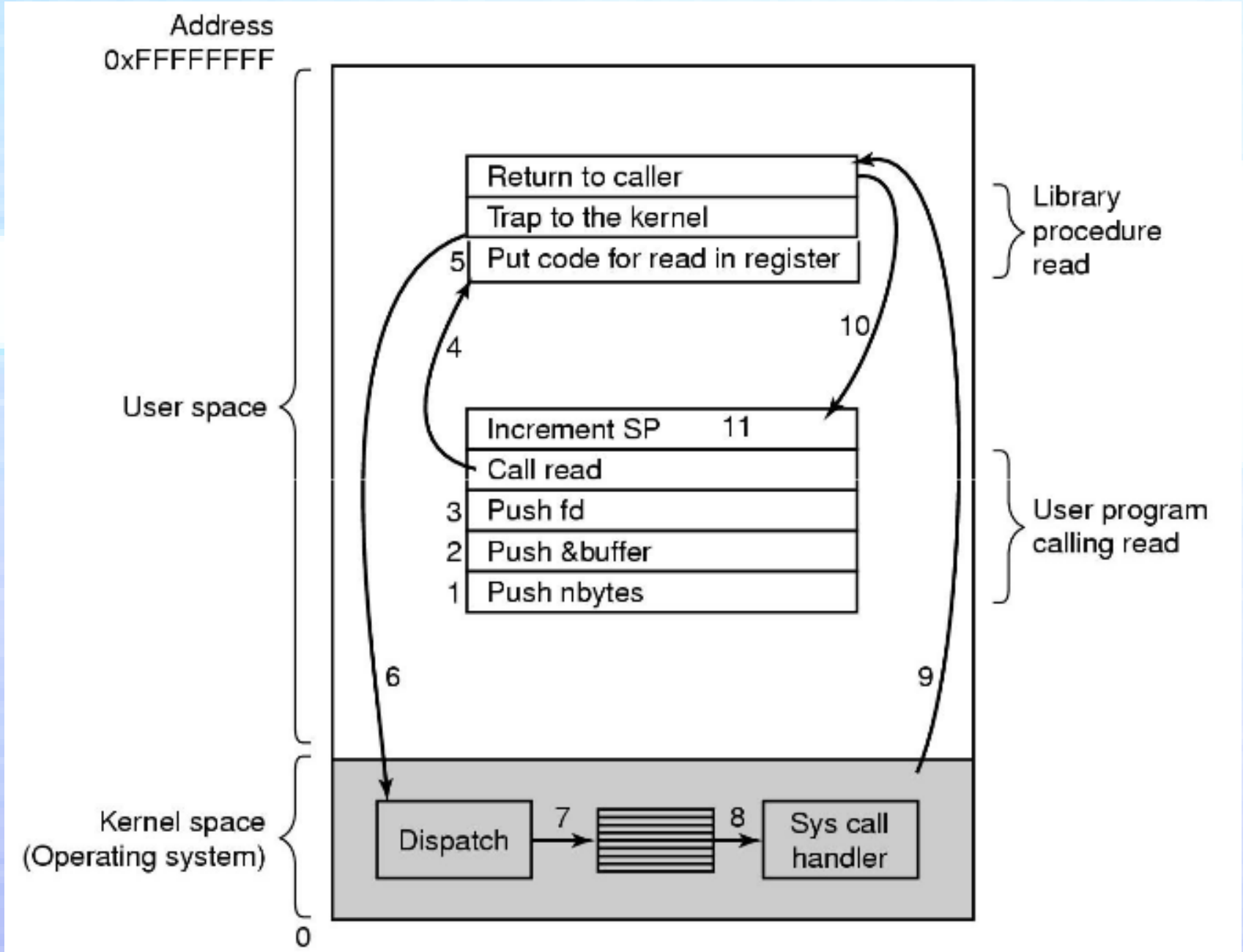
```
unsigned int ScGetProcessId(KernelState& state, InterruptFrame& frame, Tcb& tcb);
```

Обработчик системного вызова

Существует три способа передачи аргументов в системный вызов

- Через регистры процессора
- Через стек вызывающего приложения
- Через память
  - Через специальную фиксированную область памяти

Системный вызов



Исключения

Исключения

Исключение – это аномамальное состояние в которое вошел процессор во время выполнения потока инструкций. Такие состояния рапортуются процессором операционной системе через осуществление “специального” входа в ядро. Источники исключений:

- Ошибки обнаруженные процессором
- Специальные инструкции
- Machine-check exceptions

Исключения

Типы исключений:

- Отказы
- Ловушки
- Фатальные

Исключения

0	Divide Error
2	Non-Maskable Interrupt
3	Breakpoint Exception
6	Invalid Opcode
11	Segment Not Present
12	Stack-Segment Fault
13	General Protection Fault
14	Page Fault
18	Machine Check
32-255	User Defined Interrupts

Исключения

Vector No.	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug	Fault/ Trap	No	Any code or data reference or the INT 1 instruction.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT 3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (Zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[2]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory.[3]
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.[4]
19	#XF	SIMD Floating-Point Exception	Fault	No	SSE and SSE2 floating-point instructions[5]
20-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

Прервания

Прерывания

Прерывание – это сигнал генерируюемый аппаратными устройствами для процессора асинхронно (параллельно и непредсказуемо) для последнего и сообщающий о наступлении какого-либо события, которое требует внимания и обработки со стороны операционной системы. Типичной реакцией на прерывание

со стороны процессора и операционной системы является приостановка выполнения текущего потока инструкций, вытеснение текущей задачи и вызов специальной процедуры обработки прерывания.

