6.S081: Syscall Entry/Exit

Adam Belay

abelay@mit.edu

Today’s agenda

- Last week:
  - Page table hardware
  - Calling conventions + stack frames
- Today:
  - Assembly programming
  - Privileged CPU features
  - System call entry/exit
- Reminder:
  - Page table lab due this Thursday

Assembly programming

Two main approaches:

1. Inline-assembly: assembly inside C source files
    - Compiler extension to call assembly instructions
    - Compiler still helps with some things automatically (e.g., allocating, saving, and restoring registers)
    - e.g., riscv.h
2. Assembly source: separate assembly file
    - Code written directly in assembly
    - Everything must be done manually by programmer, even **calling conventions**
    - e.g., trampoline.S

Extended ASM syntax

```
asm(“assembly code” : outputs :
inputs : clobbers);
```

- Outputs: registers or memory that contain outputs
- Inputs: registers or memory that contain inputs
- Clobbers: registers or memory that are overwritten, not explicitly marked as inputs, and you don’t care about their values
- Volatile: qualifier that disables certain optimizations

Extended ASM modifiers

- Outputs:
  - “=r”: specifies a register (compiler decides which one)
  - “=m” specifies a memory location
  - “=rm” specifies either a register or memory location
- Inputs:
  - “r”: specifies a register (compiler decides which one)
  - “m” specifies a memory location
  - “rm” specifies either a register or memory location
- Clobbers:
  - Specific registers
  - “memory” or “cc”

Extended ASM example

```
static inline void w_satp(uint64 x) {
  asm volatile("csrw satp, %0" : : "r" (x));
}
```

- What does this code do?
- %0: specifies the 0th argument in outputs + inputs

Same example in .S file

```
csrw satp, a1
```

- No compiler help allocating, saving, or restoring registers
- Code manually specifies a1, overwriting its value
- Compiler cannot reorder instruction

Privileged CPU features

Two main parts:

1. Registers that can only be accessed by kernel
2. Instructions that can only be executed by kernel

What do these look like in RISC-V?

Important priv. RISC-V features

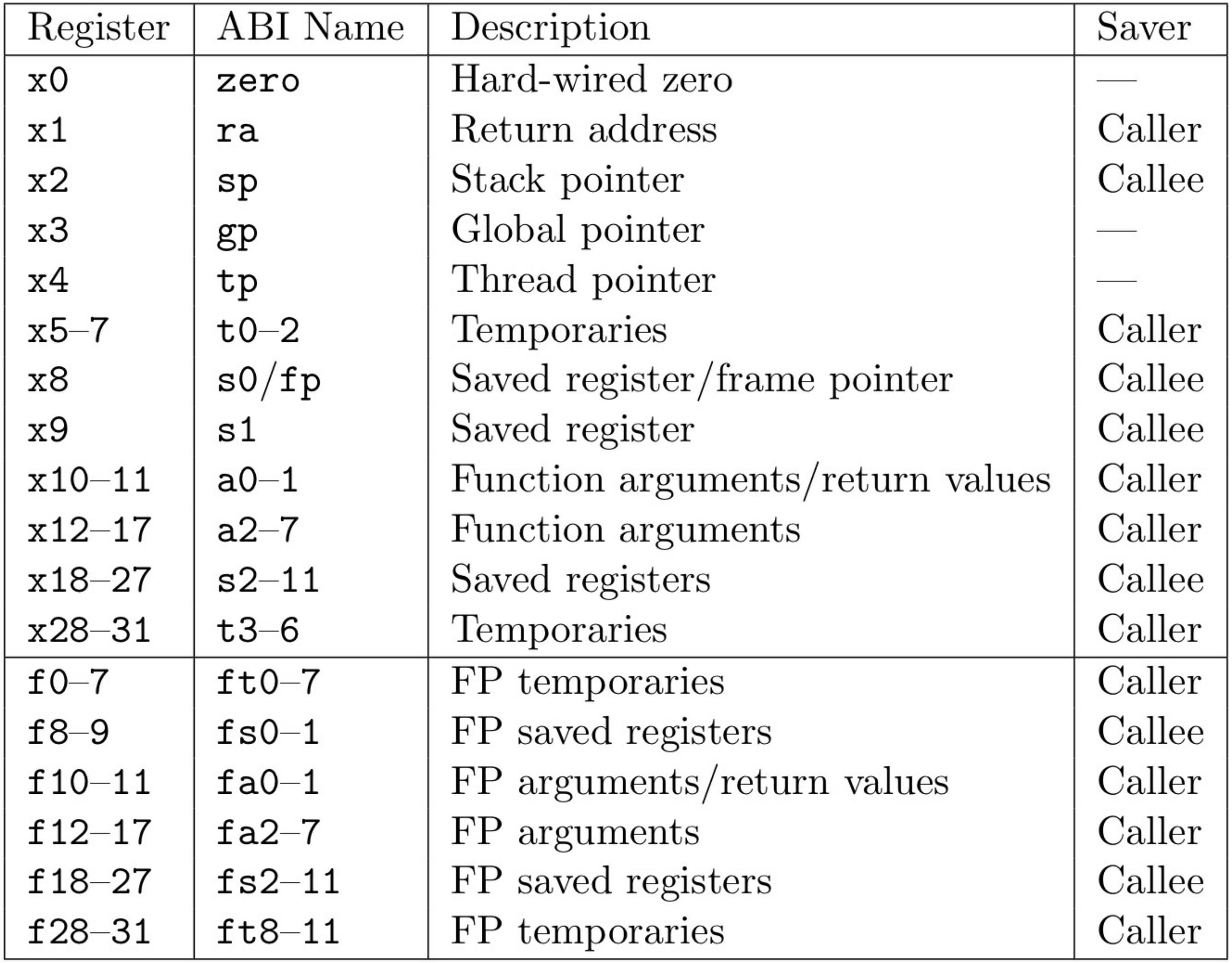
Privileged registers:

