Роль процессора в работе ОС на примере xv6-riscv

Тюрин Иван Николаевич, P33102

Рассмотрим

1. Запуск ОС и режимы работы процессора
2. Виртуальная память
3. Система прерываний
4. Многопоточность
5. Планировщик
6. * Файловая система

Дисклеймер

- Я такой же студент

Смотри

- se.ifmo.ru/os
- gitlab.se.ifmo.ru/computer-systems/csa-rolling
- xv6-book & исходники
- Спецификация RISC-V

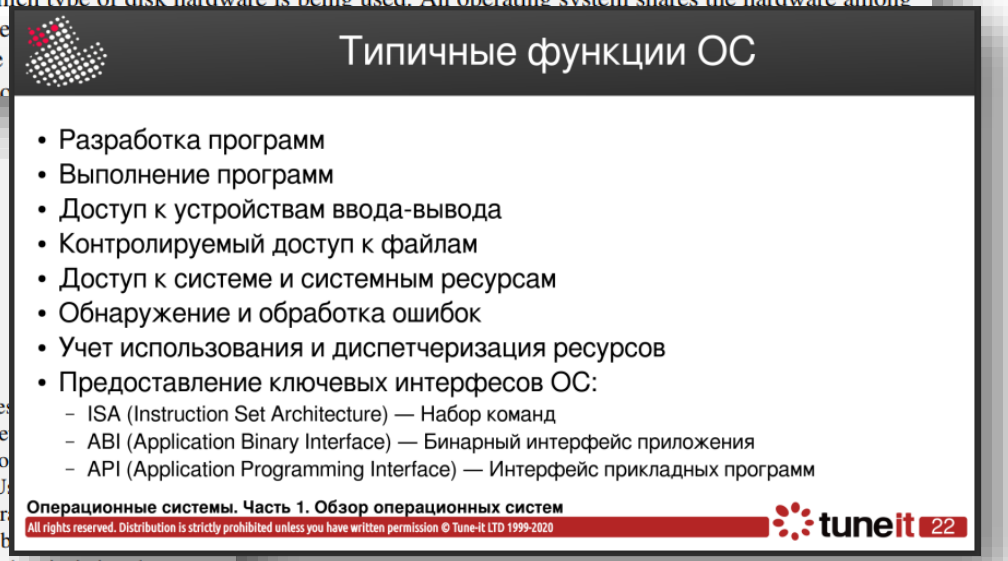
Операционная система

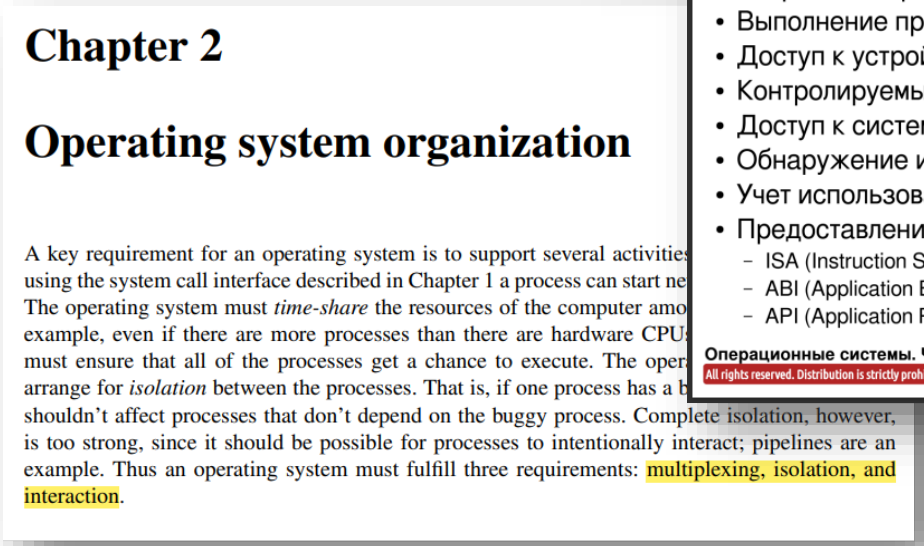
- Разделение ресурсов компьютера между программами
- Разработка программ
- Уровень абстракции и изоляции

Chapter 1

Operating system interfaces

The job of an operating system is to share a computer among multiple programs and to provide a more useful set of services than the hardware alone supports. An operating system manages and abstracts the low-level hardware, so that, for example, a word processor need not concern itself with which type of disk hardware is being used. An operating system shares the hardware among multiple
provide





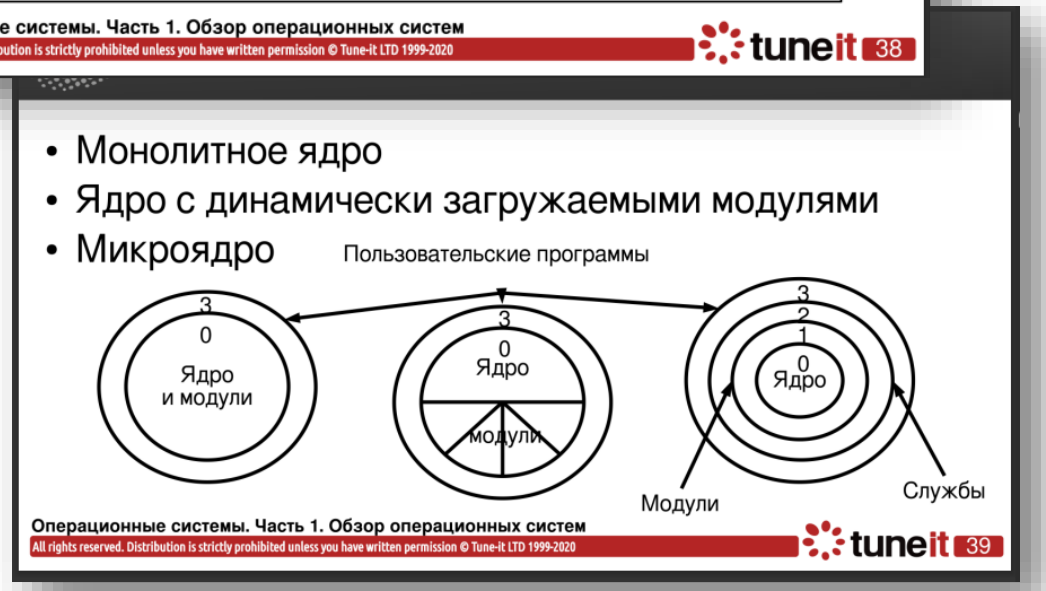
Chapter 2

Operating system organization

A key requirement for an operating system is to support several activities
using the system call interface described in Chapter 1 a process can start ne
The operating system must *time-share* the resources of the computer amo
example, even if there are more processes than there are hardware CPU
must ensure that all of the processes get a chance to execute. The oper
arrange for *isolation* between the processes. That is, if one process has a b
shouldn't affect processes that don't depend on the buggy process. Complete isolation, however, is too strong, since it should be possible for processes to intentionally interact; pipelines are an example. Thus an operating system must fulfill three requirements: multiplexing, isolation, and interaction.

Организация ОС

- Ядро
- Пользовательские процессы
- Системные вызовы
- Драйвера и устройства



Xv6-riscv

- Учебная ОС из MIT
- C & RISC-V ASM
- Sv39 RISC-V
- Unix-like
- ~~POSIX~~

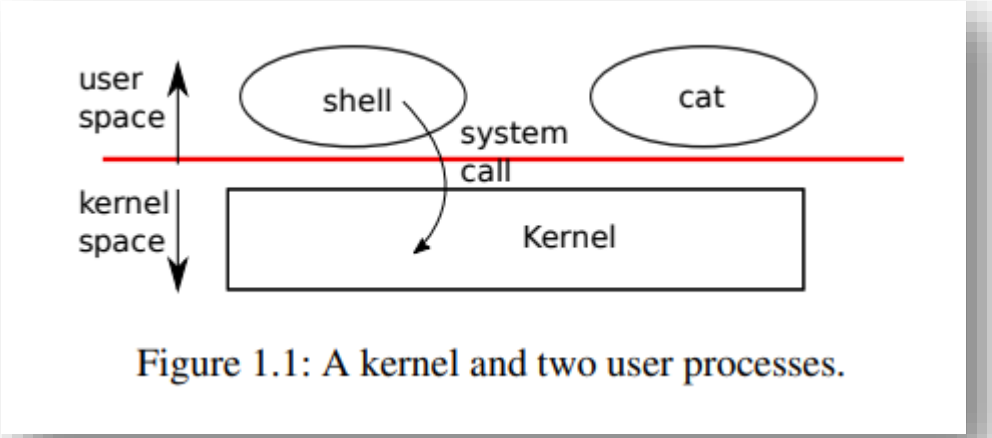


Figure 1.1: A kernel and two user processes.

- github.com/mit-pdos/xv6-riscv
- pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf

RISC-V процессор

- riscv.org
- RISC
- Открытая
- Современная
- riscv-alliance.ru

Разработка xv6-riscv

- Make
- qemu-system-riscv64
- github.com/riscv-collab/riscv-gnu-toolchain
- 128Mb RAM
- Консоль
- Образ диска

Makefile

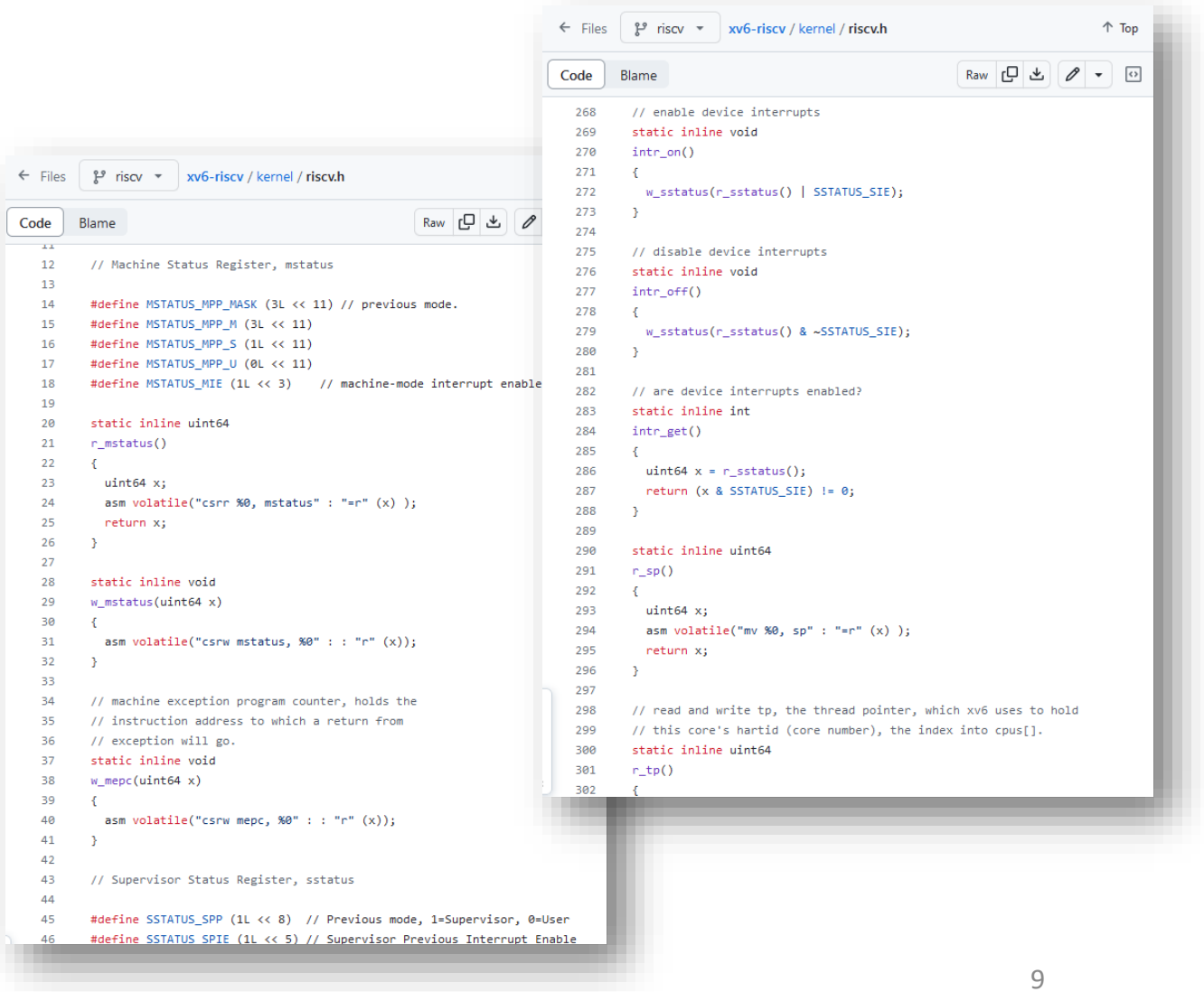
```
159    QEMUOPTS = -machine virt -bios none -kernel $K/kernel -m 128M -smp $(CPUS) -nographic
160    QEMUOPTS += -global virtio-mmio.force-legacy=false
161    QEMUOPTS += -drive file=fs.img,if=none,format=raw,id=x0
162    QEMUOPTS += -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0
```