Прерывания

Прерывания бывают маскируемыми и немаскируемыми. Большинство прерываний являются маскируемым. Получение маскируемых прерываний можно запрещать и/или разрешать с помощью специальных инструкций **cli**/**sti**.

cli/sti

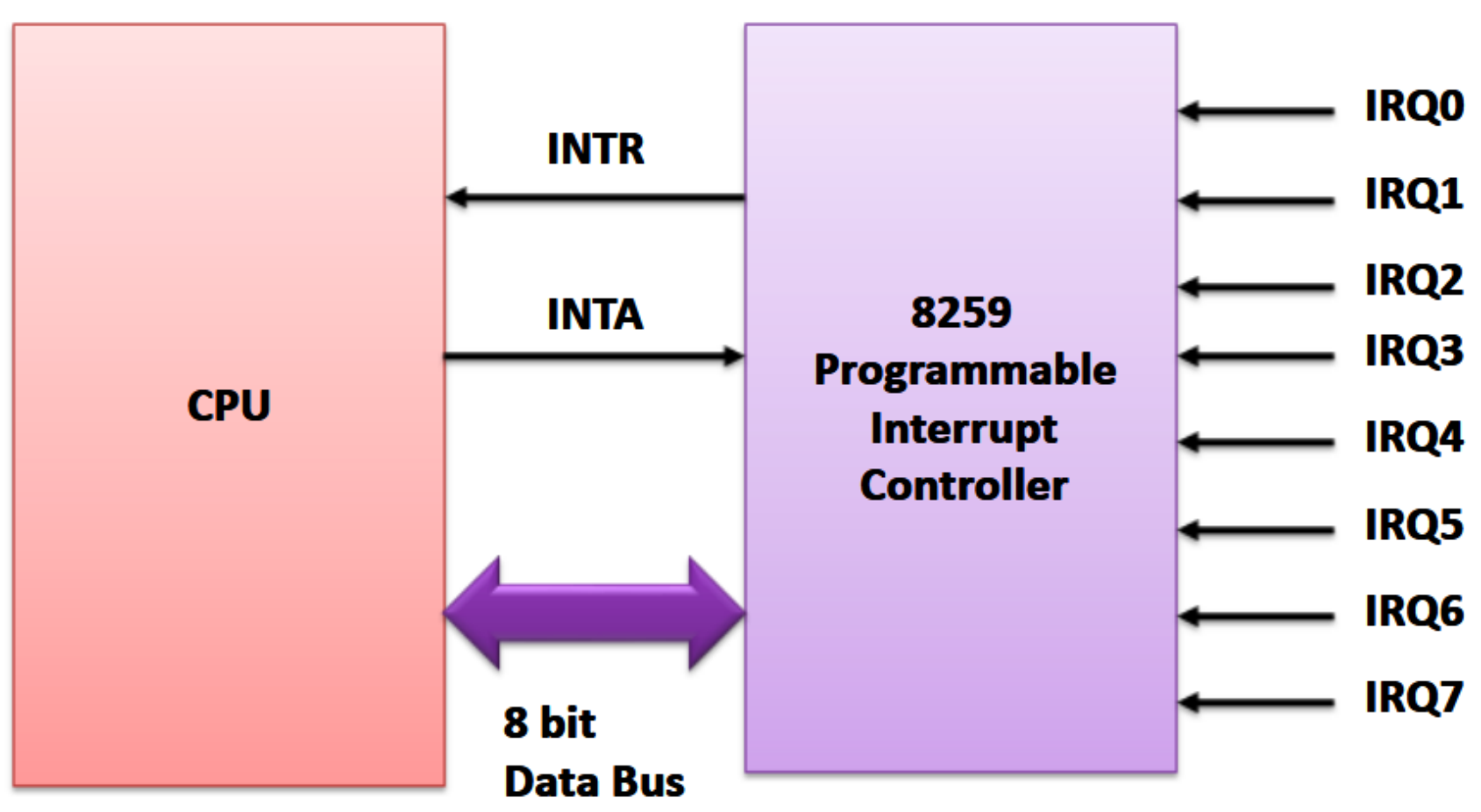
Прерывания являются асинхронными и непредсказуемы с точки зрения кода выполняющегося на процессоре. Обработчики прерываний обмениваются данными с ядром ОС и драйверами. Обработчик прерываний обладает наивысшим приоритетом выполнения в системе. Требуется синхронизация. Единственный доступный метод синхронизации в таком контексте это гарантировать отсутствие появления прерываний на время критической секции.

Приоритеты

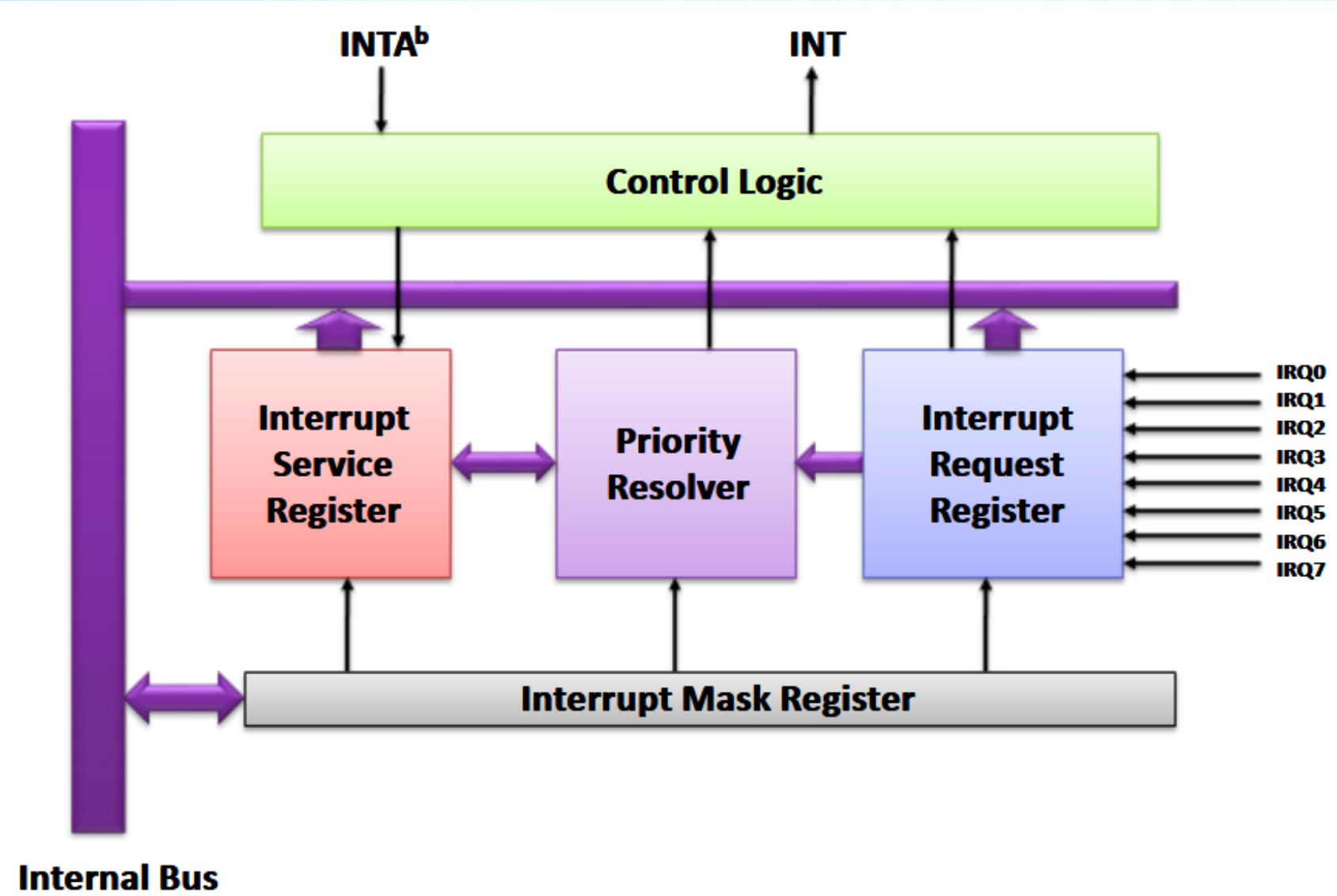
В большинстве развитых систем присутствует система приоритетов прерываний. То есть одни прерывания могут вытеснять обработку других прерываний. Проблема – Stack Overflow.