- **satp**: physical address of page table root
- **stvec**: ecall jumps here, points to trampoline
- **sepc**: ecall saves the user’s PC here
- **sscratch**: scratch space; used to store temporary data

Privileged instructions:

- **Access regs**: csrr (read), csrw (write), csrrw (swap)
- **sret**: return to userspace

Recall: Calling conventions

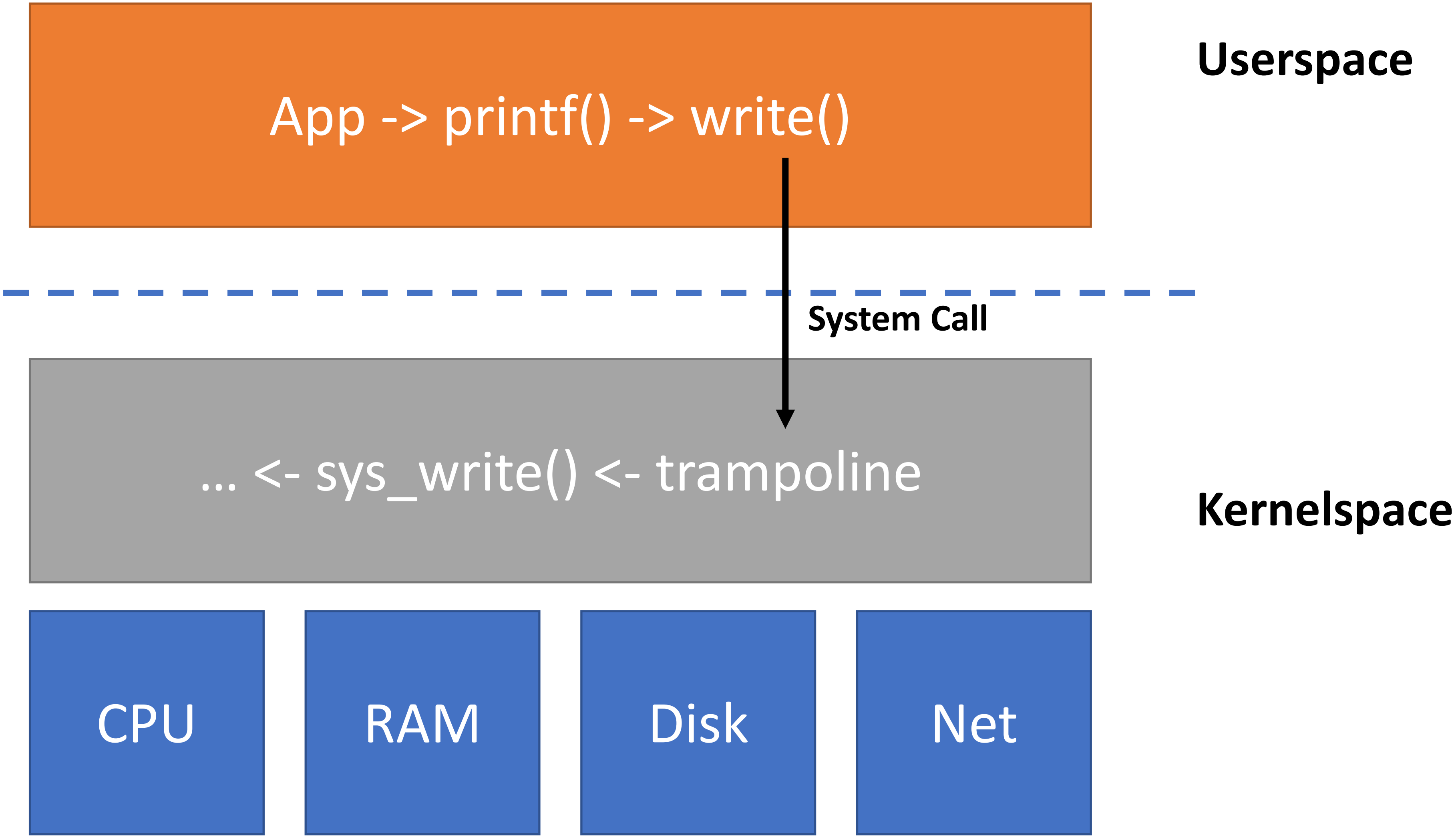


Register	ABI Name	Description	Saver
`x0`	`zero`	Hard-wired zero	—
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	—
`x4`	`tp`	Thread pointer	—
`x5–7`	`t0–2`	Temporaries	Caller
`x8`	`s0/fp`	Saved register/frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10–11`	`a0–1`	Function arguments/return values	Caller
`x12–17`	`a2–7`	Function arguments	Caller
`x18–27`	`s2–11`	Saved registers	Callee
`x28–31`	`t3–6`	Temporaries	Caller
`f0–7`	`ft0–7`	FP temporaries	Caller
`f8–9`	`fs0–1`	FP saved registers	Callee
`f10–11`	`fa0–1`	FP arguments/return values	Caller
`f12–17`	`fa2–7`	FP arguments	Caller
`f18–27`	`fs2–11`	FP saved registers	Callee
`f28–31`	`ft8–11`	FP temporaries	Caller

User -> kernel transitions

- System calls, faults, and interrupts enter kernel same way
- Important for isolation and performance
- Lots of careful design choices; details matter!

Recall: System calls



App -> printf() -> write()