- hub.docker.com/r/wtakuo/xv6-env

Программирование процессора

- Сишные функции

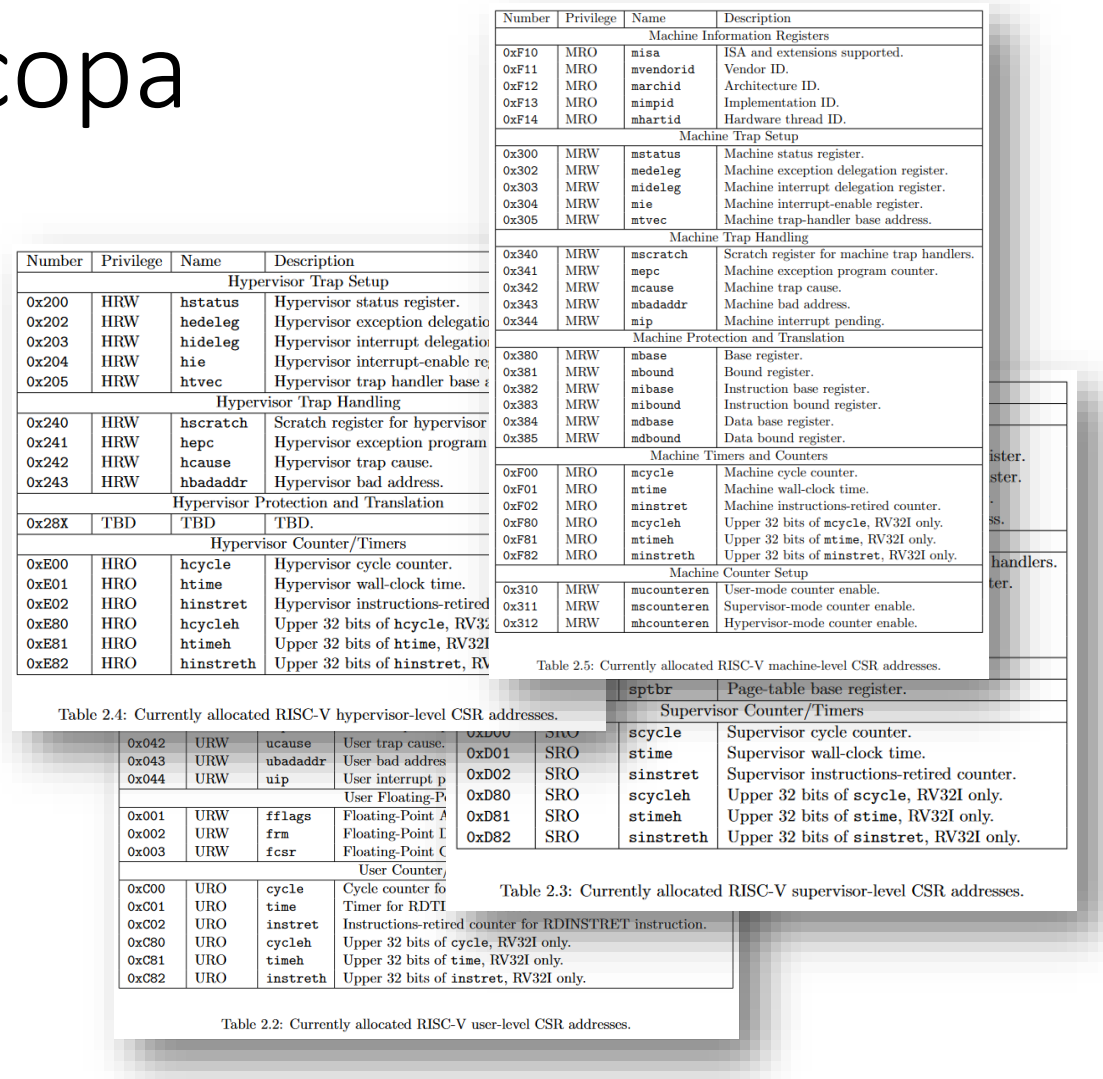
+ Ассемблерные вставки



Запуск операционной системы

Режимы работы процессора

- Modes:
  - Machine
  - Hypervisor,
  - Supervisor,
  - User;
- Control and Status Registers (CSRs)



Number	Privilege	Name	Description
		Machine Information Registers	
0xF10	MRO	misa	ISA and extensions supported.
0xF11	MRO	mvendorid	Vendor ID.
0xF12	MRO	marchid	Architecture ID.
0xF13	MRO	mimpid	Implementation ID.
0xF14	MRO	mhartid	Hardware thread ID.
		Machine Trap Setup	
0x300	MRW	mstatus	Machine status register.
0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x305	MRW	mtvec	Machine trap-handler base address.
		Machine Trap Handling	
0x340	MRW	mscratch	Scratch register for machine trap handlers.
0x341	MRW	mepc	Machine exception program counter.
0x342	MRW	mcause	Machine trap cause.
0x343	MRW	mbadaddr	Machine bad address.
0x344	MRW	mip	Machine interrupt pending.
		Machine Protection and Translation	
0x380	MRW	mbase	Base register.
0x381	MRW	mbound	Bound register.
0x382	MRW	mibase	Instruction base register.
0x383	MRW	mibound	Instruction bound register.
0x384	MRW	mdbase	Data base register.
0x385	MRW	mdbound	Data bound register.
		Machine Timers and Counters	
0xF00	MRO	mcycle	Machine cycle counter.
0xF01	MRO	mtime	Machine wall-clock time.
0xF02	MRO	minstret	Machine instructions-retired counter.
0xF80	MRO	mcycleh	Upper 32 bits of mcycle, RV32I only.
0xF81	MRO	mtimeh	Upper 32 bits of mtime, RV32I only.
0xF82	MRO	minstreth	Upper 32 bits of minstret, RV32I only.
		Machine Counter Setup	
0x310	MRW	mucounteren	User-mode counter enable.
0x311	MRW	mscounteren	Supervisor-mode counter enable.
0x312	MRW	mhcounteren	Hypervisor-mode counter enable.

Table 2.5: Currently allocated RISC-V machine-level CSR addresses.

Number	Privilege	Name	Description
		Hypervisor Trap Setup	
0x200	HRW	hstatus	Hypervisor status register.
0x202	HRW	hedeleg	Hypervisor exception delegatio
0x203	HRW	hideleg	Hypervisor interrupt delegatio
0x204	HRW	hie	Hypervisor interrupt-enable re
0x205	HRW	htvec	Hypervisor trap handler base a
		Hypervisor Trap Handling	
0x240	HRW	hscratch	Scratch register for hypervisor
0x241	HRW	hepc	Hypervisor exception program
0x242	HRW	hcause	Hypervisor trap cause.
0x243	HRW	hbadaddr	Hypervisor bad address.
		Hypervisor Protection and Translation	
0x28X	TBD	TBD	TBD.
		Hypervisor Counter/Timers	
0xE00	HRO	hcycle	Hypervisor cycle counter.
0xE01	HRO	htime	Hypervisor wall-clock time.
0xE02	HRO	hinstret	Hypervisor instructions-retired
0xE80	HRO	hcycleh	Upper 32 bits of hcycle, RV3
0xE81	HRO	htimeh	Upper 32 bits of htime, RV32I
0xE82	HRO	hinstreth	Upper 32 bits of hinstret, RV

Table 2.4: Currently allocated RISC-V hypervisor-level CSR addresses.

			sptbr — Page-table base register.
			Supervisor Counter/Timers
0x042	URW	ucause	User trap cause.
0x043	URW	ubadaddr	User bad addres
0x044	URW	uip	User interrupt p
			User Floating-P
0x001	URW	fflags	Floating-Point A
0x002	URW	frm	Floating-Point I
0x003	URW	fcsr	Floating-Point C
			User Counter/
0xC00	URO	cycle	Cycle counter fo
0xC01	URO	time	Timer for RDTI
0xC02	URO	instret	Instructions-retired counter for RDINSTRET instruction.
0xC80	URO	cycleh	Upper 32 bits of cycle, RV32I only.
0xC81	URO	timeh	Upper 32 bits of time, RV32I only.
0xC82	URO	instreth	Upper 32 bits of instret, RV32I only.

Table 2.2: Currently allocated RISC-V user-level CSR addresses.

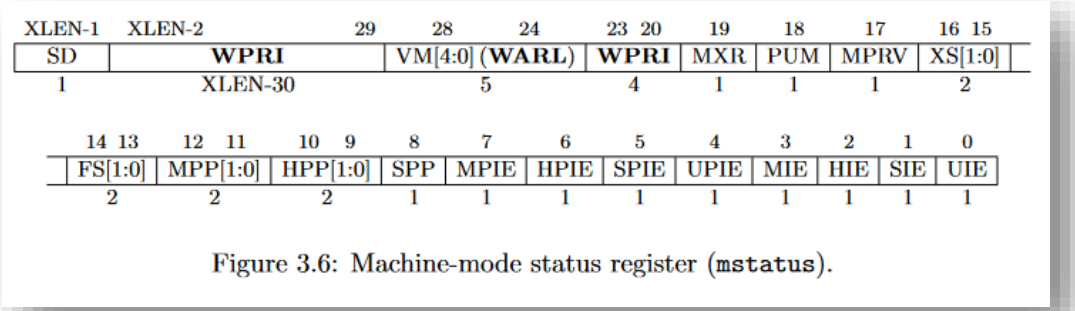
0xD00	SRO	scycle	Supervisor cycle counter.
0xD01	SRO	stime	Supervisor wall-clock time.
0xD02	SRO	sinstret	Supervisor instructions-retired counter.
0xD80	SRO	scycleh	Upper 32 bits of scycle, RV32I only.
0xD81	SRO	stimeh	Upper 32 bits of stime, RV32I only.
0xD82	SRO	sinstreth	Upper 32 bits of sinstret, RV32I only.

Table 2.3: Currently allocated RISC-V supervisor-level CSR addresses.

people.eecs.berkeley.edu/~krste/papers/riscv-privileged-v1.9.pdf

Control and Status Registers (CSRs)

- **`sp`** (stack pointer)
- **`mstatus`** (Machine status register)



XLEN-1	XLEN-2	29	28	24	23 20	19	18	17	16 15	
SD	**WPRI**		VM[4:0]	(**WARL**)	**WPRI**	MXR	PUM	MPRV	XS[1:0]	
1	XLEN-30		5		4	1	1	1	2	

14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
FS[1:0]	MPP[1:0]	HPP[1:0]	SPP	MPIE	HPIE	SPIE	UPIE	MIE	HIE	SIE	UIE
2	2	2	1	1	1	1	1	1	1	1	1

Figure 3.6: Machine-mode status register (`mstatus`).

- **`mepc`** (Machine exception program counter)
- **`satp`** (Supervisor Address Translation and Protection Register)
- **`mret`**, **`sret`** (ASM instructions)

The MRET, HRET, SRET, or URET instructions are used to return from traps in M-mode, H-mode, S-mode, or U-mode respectively. When executing an xRET instruction, supposing xPP holds the value y, yIE is set to xPIE; the privilege mode is changed to y; xPIE is set to 1; and xPP is set to U (or M if user-mode is not supported).

Процесс запуска

1. Включение (+питание)
2. Machine mode
   1. *Настройка процессора*
   2. Переход в Supervisor mode
3. Supervisor mode
   1. *Настройка ОС*
   2. *Инициализация устройств*
   3. Первый системный вызов
      и пользовательский процесс
   4. Переход в User mode
4. User mode
   1. shell в консоли

When the RISC-V computer powers on, it initializes itself and runs a boot loader which is stored in read-only memory. The boot loader loads the xv6 kernel into memory. Then, in machine mode, the CPU executes xv6 starting at `_entry` (kernel/entry.S:7). The RISC-V starts with paging hardware disabled: virtual addresses map directly to physical addresses.

The loader loads the xv6 kernel into memory at physical address `0x80000000`. The reason it places the kernel at `0x80000000` rather than `0x0` is because the address range `0x0:0x80000000` contains I/O devices.

The instructions at `_entry` set up a stack so that xv6 can run C code. Xv6 declares space for an initial stack, `stack0`, in the file `start.c` (kernel/start.c:11). The code at `_entry` loads the stack pointer register `sp` with the address `stack0+4096`, the top of the stack, because the stack on RISC-V grows down. Now that the kernel has a stack, `_entry` calls into C code at `start` (kernel/start.c:21).