*Hint: автоматическое запрещение прерываний в Interrupt Gate. End-Of-Interrupt (EOI) – специальная команда

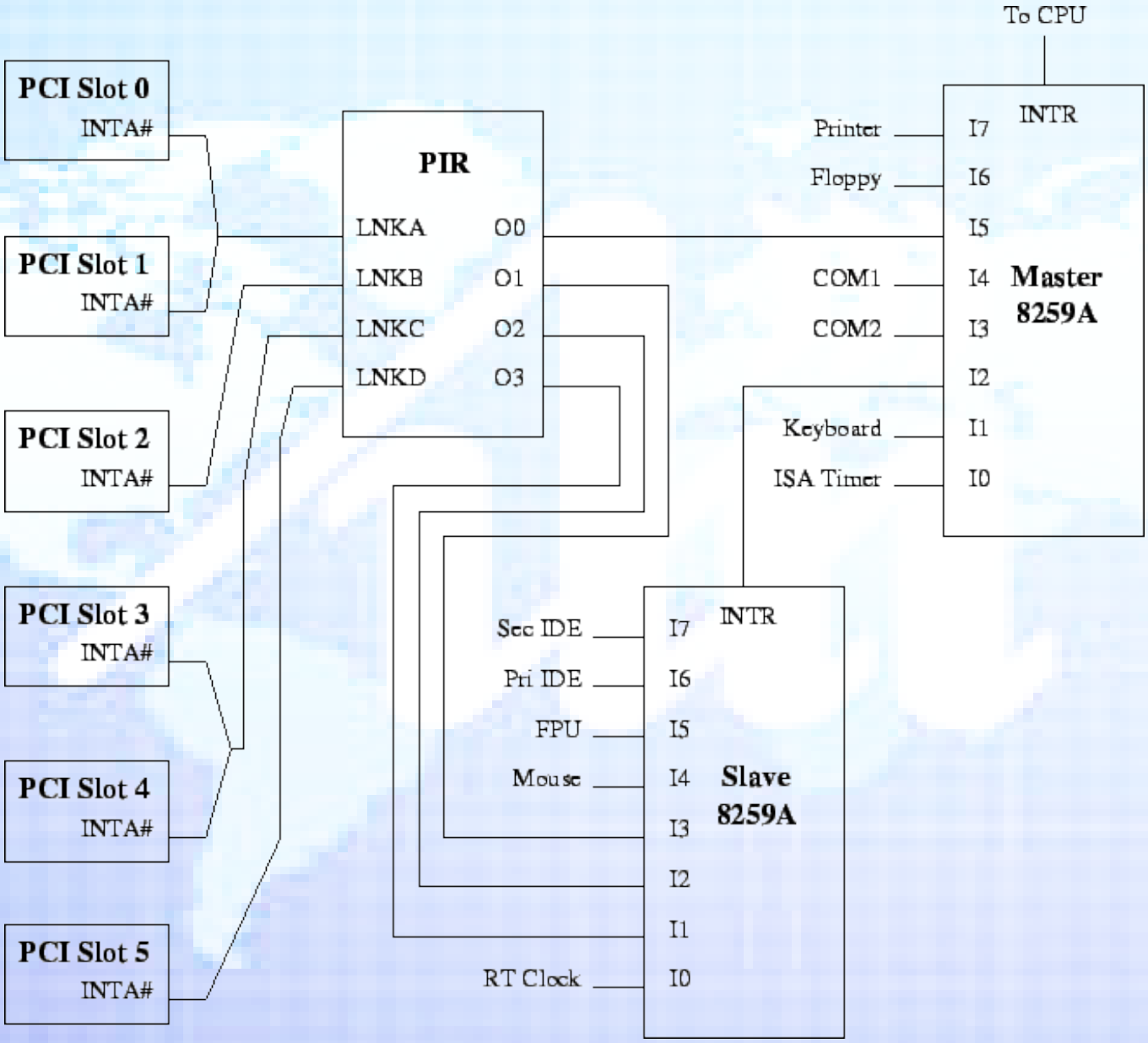
Контроллер прерываний



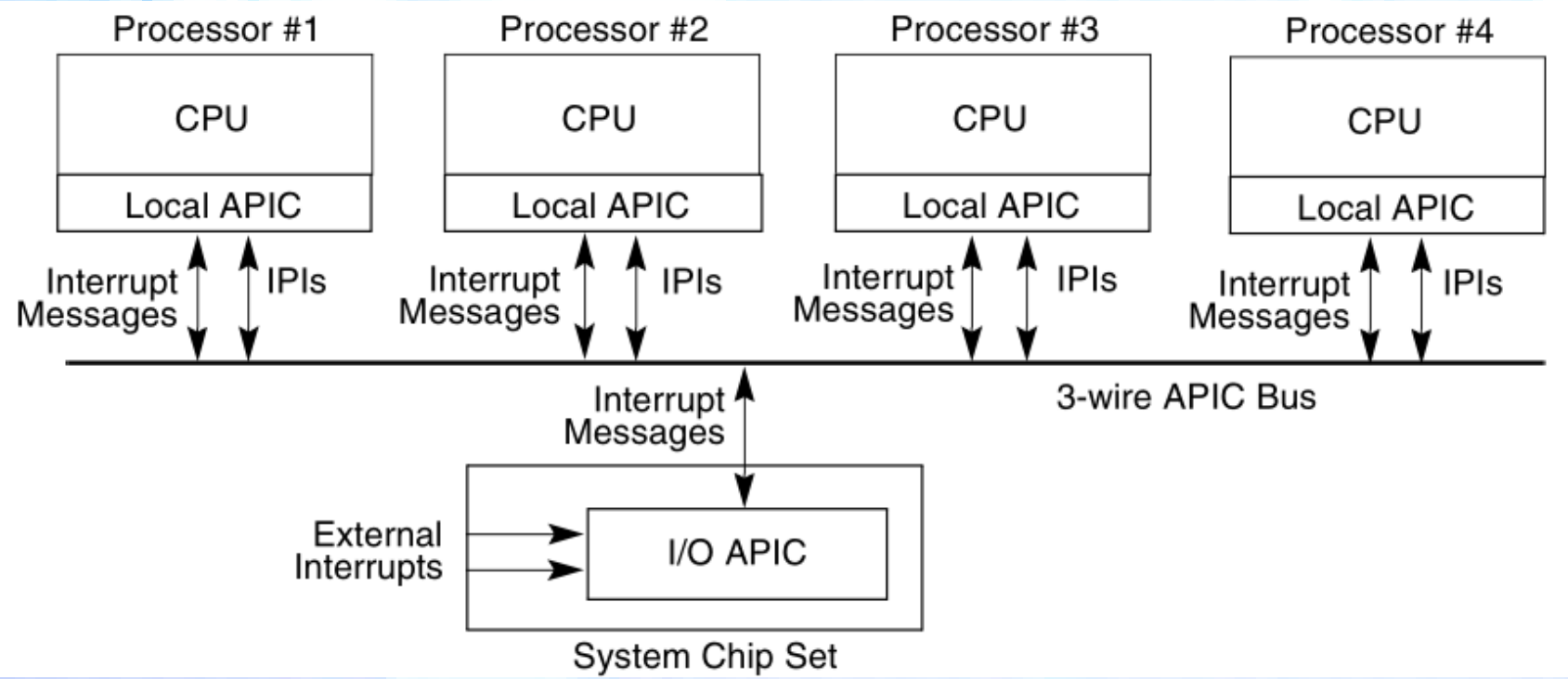
Контроллер прерываний



Контроллер прерываний



Контроллер прерываний



Local APIC

LAPIC является интегральной частью процессора в отличие от PIC, который является внешним устройством.

Local APIC

Фичи:

- Глобальное включение/отключение