System Call

… <- sys_write() <- trampoline

CPU

RAM

Disk

Net

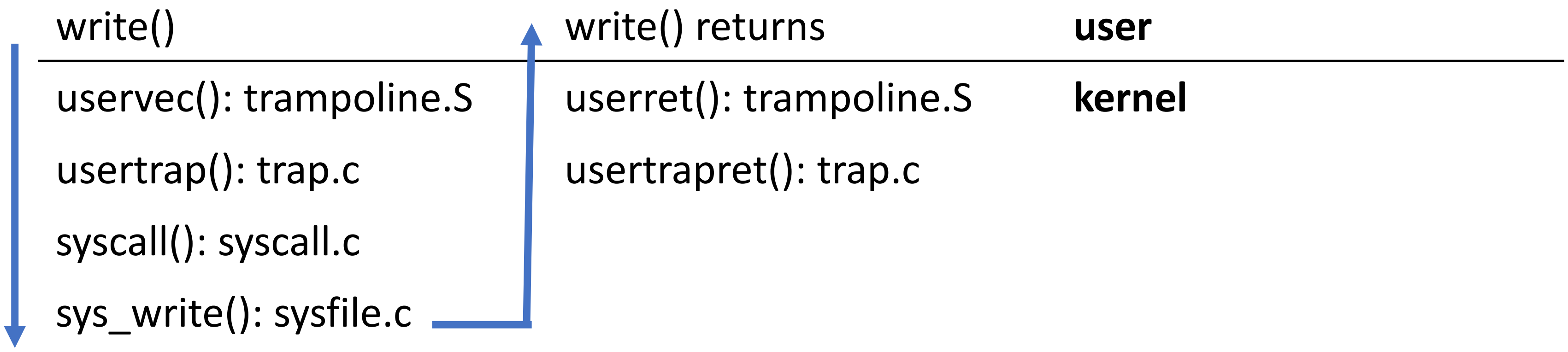
Userspace

Kernelspace

What needs to happen?

- Issue: CPU registers are set up for user, not kernel
- 32 registers, sp, pc, privilege mode, satp, stvec, sepc, ...
- Steps needed:
  - Switch to supervisor mode
  - Save 32 registers + pc
  - Switch to kernel page table and stack
  - Jump to kernel C code
- User code can’t interfere with transition
- Must be transparent, resume without disturbing user code (traps and interrupts can’t be observable)

Example: write() system call



write()