The function `start` performs some configuration that is only allowed in machine mode, and then switches to supervisor mode. To enter supervisor mode, RISC-V provides the instruction `mret`. This instruction is most often used to return from a previous call from supervisor mode to machine mode. `start` isn't returning from such a call, and instead sets things up as if there had been one: it sets the previous privilege mode to supervisor in the register `mstatus`, it sets the return address to `main` by writing `main`'s address into the register `mepc`, disables virtual address translation in supervisor mode by writing `0` into the page-table register `satp`, and delegates all interrupts and exceptions to supervisor mode.

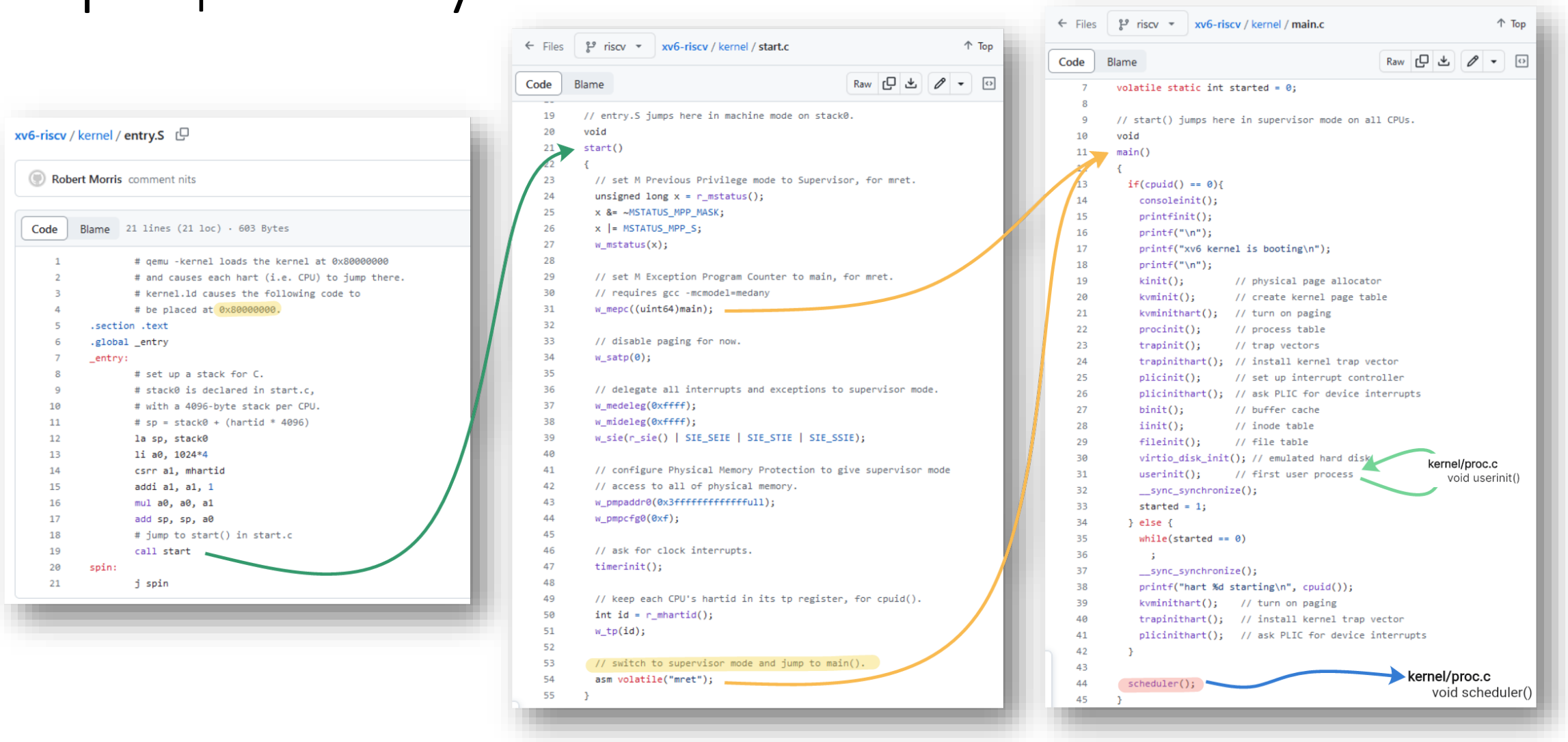
Before jumping into supervisor mode, `start` performs one more task: it programs the clock chip to generate timer interrupts. With this housekeeping out of the way, `start` "returns" to supervisor mode by calling `mret`. This causes the program counter to change to `main` (kernel/main.c:11).

After `main` (kernel/main.c:11) initializes several devices and subsystems, it creates the first process by calling `userinit` (kernel/proc.c:233). The first process executes a small program written in RISC-V assembly, which makes the first system call in xv6. `initcode.S` (user/initcode.S:3) loads the number for the `exec` system call, `SYS_EXEC` (kernel/syscall.h:8), into register `a7`, and then calls `ecall` to re-enter the kernel.

The kernel uses the number in register `a7` in `syscall` (kernel/syscall.c:132) to call the desired system call. The system call table (kernel/syscall.c:107) maps `SYS_EXEC` to `sys_exec`, which the kernel invokes. As we saw in Chapter 1, `exec` replaces the memory and registers of the current process with a new program (in this case, `/init`).

Once the kernel has completed `exec`, it returns to user space in the `/init` process. `init` (user/init.c:15) creates a new console device file if needed and then opens it as file descriptors 0, 1, and 2. Then it starts a shell on the console. The system is up.

Процесс запуска



Первый пользовательский процесс

- `userinit()` загружает небольшой код для вызова ассемблерной функции делающей первый системный вызов `exec`
- `exec` заменяет память процесса на память нужного исполняемого файла и начинает его исполнение

kernel/main.c
void main()

Files riscv xv6-riscv / kernel / proc.c Top

Code Blame Raw

```
217
218   // a user program that calls exec("/init")
219   // assembled from ../user/initcode.S
220   // od -t xC ../user/initcode
221   uchar initcode[] = {
222     0x17, 0x05, 0x00, 0x00, 0x13, 0x05, 0x45, 0x02,
223     0x97, 0x05, 0x00, 0x00, 0x93, 0x85, 0x35, 0x02,
224     0x93, 0x08, 0x70, 0x00, 0x73, 0x00, 0x00, 0x00,
225     0x93, 0x08, 0x20, 0x00, 0x73, 0x00, 0x00, 0x00,
226     0xef, 0xf0, 0x9f, 0xff, 0x2f, 0x69, 0x6e, 0x69,
227     0x74, 0x00, 0x00, 0x24, 0x00, 0x00, 0x00, 0x00,
228     0x00, 0x00, 0x00, 0x00
229   };
230
231   // Set up first user process.
232   void
233   userinit(void)
234   {
235     struct proc *p;
236
237     p = allocproc();
238     initproc = p;
239
240     // allocate one user page and copy initcode's instructions
241     // and data into it.
242     uvmfirst(p->pagetable, initcode, sizeof(initcode));
243     p->sz = PGSIZE;
244
245     // prepare for the very first "return" from kernel to user.
246     p->trapframe->epc = 0;      // user program counter
247     p->trapframe->sp = PGSIZE;  // user stack pointer
248
249     safestrcpy(p->name, "initcode", sizeof(p->name));
250     p->cwd = namei("/");
251
252     p->state = RUNNABLE;
253
254     release(&p->lock);
255   }
256
```

xv6-riscv / user / initcode.S

Robert Morris and kaashoek x 3 years ago

28 lines (23 loc) · 413 Bytes

Code Blame Raw

```
1   # Initial process that execs /init.
2   # This code runs in user space.
3
4   #include "syscall.h"
5
6   # exec(init, argv)
7   .globl start
8   start:
9           la a0, init
10          la a1, argv
11          li a7, SYS_exec
12          ecall
13
14  # for(;;) exit();
15  exit:
16          li a7, SYS_exit
17          ecall
18          jal exit
19
20  # char init[] = "/init\0";
21  init:
22    .string "/init\0"
23
24  # char *argv[] = { init, 0 };
25  .p2align 2
26  argv:
27    .long init
28    .long 0
```

Виртуальная память

Виртуальная память

все адреса теперь виртуальные

- Изоляция процессов
- Расширение доступной памяти
- Абстракция над доступом к ресурсам (mmap, DMA,…)

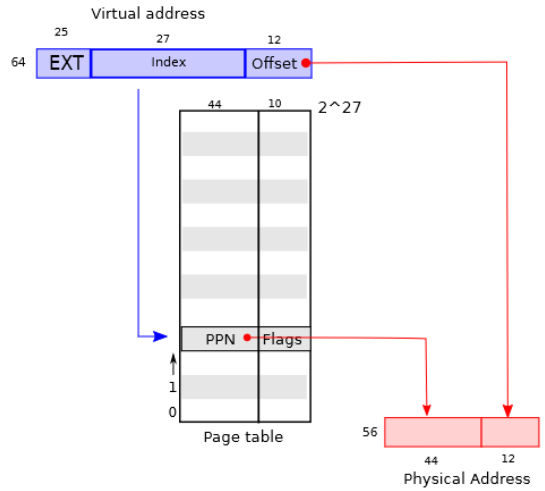


Figure 3.1: RISC-V virtual and physical addresses, with a simplified logical page table.

Виртуальная память: детали

Page Table

- 3 уровня (9 бит × 3) Page Directory’й
- Буквально страницы (4Кбайт) физической памяти
- Трансляция VA в офсеты (12 бит ↦ 4096 байт)
- Флаги доступа к PPN
- Page-fault Exception при ошибке доступа

RISC-V

- `satp` — регистр с “указателем” на корневую PD

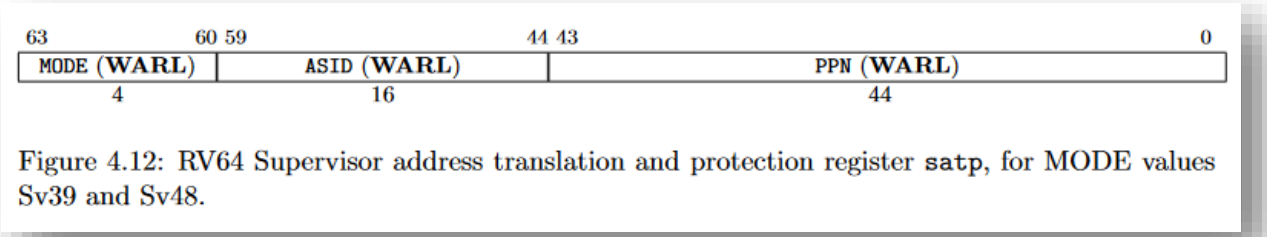


Figure 4.12: RV64 Supervisor address translation and protection register `satp`, for MODE values Sv39 and Sv48.

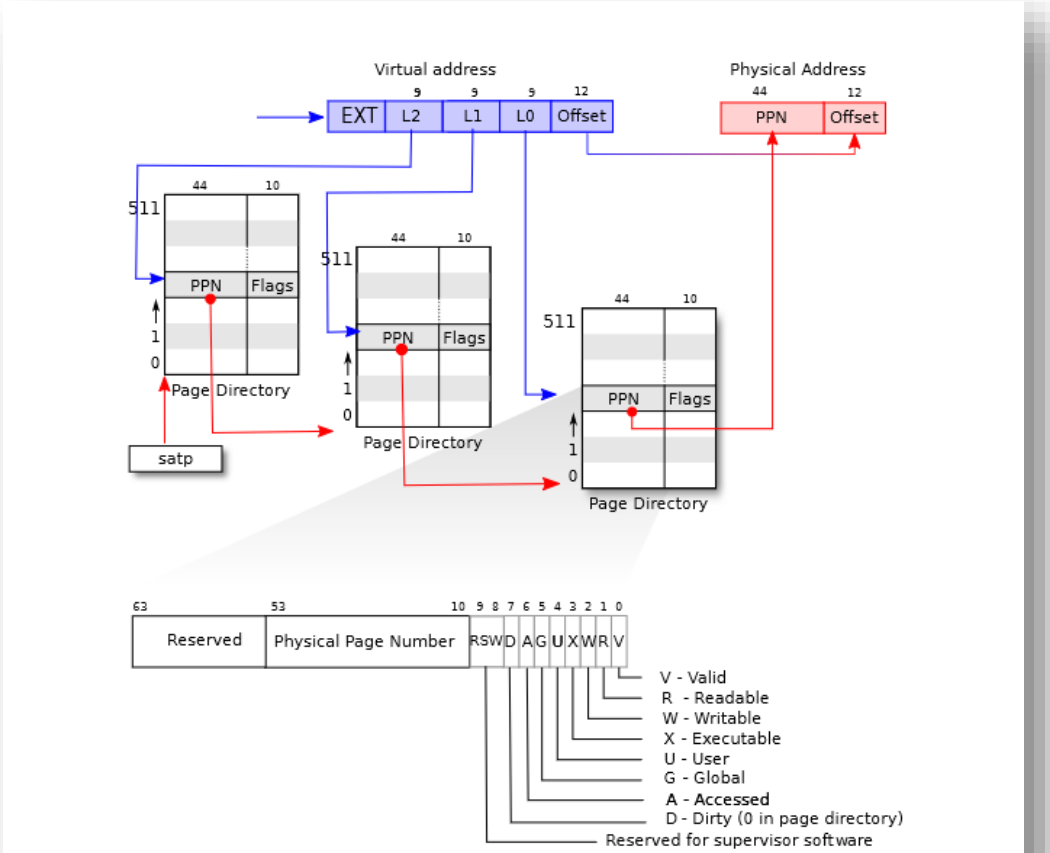


Figure 3.2: RISC-V address translation details.