- Идентификатор процессора
- Две локальные IRQ линии прерываний
- Локальный высокочастотный таймер!!!!
  - Одноразовый взвод
  - Периодическая генерация прерываний
- Межпроцессные прерывания
  - На фиксированный процессор
  - На процессор с наименьшим приоритетом
  - SMI
  - NMI
  - INIT
  - Start Up
- Приорететы прерываний
- Мониторинг
- Ложные прерывания

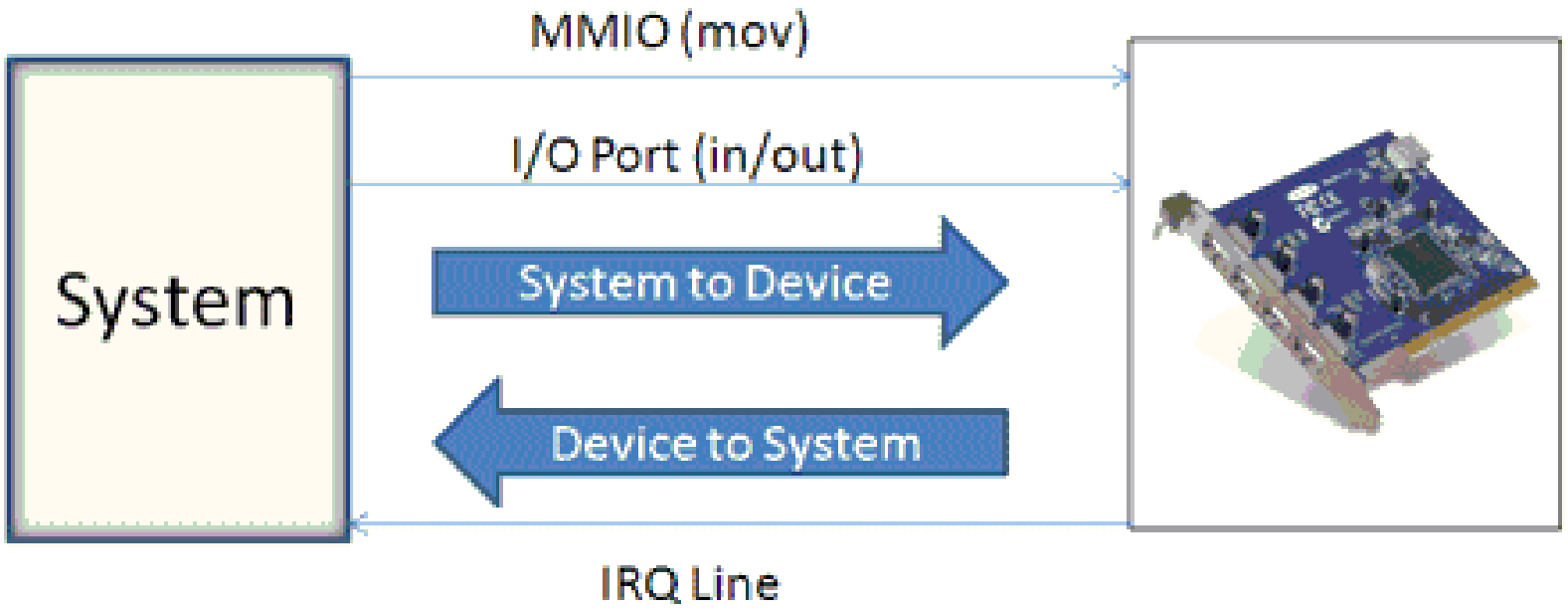
Ввод-вывод

Вввод-вывод:

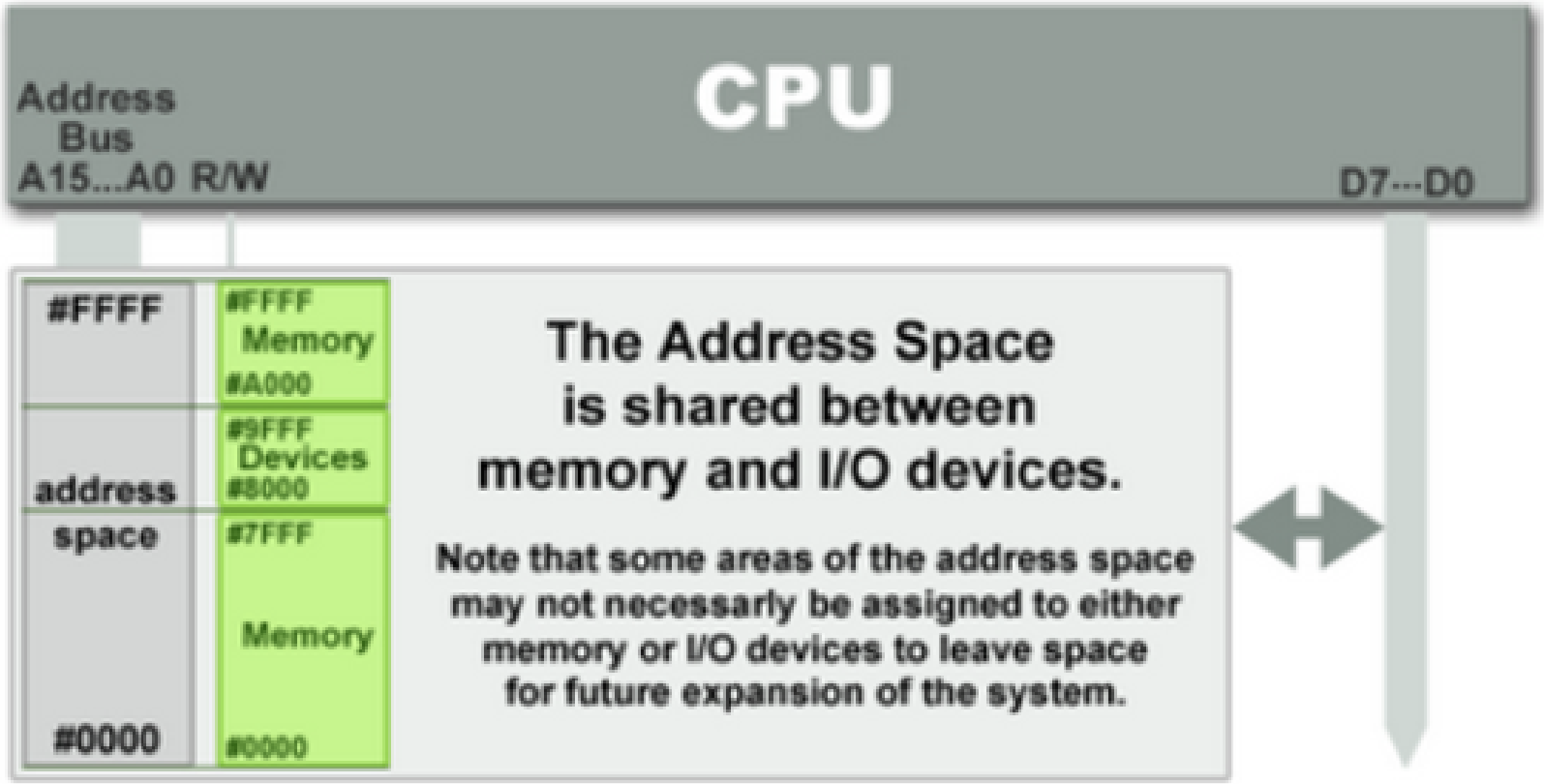
- Прерывания
- Port-Mapped IO (PMIO)
- Memory-Mapped IO (MMIO)
- Direct Memory Access

Прерывания инициируются аппаратным обеспечением и представляют собой очень специфическое средство коммуникации, которое фактически реализует только передачу управления. MMIO и PMIO инициируются только процессором (так же как

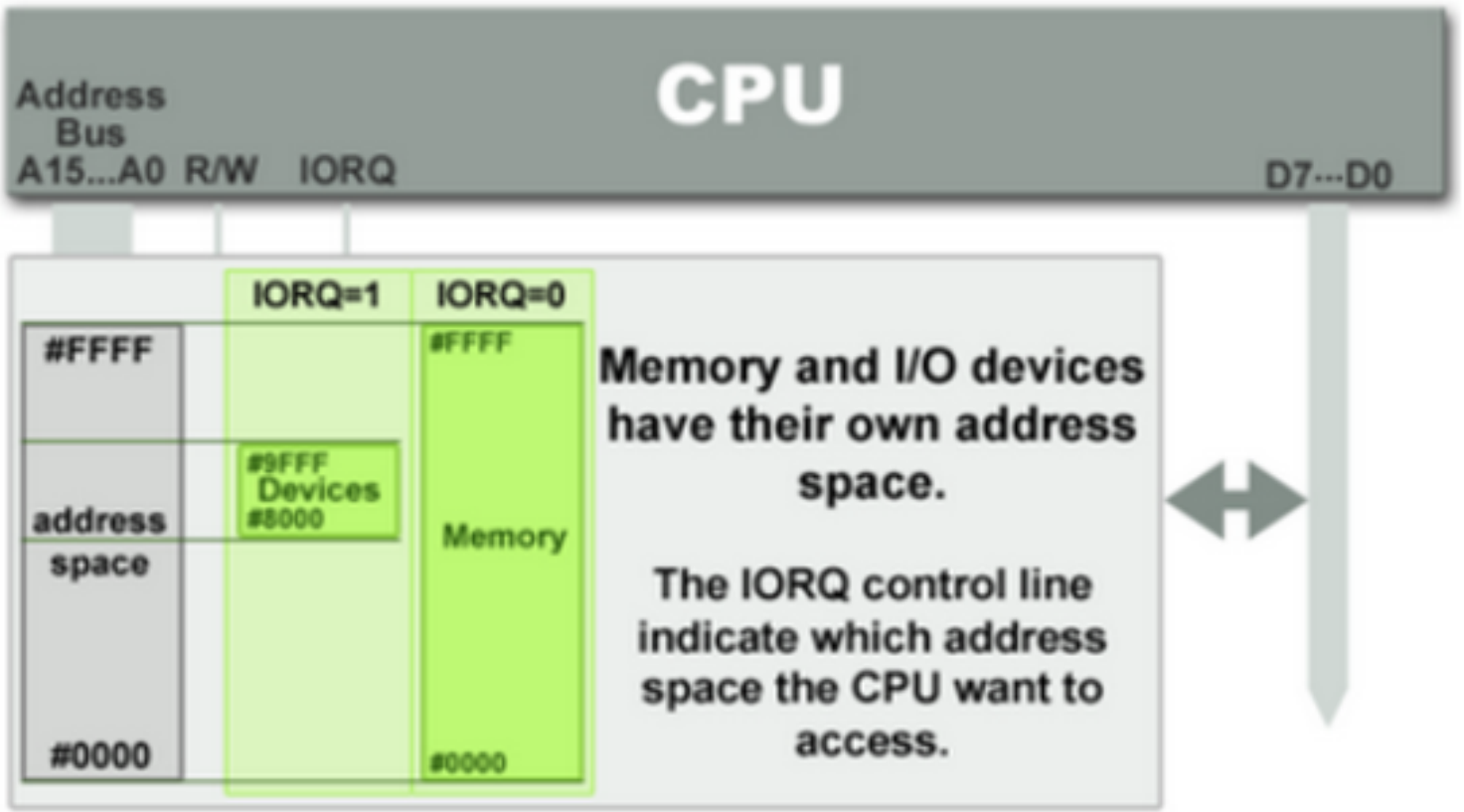
и DMA-операции) и реализуют обмен данными между памятью и аппаратными устройствами.