uservec(): trampoline.S

usertrap(): trap.c

syscall(): syscall.c

sys_write(): sysfile.c

write() returns

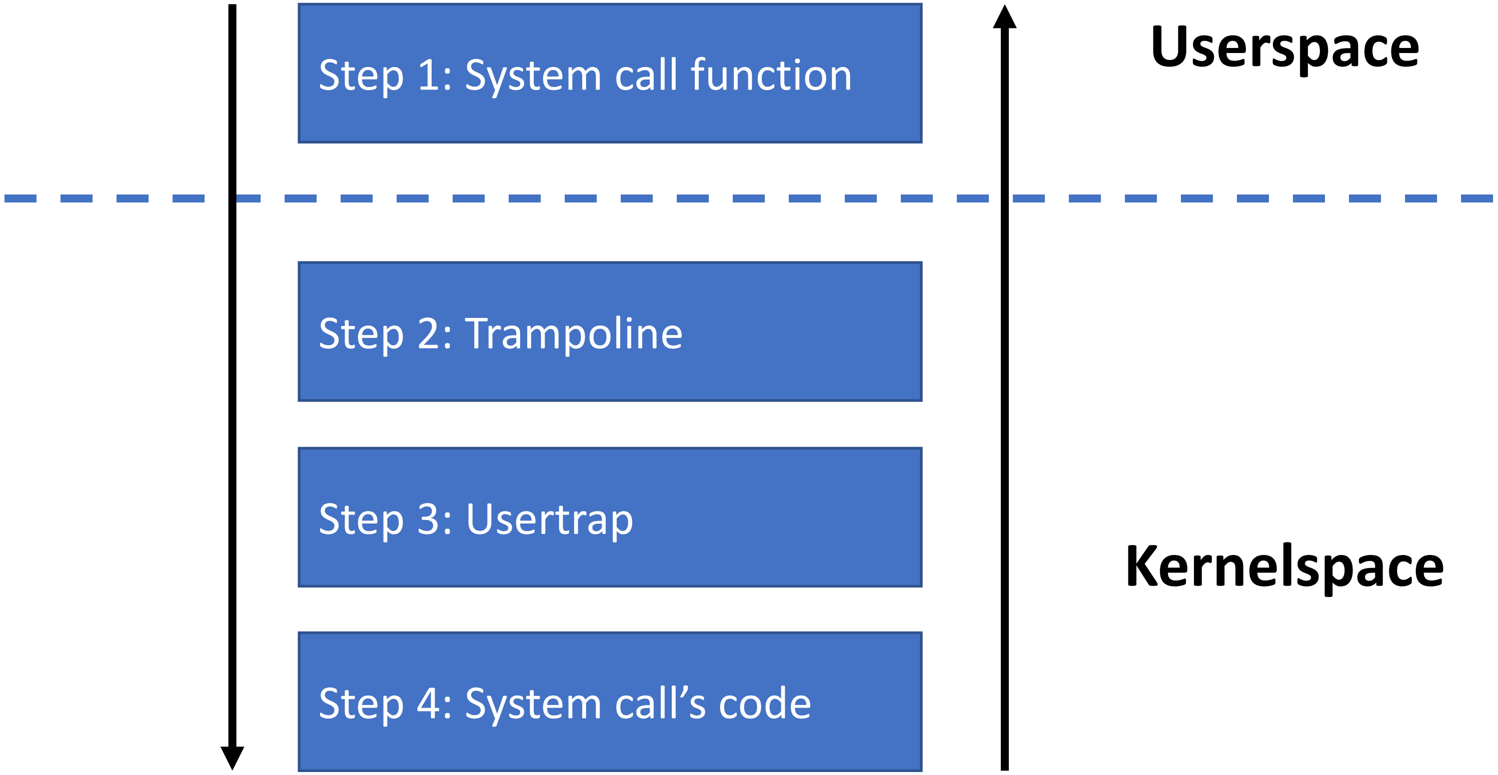
userret(): trampoline.S

usertrapret(): trap.c

user

kernel

Steps in a system call



Step 1: System call function

Step 2: Trampoline

Step 3: Usertrap

Step 4: System call's code

Userspace

Kernelspace

Step 1: System call function

- Use **ecall** instruction to enter supervisor mode
- a7 register stores system call number. Why?
- System call’s arguments stored in a0-a6
- Return value stored in a0
- No other registers changed after return! Why?

Step 1: System call func (usys.S)

```
…
.global write
write:
 li a7, SYS_write
 ecall
 ret
…
```

ecall does very little

- ecall performs just two actions:
  - Change to supervisor mode