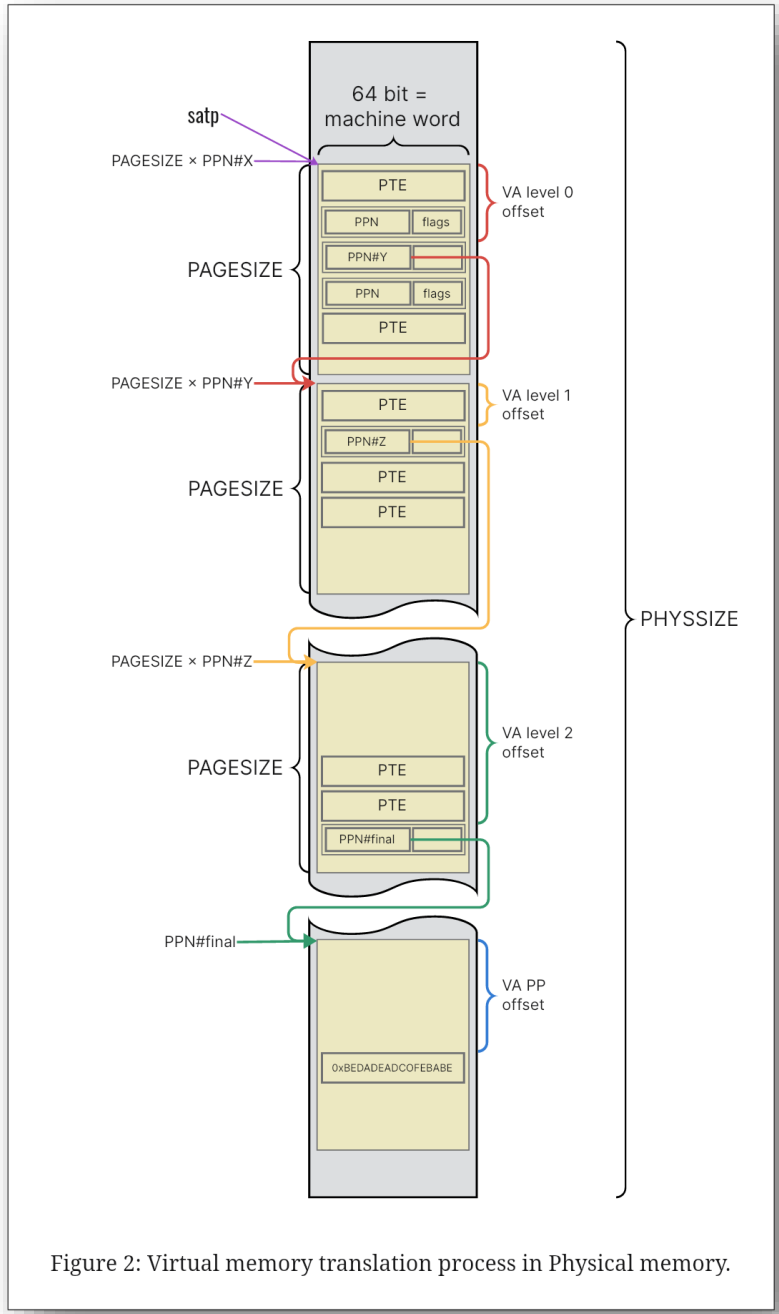


Figure 2: Virtual memory translation process in Physical memory.

satp = Supervisor Address Translation and Protection Register
PT = Page Table
PD = Page Directory = single-level PT
PTE = Page Table Entry
PP = Physical Page
PPN = Physical Page Number

Figure 1: Abbreviations in the context of virtual memory

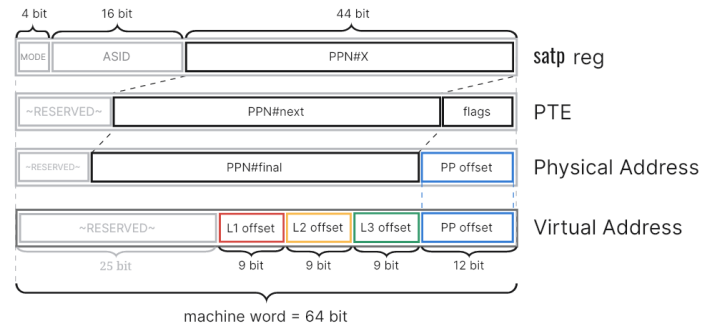


Figure 3: Virtual memory translation process in memory layout.

9 bit	⇒ 0..511	= #(PTE/PD)
512 × 64 bit	⇒ 4096 byte	= PAGESIZE
12 bit	⇒ 0..4095	= PAGESIZE
44 + 12 bit	⇒ ~75 PByte	= PHYSSIZE

Figure 4: Virtual memory translation meaningful constants

4.1.12 Supervisor Address Translation and Protection (satp) Register

The satp register is an XLEN-bit read/write register, formatted as shown in Figure 4.11 for RV32 and Figure 4.12, which controls supervisor-mode address translation and protection. This register holds the physical page number (PPN) of the root page table, i.e., its supervisor physical address divided by 4 KiB; an address space identifier (ASID), which facilitates address-translation fences on a per-address-space basis; and the MODE field, which selects the current address-translation scheme.

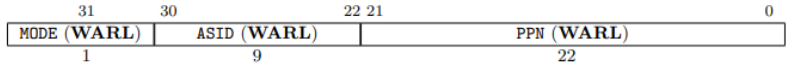


Figure 4.11: RV32 Supervisor address translation and protection register satp.

Storing a PPN in satp, rather than a physical address, supports a physical address space larger than 4 GiB for RV32.

We store the ASID and the page table base address in the same CSR to allow the pair to be changed atomically on a context switch. Swapping them non-atomically could pollute the old virtual address space with new translations, or vice-versa. This approach also slightly reduces the cost of a context switch.

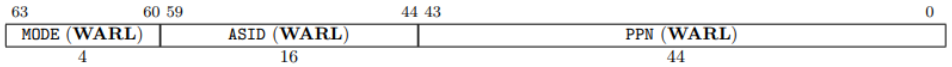


Figure 4.12: RV64 Supervisor address translation and protection register satp, for MODE values Sv39 and Sv48.

Table 4.3 shows the encodings of the MODE field for RV32 and RV64. When MODE=Bare, supervisor virtual addresses are equal to supervisor physical addresses, and there is no additional memory protection beyond the physical memory protection scheme described in Section 3.6. In this case, the remaining fields in satp have no effect.

For RV32, the only other valid setting for MODE is Sv32, a paged virtual-memory scheme described in Section 4.3.

For RV64, two paged virtual-memory schemes are defined: Sv39 and Sv48, described in Sections 4.4 and 4.5, respectively. Two additional schemes, Sv57 and Sv64, will be defined in a later version of this specification. The remaining MODE settings are reserved for future use and may define different interpretations of the other fields in satp.

Implementations are not required to support all MODE settings, and if satp is written with an unsupported MODE, the entire write has no effect; no fields in satp are modified.

RV32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing.

RV64		
Value	Name	Description
0	Bare	No translation or protection.
1–7	—	*Reserved*
8	Sv39	Page-based 39-bit virtual addressing.
9	Sv48	Page-based 48-bit virtual addressing.
10	*Sv57*	*Reserved for page-based 57-bit virtual addressing.*
11	*Sv64*	*Reserved for page-based 64-bit virtual addressing.*
12–15	—	*Reserved*

Table 4.3: Encoding of satp MODE field.

The number of supervisor physical address bits is implementation-defined; any unimplemented address bits are hardwired to zero in the satp register. The number of ASID bits is also implementation-defined and may be zero. The number of implemented ASID bits, termed *ASIDLEN*, may be determined by writing one to every bit position in the ASID field, then reading back the value in satp to see which bit positions in the ASID field hold a one. The least-significant bits of ASID are implemented first: that is, if ASIDLEN > 0, ASID[ASIDLEN-1:0] is writable. The maximal value of ASIDLEN, termed ASIDMAX, is 9 for Sv32 or 16 for Sv39 and Sv48

riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf

4.3.2 Virtual Address Translation Process

A virtual address *va* is translated into a physical address *pa* as follows:

1. Let a be `satp`.$ppn \times$ PAGESIZE, and let $i =$ LEVELS $- 1$. (For Sv32, PAGESIZE=2^{12} and LEVELS=2.)
2. Let *pte* be the value of the PTE at address $a + va.vpn[i] \times$ PTESIZE. (For Sv32, PTESIZE=4.) If accessing *pte* violates a PMA or PMP check, raise an access exception.
3. If $pte.v = 0$, or if $pte.r = 0$ and $pte.w = 1$, stop and raise a page-fault exception.
4. Otherwise, the PTE is valid. If $pte.r = 1$ or $pte.x = 1$, go to step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let $i = i - 1$. If $i < 0$, stop and raise a page-fault exception. Otherwise, let $a = pte.ppn \times$ PAGESIZE and go to step 2.
5. A leaf PTE has been found. Determine if the requested memory access is allowed by the *pte.r*, *pte.w*, *pte.x*, and *pte.u* bits, given the current privilege mode and the value of the SUM and MXR fields of the `mstatus` register. If not, stop and raise a page-fault exception.
6. If $i > 0$ and $pa.ppn[i - 1 : 0] \neq 0$, this is a misaligned superpage; stop and raise a page-fault exception.
7. If $pte.a = 0$, or if the memory access is a store and $pte.d = 0$, either raise a page-fault exception or:
   - Set *pte.a* to 1 and, if the memory access is a store, also set *pte.d* to 1.
   - If this access violates a PMA or PMP check, raise an access exception.
   - This update and the loading of *pte* in step 2 must be atomic; in particular, no intervening store to the PTE may be perceived to have occurred in-between.
8. The translation is successful. The translated physical address is given as follows:
   - $pa.pgoff = va.pgoff$.
   - If $i > 0$, then this is a superpage translation and $pa.ppn[i - 1 : 0] = va.vpn[i - 1 : 0]$.
   - $pa.ppn[\text{LEVELS} - 1 : i] = pte.ppn[\text{LEVELS} - 1 : i]$.

riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf

4.4.1 Addressing and Memory Protection

Sv39 implementations support a 39-bit virtual address space, divided into 4 KiB pages. An Sv39 address is partitioned as shown in Figure 4.16. Load and store effective addresses, which are 64 bits, must have bits 63–39 all equal to bit 38, or else a page-fault exception will occur. The 27-bit VPN is translated into a 44-bit PPN via a three-level page table, while the 12-bit page offset is untranslated.

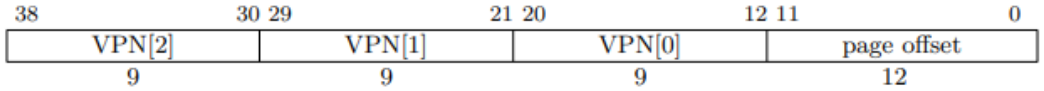


Figure 4.16: Sv39 virtual address.

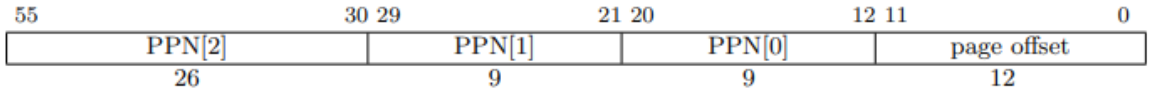


Figure 4.17: Sv39 physical address.