MMIO (mov)
I/O Port (in/out)
System
System to Device
Device to System
IRQ Line

Memory-mapped IO (MMIO)



Port-mapped IO (PMIO or Isolated IO)

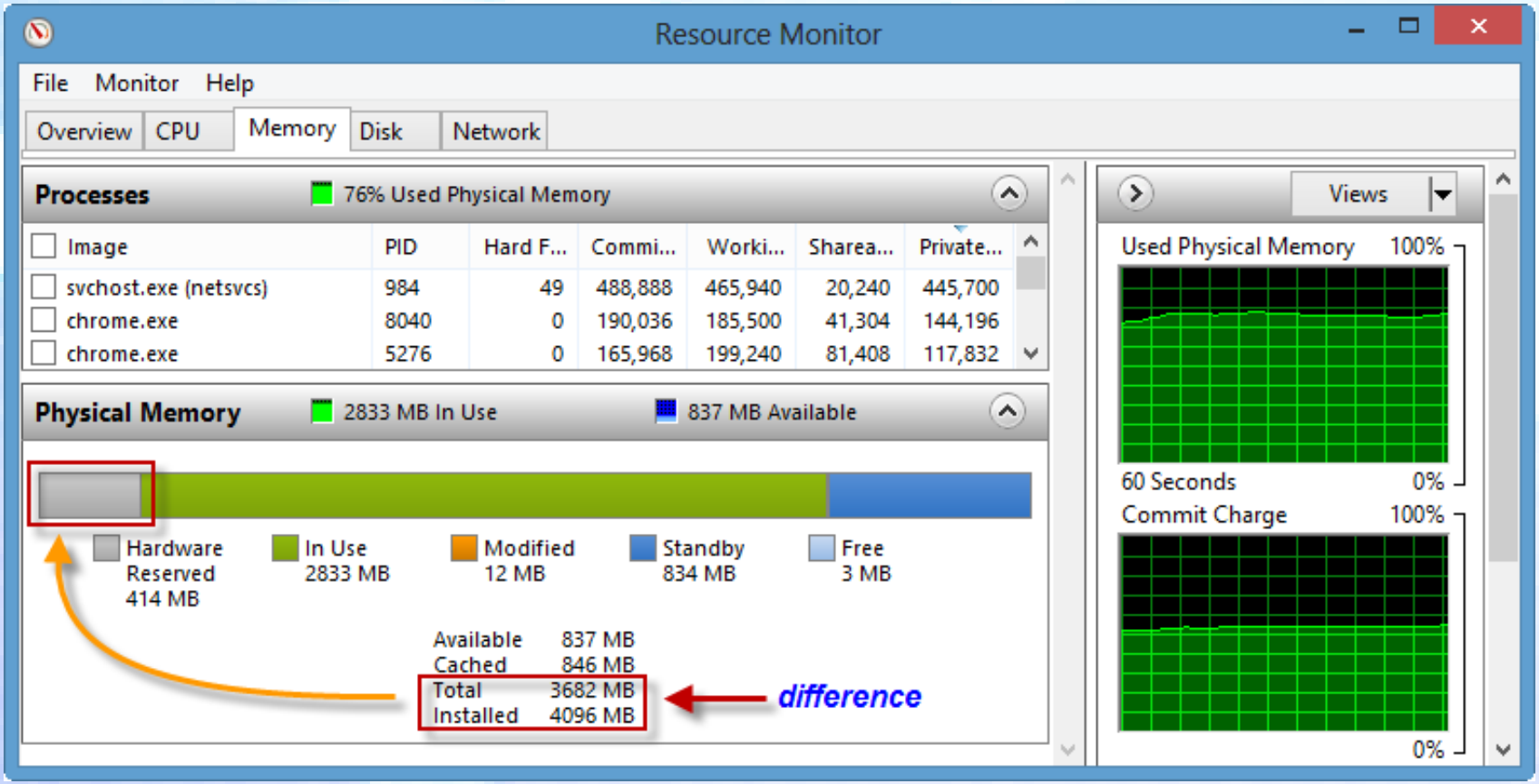


MMIO: Пример

Table 8-1. Local APIC Register Address Map

Address	Register Name	Software Read/Write
FEE0 0000H	Reserved	
FEE0 0010H	Reserved	
FEE0 0020H	Local APIC ID Register	Read/Write.
FEE0 0030H	Local APIC Version Register	Read Only.
FEE0 0040H	Reserved	
FEE0 0050H	Reserved	
FEE0 0060H	Reserved	
FEE0 0070H	Reserved	
FEE0 0080H	Task Priority Register (TPR)	Read/Write.
FEE0 0090H	Arbitration Priority Register[1] (APR)	Read Only.
FEE0 00A0H	Processor Priority Register (PPR)	Read Only.
FEE0 00B0H	EOI Register	Write Only.
FEE0 00C0H	Reserved	

MMIO: Пример


Resource Monitor
File Monitor Help
Overview CPU Memory Disk Network
Processes
76% Used Physical Memory
Image PID Hard F... Commi... Worki... Sharea... Private...
svchost.exe (netsvcs) 984 49 488,888 465,940 20,240 445,700
chrome.exe 8040 0 190,036 185,500 41,304 144,196
chrome.exe 5276 0 165,968 199,240 81,408 117,832
Physical Memory
2833 MB In Use
837 MB Available
Hardware Reserved 414 MB
In Use 2833 MB
Modified 12 MB
Standby 834 MB
Free 3 MB
Available 837 MB
Cached 846 MB
Total 3682 MB
Installed 4096 MB
difference
Views
Used Physical Memory 100%
60 Seconds 0%
Commit Charge 100%
0%

MMIO: Пример

```
int GetCpuId()
{
    return *reinterpret_cast<uint32_t*>(0xFEE00020) » 24;
}

int EndOfInterrupt()
{
    *reinterpret_cast<uint32_t*>(0xFEE000B0) = 0;
}
```

MMIO

Доступ к MMIO контролируется и управляется MMU.