  - Jump to **stvec** location
  - That’s it!
- Why so simple?
  - Give designers flexibility to optimize their kernel
- Optimization ideas:
  - No pgtbl switch: Use one table for user and kernel
  - Don’t use a stack for some simple system calls
  - Optimize which registers need to be saved

Step 2: Trampoline

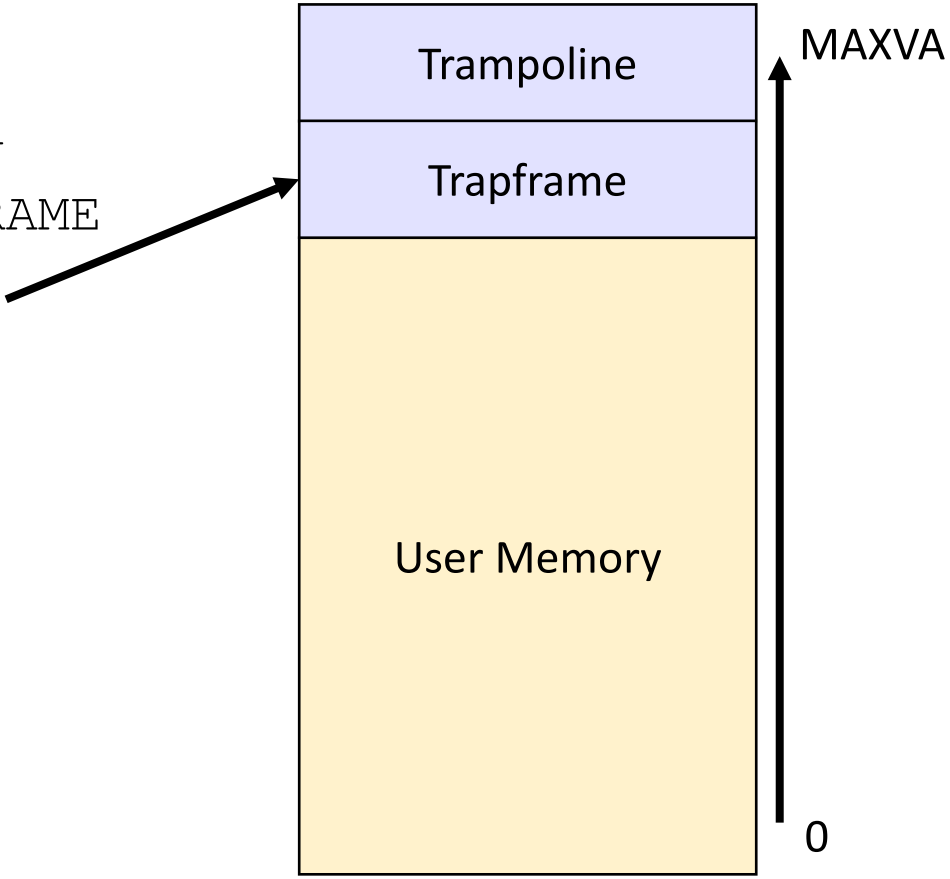
- Must save registers to the trapframe. Why?
  - So they can be restored before returning
  - So the kernel can retrieve the syscall # and arguments
- Must restore the kernel’s stack. Why?
  - So normal kernel code can execute
- Must restore the kernel’s page table. Why?
  - Kernel’s code and data aren’t mapped in user pgtbl
- Must jump into usertrap(). Why?
  - Figures out how to handle different types of traps