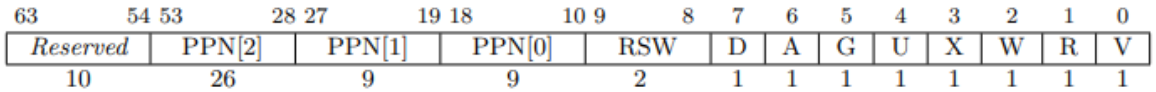


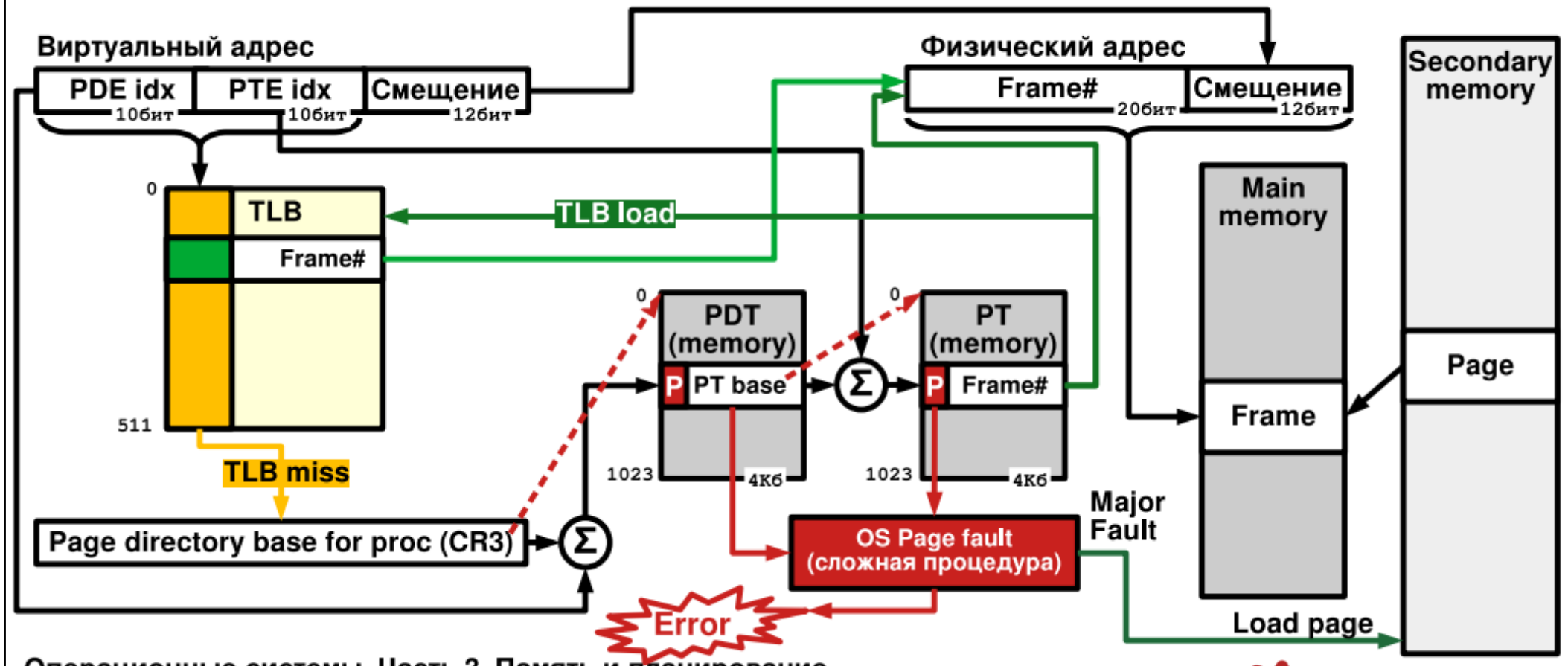
Figure 4.18: Sv39 page table entry.

Sv39 page tables contain 2^9 page table entries (PTEs), eight bytes each. A page table is exactly the size of a page and must always be aligned to a page boundary. The physical page number of the root page table is stored in the `satp` register's PPN field.

The PTE format for Sv39 is shown in Figure 4.18. Bits 9–0 have the same meaning as for Sv32. Bits 63–54 are reserved for future use and must be zeroed by software for forward compatibility.

riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf

2 level - virtual memory paging схема с TLB



Адресное пространство xv6

- Каждый процесс имеет определенную разметку адресов
- Ядро начинается после `0x80000000`
- Ниже находится память устройств
- Ядро имеет прямую адресацию по всей физической памяти
- У пользовательского процесса есть 2 стека: для User-space и Kernel-space
- Для защиты используются Guard page (неразмапленные страницы)

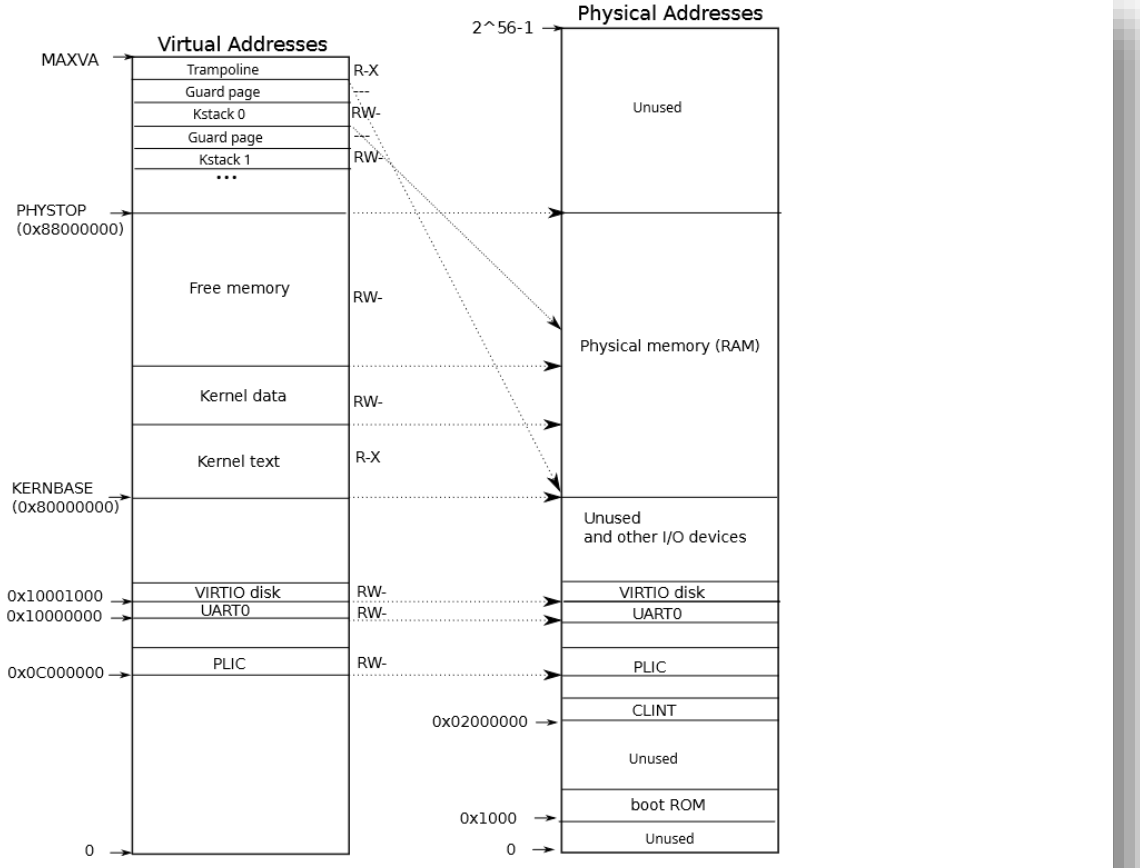


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

Адресное пространство xv6: детали

- Для создания Page Table для каждого процесса используются функции модуля `kernel/vm.c`
  - `walk` — функция проходящая (3 обращения) по Page Table для вычисления PPN
    - Отсюда *медленные* аллокации и *быстрая* виртуальная память
  - `kvmmap`, `kvmmake`, … — разметка VA ядра
  - `uvmmap`, `uvmcreate`, … — разметка VA пользователя

- Для аллокаций памяти используются функции из `kernel/kalloc.c`
  - `kalloc` — размечает свободную страницу в виртуальной памяти
  - `kfree` — удаляет разметку страницы из вирт. памяти

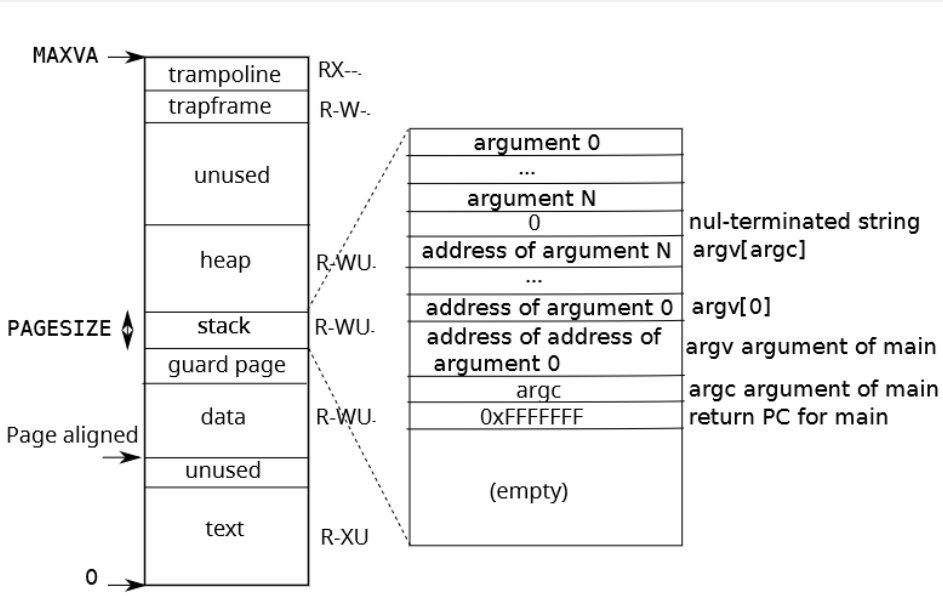
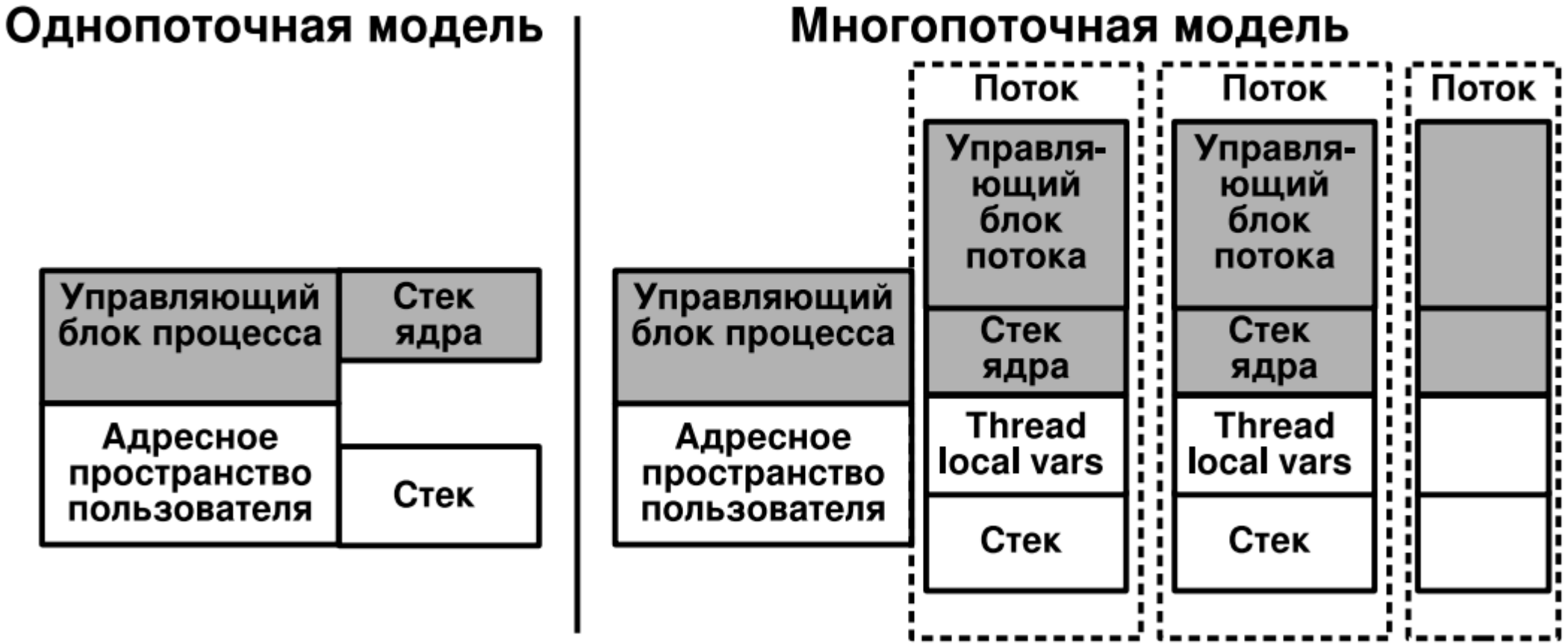


Figure 3.4: A process's user address space, with its initial stack.