MMU дает возможность мелкогранулярного управления доступом к вводу-выводу и его проброс в процессы пользовательского режима.

MMIO удобен и поддерживается практически всеми процессорами.

PMIO

Редко используется но применяется в x86.

Работает с отдельным 16-битным адресным пространством.

Использует специальные инструкции:

- in *port*, *value*
- out *port*, *value*

Системный флаг IOPL контролирует глобальный доступ кода к PMIO.

Битовая карта разрешений доступа в TSS предоставлет средство мелко-зернистого контроля за доступом к портам.

PMIO

Классическая реализация управления доступом к PMIO подразумевает, что весь код осуществляющий такой доступ выполняется с привелегиями ядра.

Вспоминаем монолитные ядра...

Вспоминаем что мы доверяем драйверам и надеемся что они не будут портить жизнь ни друг другу ни ядру...

Концепция доверенных потоков

Данная концепция реализует контроль доступа к PMIO через IOPL флаг.

В контексте доверенных потоков этот флаг выставляется в состояние разрешающее ввод-вывод, в контексте недоверенных – запрещающий.

Мелко-гранулярный контроль

Данная концепция реализует контроль доступа к PMIO через Permisson Bitmap в TSS.

Позволяет осуществлять прямой доступ PMIO из драйвера.

Реализует низко-гранулярный контроль над портами через которые драйвер может осуществлять ввод-вывод.

Усложняет ядро.

Накладные расходы в терминах адресного пространства ядра.

Накладные расходы на переключение контекстов.

Прямой контроль со стороны ядра

Всеоперации PMIO реализуются через специальные системные вызовы.

Ядро применяет политику прав доступа.

Дорого в терминах процессорного времени.

Прямой контроль*

Всеоперации PMIO реализуются через специальные системные вызовы.

Ядро применяет политику прав доступа.

Однако...

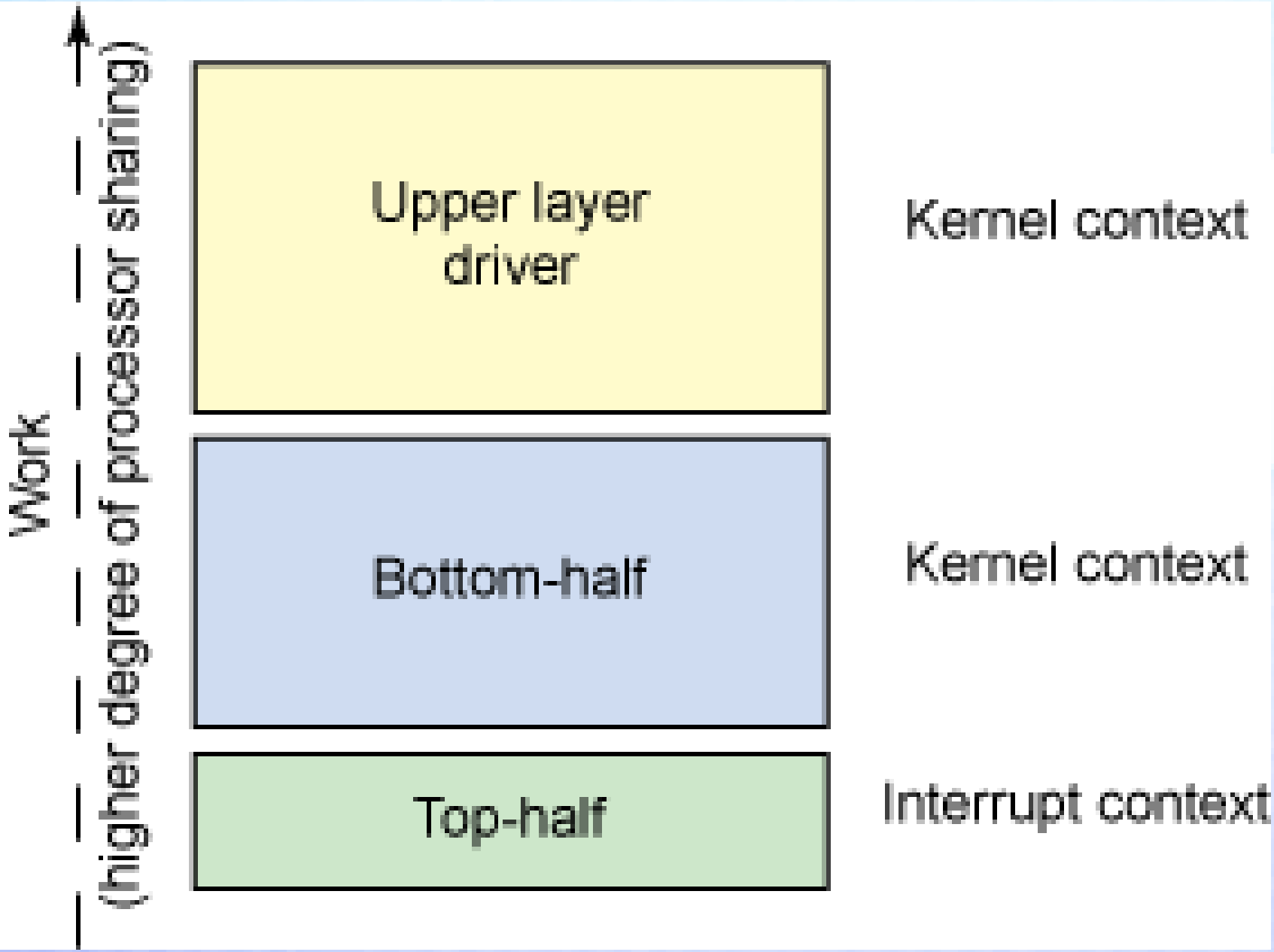
Специальное соглашение о системных вызовах позволяет упаковывать несколько операций ввода-авода в один системный вызов.

Меньше накладные расходы но нужна специальная поддержка для программистов драйверов.

Прерывания

Прерывание – событие генерируемое внешним устройством, которое требует немедленной обработки и которое приводит к вытеснению задачи выполняющейся на процессоре кодом предназначенны для его обработки.
Прерывания являются асинхронными событиями и поэтому требуют специальных подходов для их обслуживания.