Step 2: Trampoline (trampoline.S)

…

```
# swap a0 and sscratch
# so that a0 is TRAPFRAME
csrrw a0, sscratch, a0
```

…



Contents of trapframe

```
struct trapframe {
  /*   0 */ uint64 kernel_satp;   // kernel page table
  /*   8 */ uint64 kernel_sp;     // top of process's kernel stack
  /*  16 */ uint64 kernel_trap;   // usertrap()
  /*  24 */ uint64 epc;           // saved user program counter
  /*  32 */ uint64 kernel_hartid; // saved kernel tp
  /*  40 */ uint64 ra;
  /*  48 */ uint64 sp;
  /*  56 */ uint64 gp;
  /*  64 */ uint64 tp;
  /*  72 */ uint64 t0;
…
};
```

Step 2: Trampoline (trampoline.S)

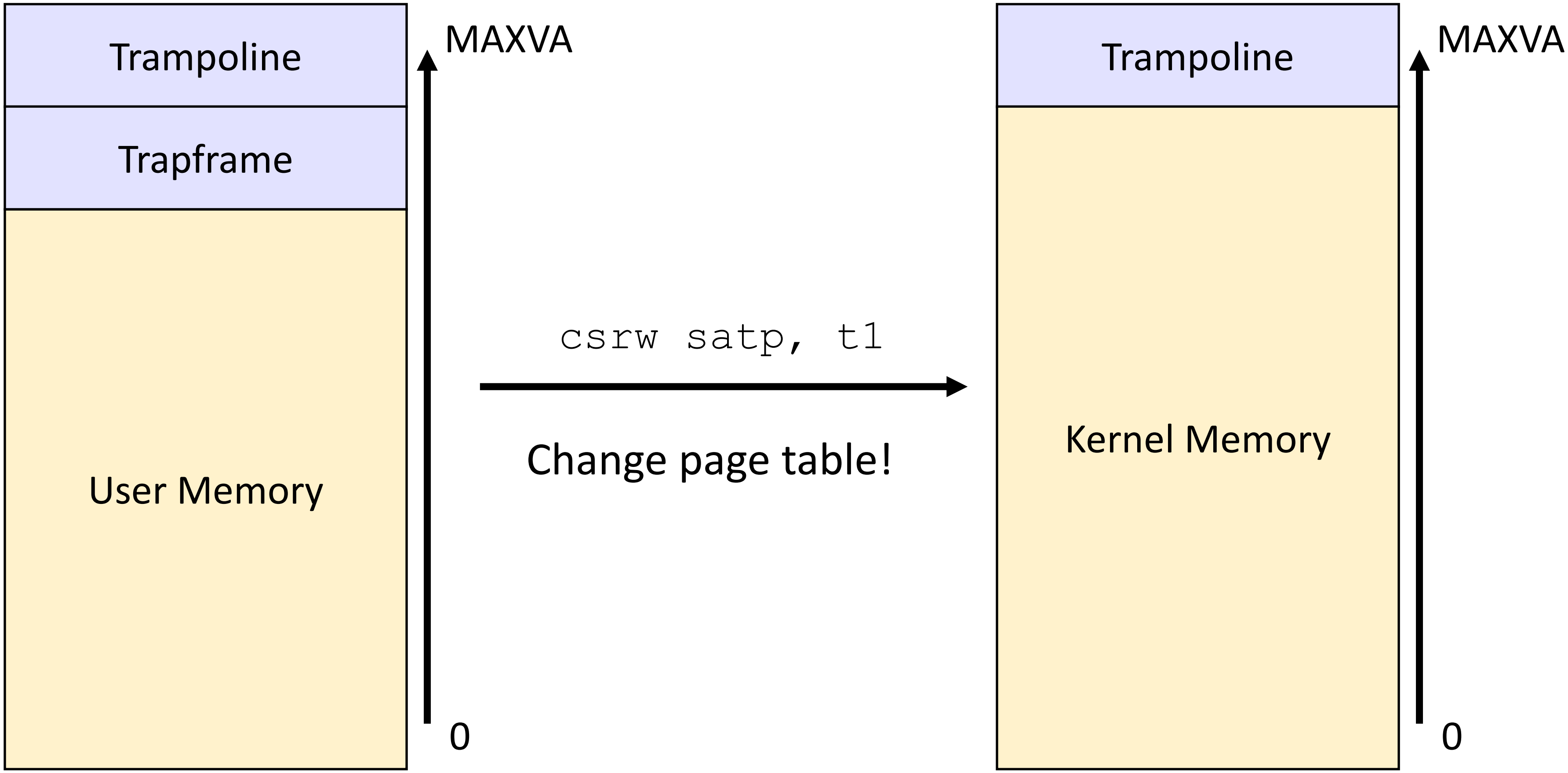
```
# save the user registers in TRAPFRAME
       sd ra, 40(a0)
       sd sp, 48(a0)
       sd gp, 56(a0)
       sd tp, 64(a0)
       sd t0, 72(a0)
       sd t1, 80(a0)
       sd t2, 88(a0)
       sd s0, 96(a0)
       sd s1, 104(a0)
       sd a1, 120(a0)
       sd a2, 128(a0)
       sd a3, 136(a0)
       sd a4, 144(a0)
       sd a5, 152(a0)
       sd a6, 160(a0)
       sd a7, 168(a0)
```

```
sd s2, 176(a0)
sd s3, 184(a0)
sd s4, 192(a0)
sd s5, 200(a0)
sd s6, 208(a0)
sd s7, 216(a0)
sd s8, 224(a0)
sd s9, 232(a0)
sd s10, 240(a0)
sd s11, 248(a0)
sd t3, 256(a0)
sd t4, 264(a0)
sd t5, 272(a0)
sd t6, 280(a0)
# save the user a0 in p->trapframe->a0
csrr t0, sscratch