Связь структур ядра процесса и потока

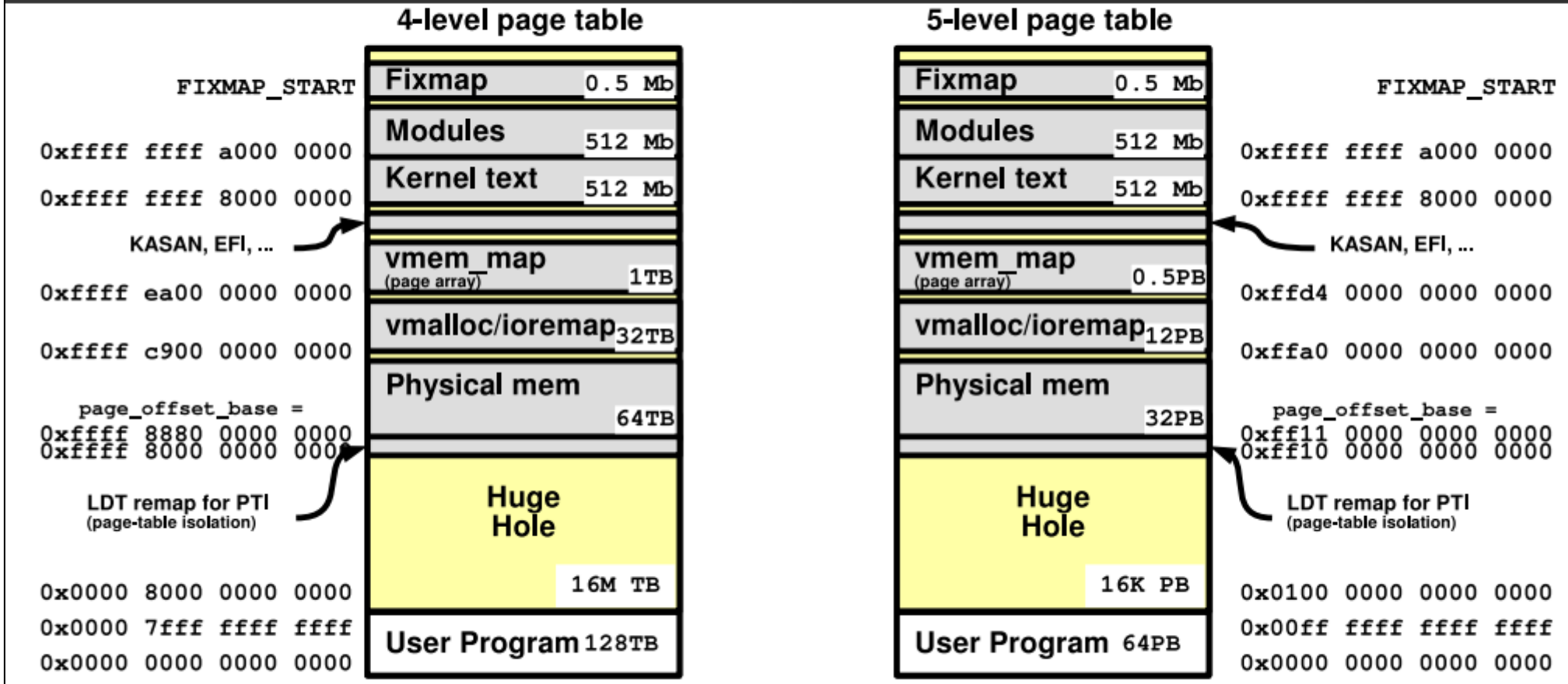


Операционные системы. Часть 2. Процессы и потоки

tuneit 33

Memory Layout 64bit

https://elixir.bootlin.com/linux/v5.4.74/source/Documentation/x86/x86_64/mm.rst



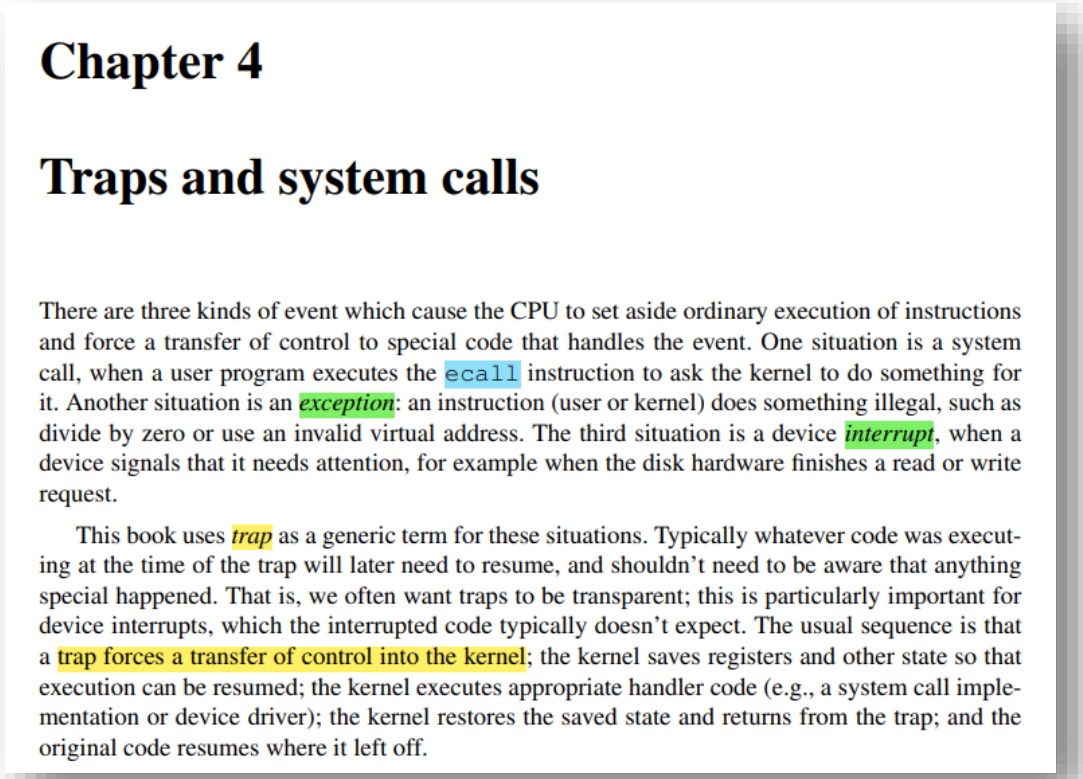
Операционные системы. Часть 3. Память и планирование

tuneit 31

Система прерываний

Система прерываний

- 3 типа событий вызывающих *trap*:
  - Системный вызов (`ecall`)
  - Исключение процессора
  - Прерывание устройством
- Повышается уровень разрешений процессора
- Выполнение передается специальному коду ядра
- Ядро проверяет причину прерывания и соответствующе реагирует



Chapter 4

Traps and system calls

There are three kinds of event which cause the CPU to set aside ordinary execution of instructions and force a transfer of control to special code that handles the event. One situation is a system call, when a user program executes the `ecall` instruction to ask the kernel to do something for it. Another situation is an *exception*: an instruction (user or kernel) does something illegal, such as divide by zero or use an invalid virtual address. The third situation is a device *interrupt*, when a device signals that it needs attention, for example when the disk hardware finishes a read or write request.

This book uses *trap* as a generic term for these situations. Typically whatever code was executing at the time of the trap will later need to resume, and shouldn't need to be aware that anything special happened. That is, we often want traps to be transparent; this is particularly important for device interrupts, which the interrupted code typically doesn't expect. The usual sequence is that a trap forces a transfer of control into the kernel; the kernel saves registers and other state so that execution can be resumed; the kernel executes appropriate handler code (e.g., a system call implementation or device driver); the kernel restores the saved state and returns from the trap; and the original code resumes where it left off.

Control and Status Registers (CSRs)

- **stvec** — адрес обработчика прерывания, установленный ядром;
- **sepc**: — сохраненный program counter во время перрывания, **sret** вернет при выходе из *trap* значение обратно **sepc → pc;**
- **scause** — код причины прерывания;
- **sscratch** — вспомогательный регистр для временного сохранения значения другого регистра.
- **sstatus** — содержит
  - бит SIE отвечающий за в[ы]ключение прерываний,
  - бит SPP отвечающий за режим из которого пришло исключение, туда нужно будет вернуться с помощью **sret**.

Number	Privilege	Name	Description
Supervisor Trap Setup			
0x100	SRW	sstatus	Supervisor status register.
0x102	SRW	sedeleg	Supervisor exception delegation register.
0x103	SRW	sideleg	Supervisor interrupt delegation register.
0x104	SRW	sie	Supervisor interrupt-enable register.
0x105	SRW	stvec	Supervisor trap handler base address.
Supervisor Trap Handling			
0x140	SRW	sscratch	Scratch register for supervisor trap handlers.
0x141	SRW	sepc	Supervisor exception program counter.
0x142	SRW	scause	Supervisor trap cause.
0x143	SRW	sbadaddr	Supervisor bad address.
0x144	SRW	sip	Supervisor interrupt pending.
Supervisor Protection and Translation			
0x180	SRW	sptbr	Page-table base register.
Supervisor Counter/Timers			
0xD00	SRO	scycle	Supervisor cycle counter.
0xD01	SRO	stime	Supervisor wall-clock time.
0xD02	SRO	sinstret	Supervisor instructions-retired counter.
0xD80	SRO	scycleh	Upper 32 bits of scycle, RV32I only.
0xD81	SRO	stimeh	Upper 32 bits of stime, RV32I only.
0xD82	SRO	sinstreth	Upper 32 bits of sinstret, RV32I only.

Table 2.3: Currently allocated RISC-V supervisor-level CSR addresses.

Прерывания: ловкость рук и никакого мошенничества

When it needs to force a trap, the RISC-V hardware does the following for all trap types (other than timer interrupts):

1. If the trap is a device interrupt, and the `sstatus` SIE bit is clear, don't do any of the following.
2. Disable interrupts by clearing the SIE bit in `sstatus`.
3. Copy the `pc` to `sepc`.
4. Save the current mode (user or supervisor) in the SPP bit in `sstatus`.
5. Set `scause` to reflect the trap's cause.
6. Set the mode to supervisor.
7. Copy `stvec` to the `pc`.
8. Start executing at the new `pc`.

- Тут оказывается важным то, что CPU не сменяет Page Table после перехода в ядро и не сохраняет автоматически никакие регистры кроме pc
- Более легковесное прерывание
- Обработчики (*trampoline*) должны быть на одинаковых адресах

- Аналогично работает для прерываний по таймеру, но в Machine mode

Прерывания: user space

- Для пользовательских процессов требуется дополнительно сохранять состояние перед переходом в ядро в отдельную страницу *trapframe*
- *trampoline* размаплена у каждого процесса и в ядре, поэтому можно без проблем продолжать выполнение
- Путь при прерывании в user space:
  1. `uservec` (kernel/trampoline.S:21),
  2. `usertrap` (kernel/trap.c:37);
- Путь при возврате из kernel space:
  1. `usertrapret` (kernel/trap.c:90),
  2. `userret` (kernel/trampoline.S:101);

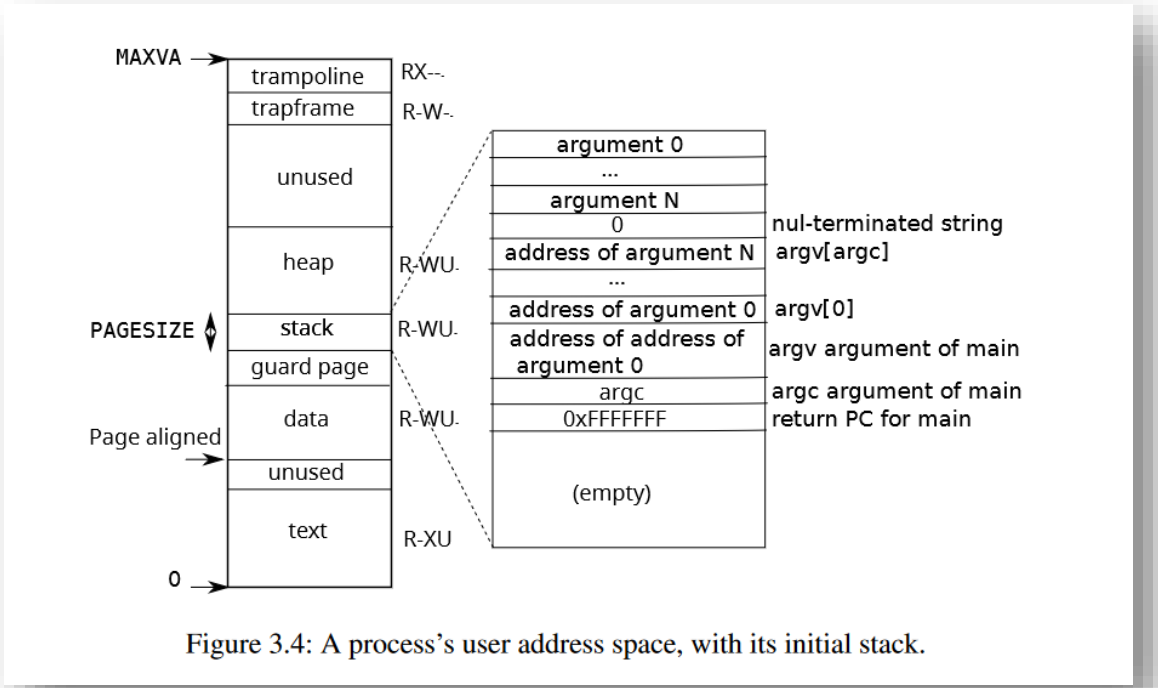


Figure 3.4: A process's user address space, with its initial stack.