Прерывания: Linux

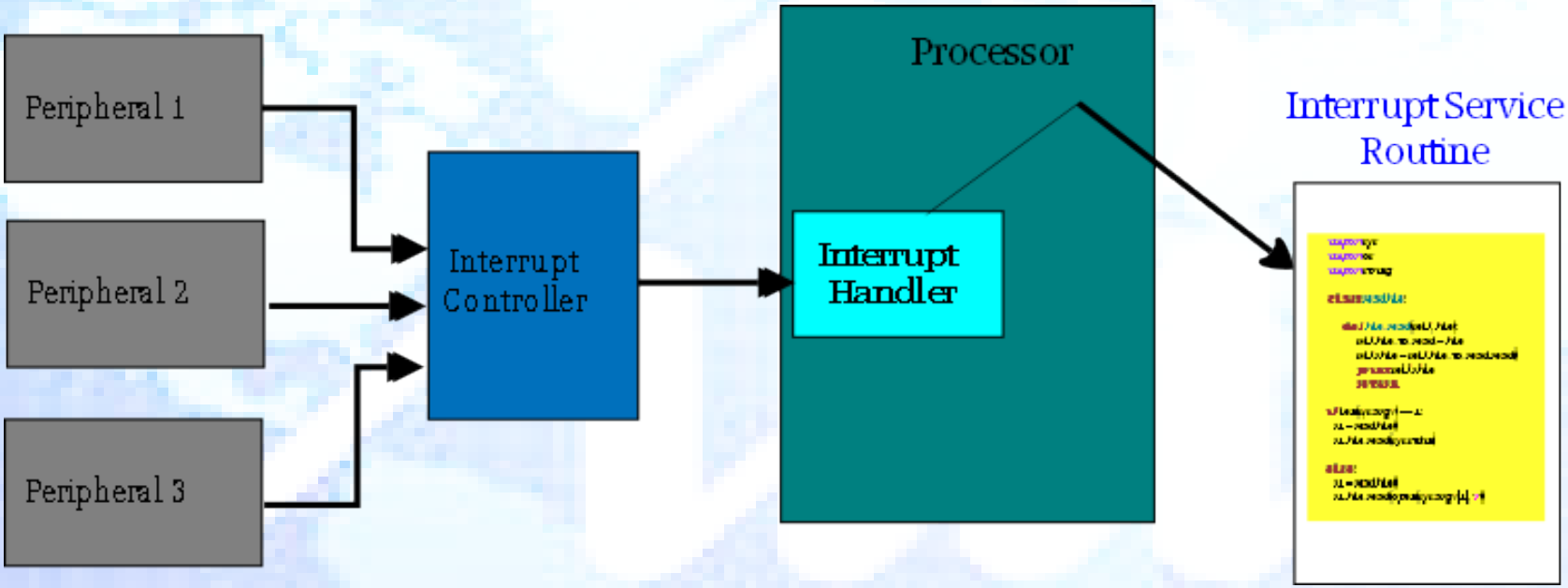


Прерывания: Linux

Interrupt Service Routine (ISR), также известна как IRQ Handler – связанная с прерыванием (а следовательно с устройством генерирующим прерывания) процедура их обработки.

ISR является ключевой компонентой драйверов аппаратных устройств.

Прерывания: Linux



Для систем на базе монолитного ядра, классическим подходом в обработке перерываний является разделение ISR на две части – так называемые Top Halfs и Bottom Halfs.

Top Half Vs Bottom Half

Top Half	Bottom Half
Processing of tasks in Interrupt Handler (Interrupt context)	Processing of tasks in Kernel context
Interrupts are disabled	Interrupts are not disabled
Add Deferrable functions for delayed execution	Handles Deferrable functions
Processing time should be less	-NA-
Uses Tasklets and work queue APIs for deferrable mechanism	

Bottom Half: Linux

- Softirq
- Tasklets
- Workqueue
- Threaded Interrupts

Softirq

- Softirqs предназначены для долгоиграющей неблокирующей обработки прерываний.
- Выполняются сразу же после Top Half ISR (и до всех остальных потоков управляемых планировщиком)
- Softirqs являются прерываемыми, но обладают практически наивысшим приоритетом.
- Softirqs задаются статически. (определены в коде ядра и не могут изменяться динамически. (32 Softirqs, 1 обработчик для каждого)
- Softirqs выполняются на процессоре, на котором они были вызваны (никакой миграции между процессорами).
- Подобно сигналам, Softirqs управляются битовой маской. Соответственно, Softirqs аккумулируются.

Softirq

То есть softirqs:

- Задаются во время компиляции, а не во время выполнения.
- Имеют практически наивысший приоритет.
- Могут выполняться параллельно.
- Имеют фиксированную очередность выполнения (приоритет между собой).

```
void open_softirq(int nr, void (*action)(struct softirq_action *));
void raise_softirq(unsigned int nr);
```

Softirq

ksoftirqd – поток ядра, привязанный к процессору, на который перекладывается выполнение Softirq в случае перегрузки.

Tasklets

- Tasklets предназначены для долгоиграющей неблокирующей обработки прерываний.
- Tasklets в отличие от Softirqs являются динамическими.
- Tasklets реализованы на базе Softirqs.
- Выполняются сразу же после Top Half ISR (и до всех остальных потоков управляемых планировщиком)
- Tasklets не могут выполняться ни конкурентно ни параллельно.
- Tasklets управляются динамическими списками.

Workqueues

- Workqueues представляют собой транспорт между ISR и рабочим потоком ядра (драйвером).
- ISR добавляет данные о событиях в очередь (производитель/писатель).
- Рабочий поток изымает данные о событиях из очереди (потребитель/читатель) и обрабатывает их.
- В ядре поддерживается пул рабочих потоков (по одному потоку на ядро процессора).

Threaded Interrupts

- Обработчик выполняется в своем собственном потоке ядра.
- Threaded Interrupts являются планируемыми.
- Не могут привести систему к голоданию или зависанию.
- Могут блокироваться и засыпать.
- Однако не могут блокировать или откладывать надолго выполнения других отлоденных прерываний.
- Существует гибкость в задании приоритета выполнения.

Обработка прерываний в микроядрах

Микроядра используют только Threaded Interrupts.
Прерывание передается драйверу в виде сообщения.
Ядро выступает в качестве одного из коммуницирующих сервисов системы.
Однако, ядро только отправлаяет сообщения, но не принимает их.
Драйверы регистрируют свой интерес в получении прерываний в ядре.

Пару слов о виртуализации

Виртуализация бывает:

- Приложений
- Систем

Примеры виртуализации уровня приложений: JVM, .Net Framework, Lua.

Примеры виртуализации уровня систем: VmWare Workstation, VmWare ESX Server.