sd t0, 112(a0)
```

Step 2: Trampoline (trampoline.S)

```
…
# restore kernel stack pointer from p->trapframe->kernel_sp
ld sp, 8(a0)
# make tp hold the current hartid, from p->trapframe->kernel_hartid
ld tp, 32(a0)
# load the address of usertrap(), p->trapframe->kernel_trap
ld t0, 16(a0)
# restore kernel page table from p->trapframe->kernel_satp
ld t1, 0(a0)
csrw satp, t1
sfence.vma zero, zero
# jump to usertrap(), which does not return
jr t0
```

Step 2: Trampoline



`csrw satp, t1`

Change page table!

Step 3: Usertrap

- Entry point into kernel C code
  - Handles system calls, traps, and interrupts
  - Must figure which it is to dispatch to right part of kernel
- r_scause() helps usertrap() figure this out
  - A value of 8 indicates system call

Step 4: System call

- Must determine which system call
- Kernel maintains table of function pointers
- A7 register value in TF determines index in table
- e.g., sys_write()

Summary

- System calls are much more complex than func calls
- Major reasons:
  - Requirement of isolation
  - Desire for simple, fast, and flexible hardware
- Questions to consider for entry/exit
  - Can an evil program abuse the entry mechanism?
  - Can you think of ways to make hardware or software simpler?
  - Can you think of ways to make traps faster?