Прерывания: вывод

- прерывания, режимы работы и виртуальная память тесно связаны при обработке исключений

- Copy-on-Write (CoW) и lazy allocation
  — обработка Page Fault исключения
- Работа с устройствами
- Разделение времени между процессами
- …

Concurrency

Concurrency

- Xv6 по умолчанию работает на 3 ядерном процессоре
- Thread = CPU
- Состояния гонки

⇒ нужна синхронизация

```
155     ifndef CPUS
156     CPUS := 3
157     endif
```

```
(gdb) info threads
  Id   Target Id                    Frame
* 1    Thread 1.1 (CPU#0 [running]) 0x0000000000001000 in ?? ()
  2    Thread 1.2 (CPU#1 [running]) 0x0000000000001000 in ?? ()
  3    Thread 1.3 (CPU#2 [running]) 0x0000000000001000 in ?? ()
```

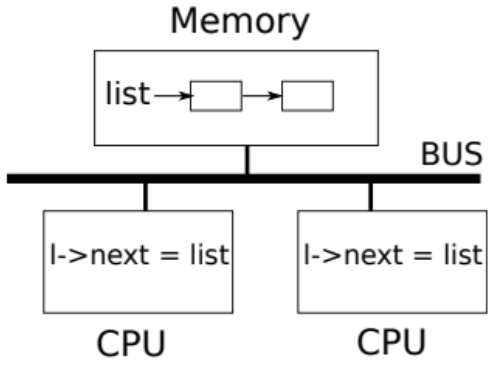


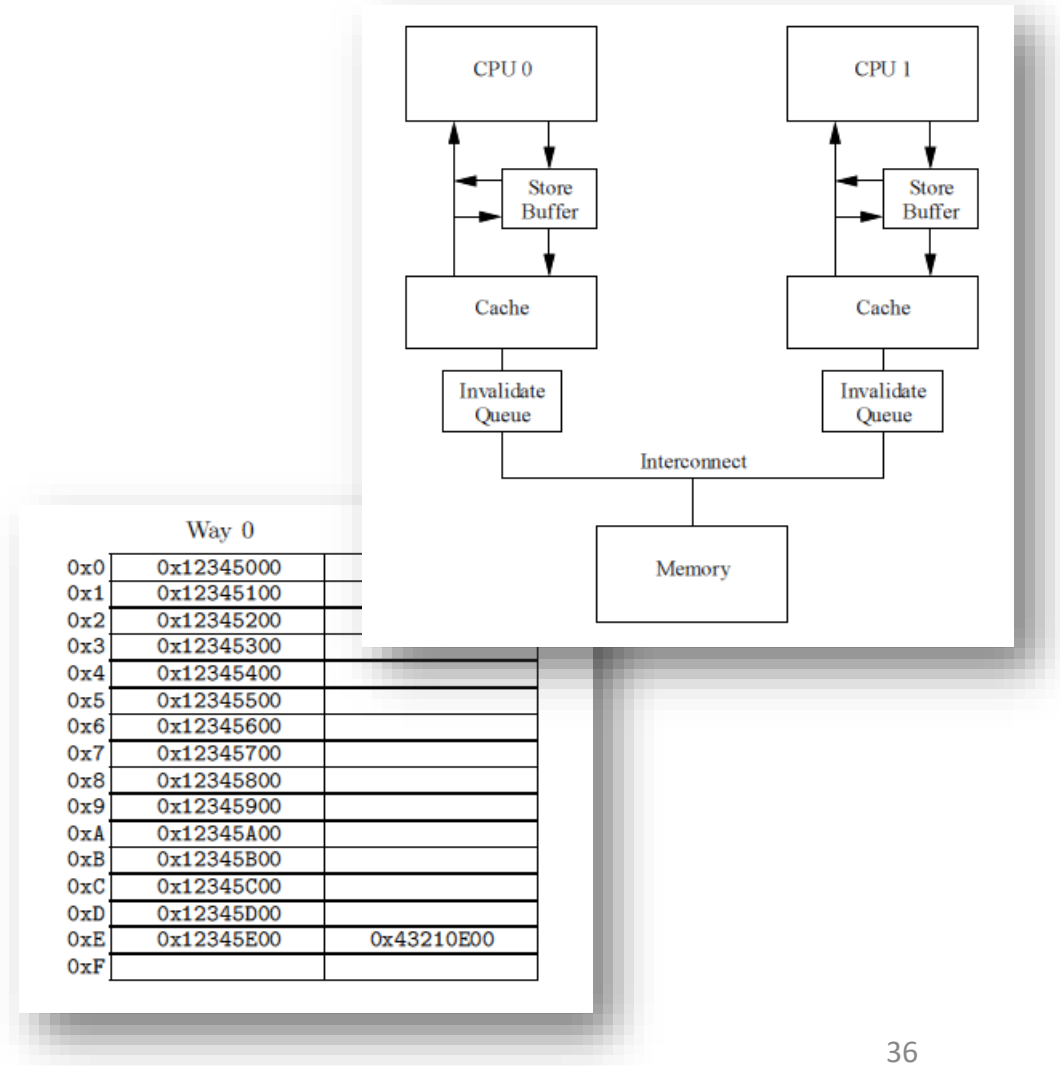
Figure 6.1: Simplified SMP architecture

Concurrency: барьеры

- Барьеры памяти:
- Платформозависимые инструкции для работы с кэшем процессора
- Протоколы согласованности кешей

- Барьеры компилятора:
- Любые ассемблерные вставки

habr.com/ru/articles/196548/

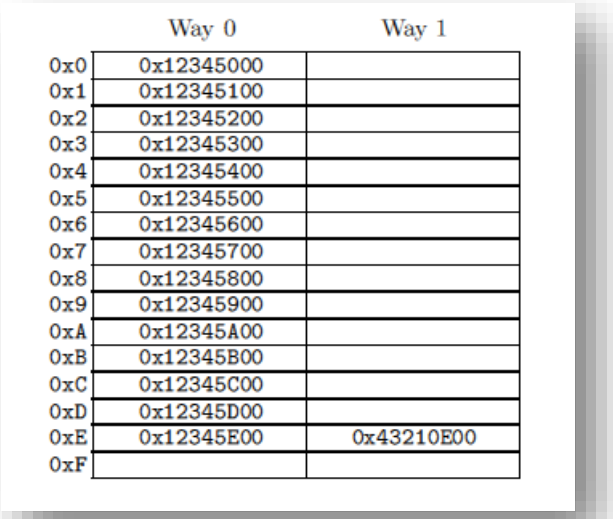


Concurrency: атомарные операции

- Либо полностью завершилась, либо не происходила

- Compare-and-Swap (ABA-problem)
- Load-Linked / Store-Conditional
- Стандартизованные семантики и модели памяти
  - gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html
  - cppreference.com/w/cpp/atomic/atomic

- Выравнивание памяти!
- Lifelock на ровном месте: две атомарных переменных в одной кеш линии



	Way 0	Way 1
0x0	0x12345000	
0x1	0x12345100	
0x2	0x12345200	
0x3	0x12345300	
0x4	0x12345400	
0x5	0x12345500	
0x6	0x12345600	
0x7	0x12345700	
0x8	0x12345800	
0x9	0x12345900	
0xA	0x12345A00	
0xB	0x12345B00	
0xC	0x12345C00	
0xD	0x12345D00	
0xE	0x12345E00	0x43210E00
0xF		

habr.com/ru/articles/195948/

Блокировки

- Специальные примитивы останавливающие выполнение кода потоком

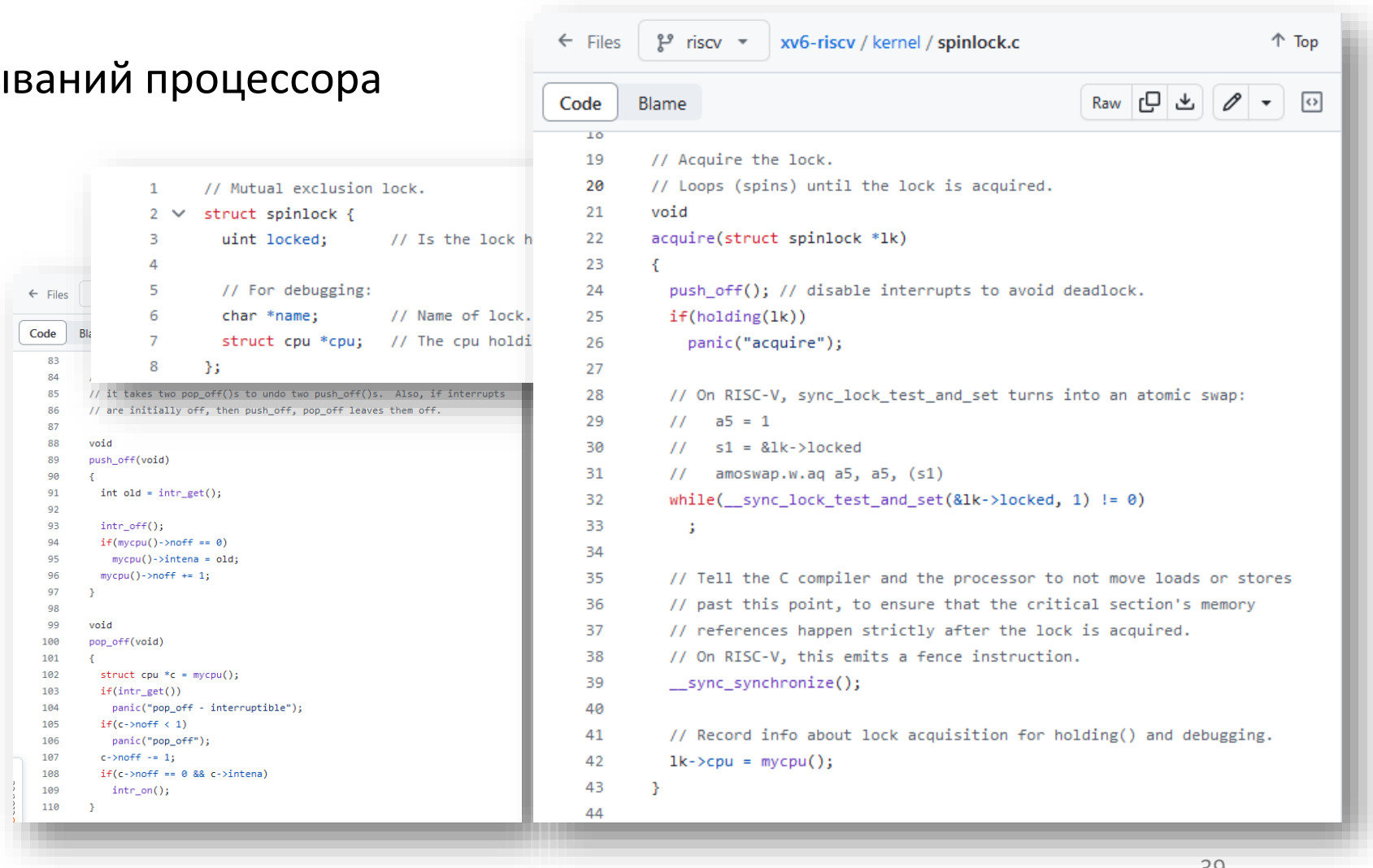
- Сохранение инварианта и последовательности

- На основе атомарной операции в цикле
  - spinlock
- или изменение состояния процесса в ОС
  - sleeplock
- …

Синхронизация в xv6: spinlock

- Простое отключение прерываний процессора
  - `push_off()`
- Атомарные операции

`__sync_lock_test_and_set`
`__sync_synchronize`
`(amoswap r, a)`



* Номер процессора нельзя получить в S-mode, поэтому он сохраняется во время M-mode в специальный регистр, который не может быть изменен в U-mode (`tp` ↦ `mycpu()`)

Аппаратная поддержка взаимных исключений

- Запрет прерываний на однопроцессорных системах
  - DI; Критический участок; EI
- Атомарные инструкции TAS, CAS, CMPXCHG, ...

```
spin_acqure_lock: // locked = 1, released = 0
      movl    $1, %ebx
      movl    lock_var_addr, %ecx
loop: pause   //rep nop
      movl    (%ecx), %eax //avoid unness. lock
      test    %eax, %eax
      jnz     loop
      lock cmpxchg %ebx, (%ecx) // %eax = 0
      jnz     loop
      ret

spin_unlock:
      mfence
      movl$0, (lock_var_addr)
      ret
```

```
/* Atomic compare-and-exchange: */
Compare eax with memory (%ecx)
if equal
  load %ebx in memory (%ecx), ZF=1
else
  load memory in %eax, ZF=0


pause – hint CPU we are spining
mfence – load/store bariers
```

https://www.cs.virginia.edu/~cr4bd/4414/F2018/slides/20180925--slides-1up.pdf

Синхронизация в xv6: sleeplock

- Защита состояния через spinlock
- Проверка состояния и передача управления планировщику в цикле

```
1    // Long-term locks for processes
2    struct sleeplock {
3      uint locked;       // Is the lock held?
4      struct spinlock lk; // spinlock protecting this sleep lock
5
6      // For debugging:
7      char *name;        // Name of lock.
8      int pid;           // Process holding lock
9    };
```

Files | riscv | xv6-riscv / kernel / sleeplock.c | Top

Code | Blame | Raw

```
20
21   void
22   acquiresleep(struct sleeplock *lk)
23   {
24     acquire(&lk->lk);
25     while (lk->locked) {
26       sleep(lk, &lk->lk);
27     }
28     lk->locked = 1;
29     lk->pid = myproc()->pid;
30     release(&lk->lk);
31   }
32
33   void
34   releasesleep(struct sleeplock *lk)
35   {
36     acquire(&lk->lk);
37     lk->locked = 0;
38     lk->pid = 0;
39     wakeup(lk);
40     release(&lk->lk);
41   }
42
```

Планировщик

Планировщик

- Разделение времени процессора между процессами
- Переключение за счет смены контекста (регистров) потока

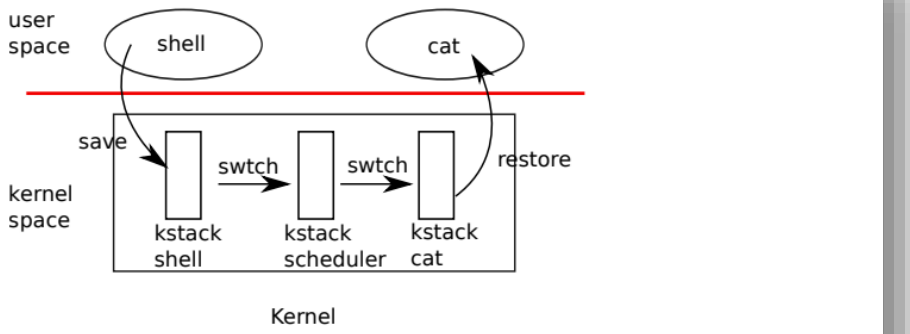


Figure 7.1: Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

* Процесс: *много задач на одном компьютере*
* Поток: *одна задача на разных процессорах*
* *Ядро CPU = процессор*

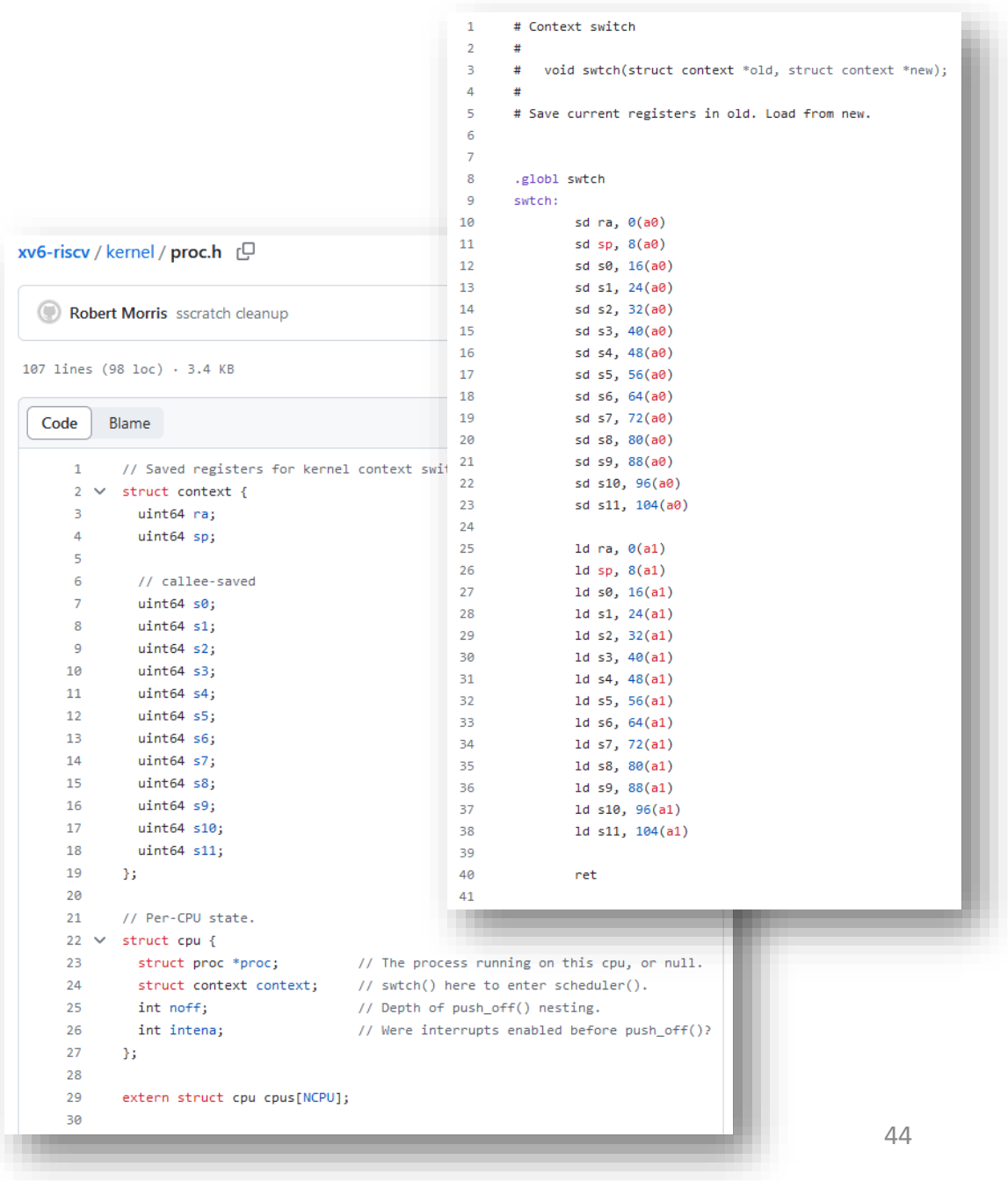
* Scheduler-space*: event-loop на каждый процессор (между kernel/user-space'ами)*

Планировщик в xv6

- Round robin
- 6 состояний процесса
- В процессе 1 поток

- **kernel/proc.c**:

```
scheduler() ↔ sched()
                  ⇅
               swtch()
                  ⇅
yield()/sleep()/exit()
```



xv6-riscv / kernel / proc.h

Robert Morris sscratch cleanup

107 lines (98 loc) · 3.4 KB

Code Blame

```
// Saved registers for kernel context swi
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};

// Per-CPU state.
struct cpu {
  struct proc *proc;          // The process running on this cpu, or null.
  struct context context;     // swtch() here to enter scheduler().
  int noff;                   // Depth of push_off() nesting.
  int intena;                 // Were interrupts enabled before push_off()?
};

extern struct cpu cpus[NCPU];
```

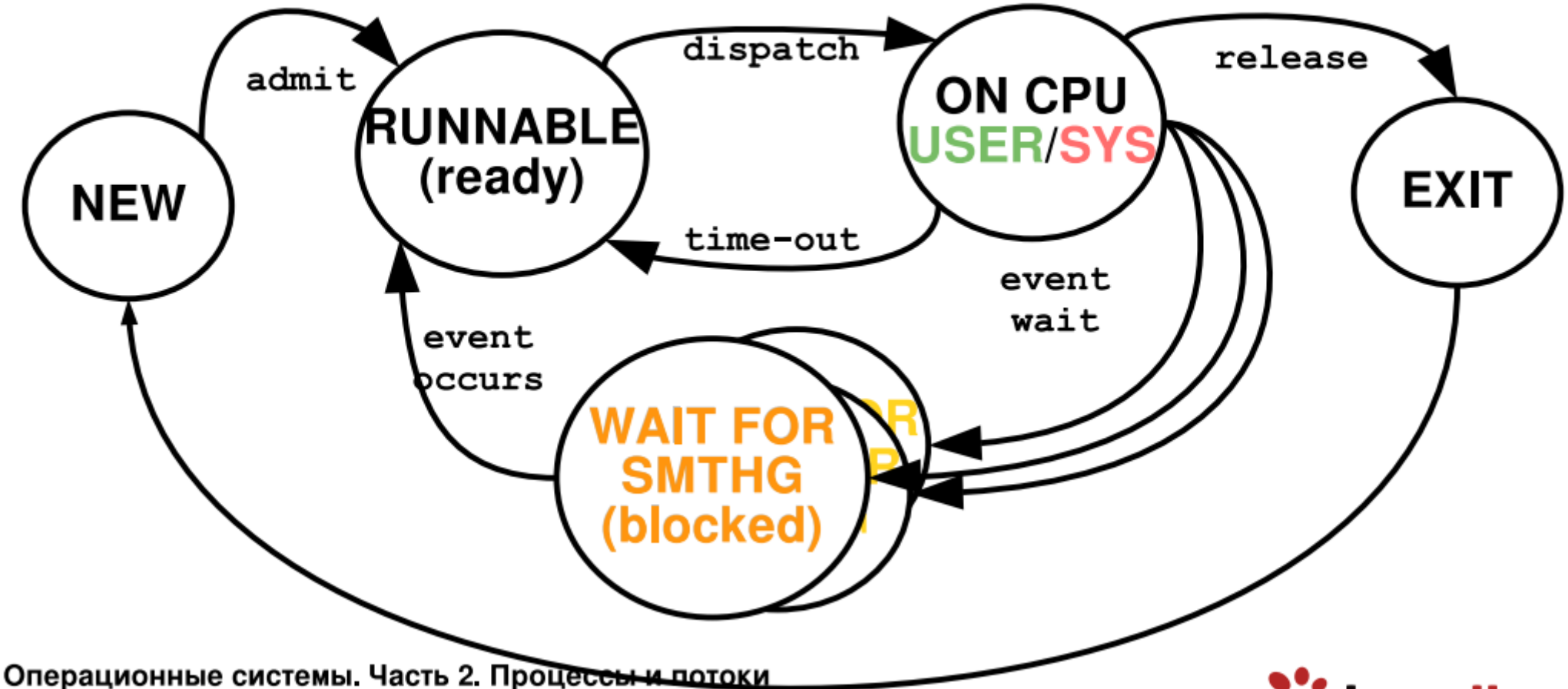
```
# Context switch
#
#   void swtch(struct context *old, struct context *new);
#
# Save current registers in old. Load from new.


.globl swtch
swtch:
        sd ra, 0(a0)
        sd sp, 8(a0)
        sd s0, 16(a0)
        sd s1, 24(a0)
        sd s2, 32(a0)
        sd s3, 40(a0)
        sd s4, 48(a0)
        sd s5, 56(a0)
        sd s6, 64(a0)
        sd s7, 72(a0)
        sd s8, 80(a0)
        sd s9, 88(a0)
        sd s10, 96(a0)
        sd s11, 104(a0)

        ld ra, 0(a1)
        ld sp, 8(a1)
        ld s0, 16(a1)
        ld s1, 24(a1)
        ld s2, 32(a1)
        ld s3, 40(a1)
        ld s4, 48(a1)
        ld s5, 56(a1)
        ld s6, 64(a1)
        ld s7, 72(a1)
        ld s8, 80(a1)
        ld s9, 88(a1)
        ld s10, 96(a1)
        ld s11, 104(a1)

        ret
```

Модель процесса с 5-ю состояниями



Операционные системы. Часть 2. Процессы и потоки

tuneit 7

Итог

Итог

- Запуск операционной системы с нуля
- Режимы процессора
- Устройство виртуальной памяти
- Система прерываний
- Многопоточная работа ОС
- *Файловая система

Дополнение

- github.com/embox/embox

- Персистентная Phantom OS
- youtu.be/DBOTcG6iJEI — 007. За гранью Intel и Linux – необычные процессоры и операционные системы – Дмитрий Завалишин,
- youtu.be/6SZZ7ASOR7s — Архитектура процессора Эльбрус 2000 / Дмитрий Завалишин,
- www.youtube.com/live/iJ-WJsRYCVU?t=18818 — Selectel про будущее ARM
- youtu.be/_v7ddWj8buk — JPoint, JVM на RISC-V / Syntacore

- habr.com/ru/users/khizmax/publications/articles/ — lockfree во всех деталях
- akkadia.org/drepper/cpumemory.pdf — What Every Programmer Should Know About Memory

Спасибо за внимание

Давайте обсудим, что вы сейчас услышали