Address Translation

How to Support Virtual Memory?

By Matthew Tancreti
For COM S 352
Iowa State University

Address Translation

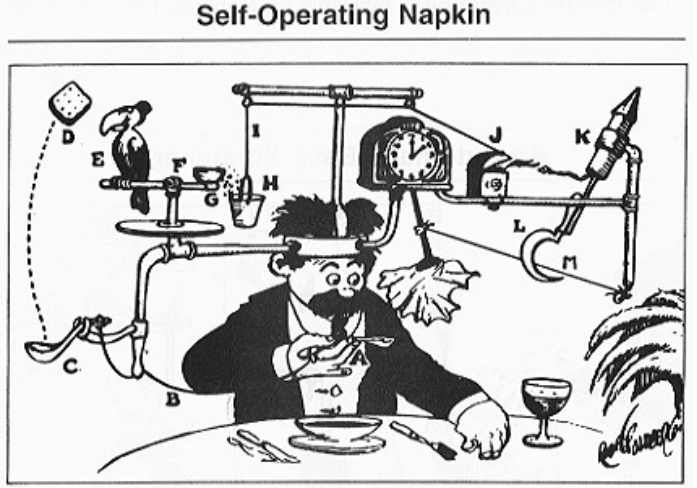
Memory virtualization – processes have abstract view of memory (their address space) but share single physical memory

Address translation – translate process address into physical address

To solve the problem of CPU virtualization we used **limited direct execution** – context switch to kernel mode, then returns to user mode

Address translation similar conceptually, but memory accesses happen every instruction, not realistic to trap on every access

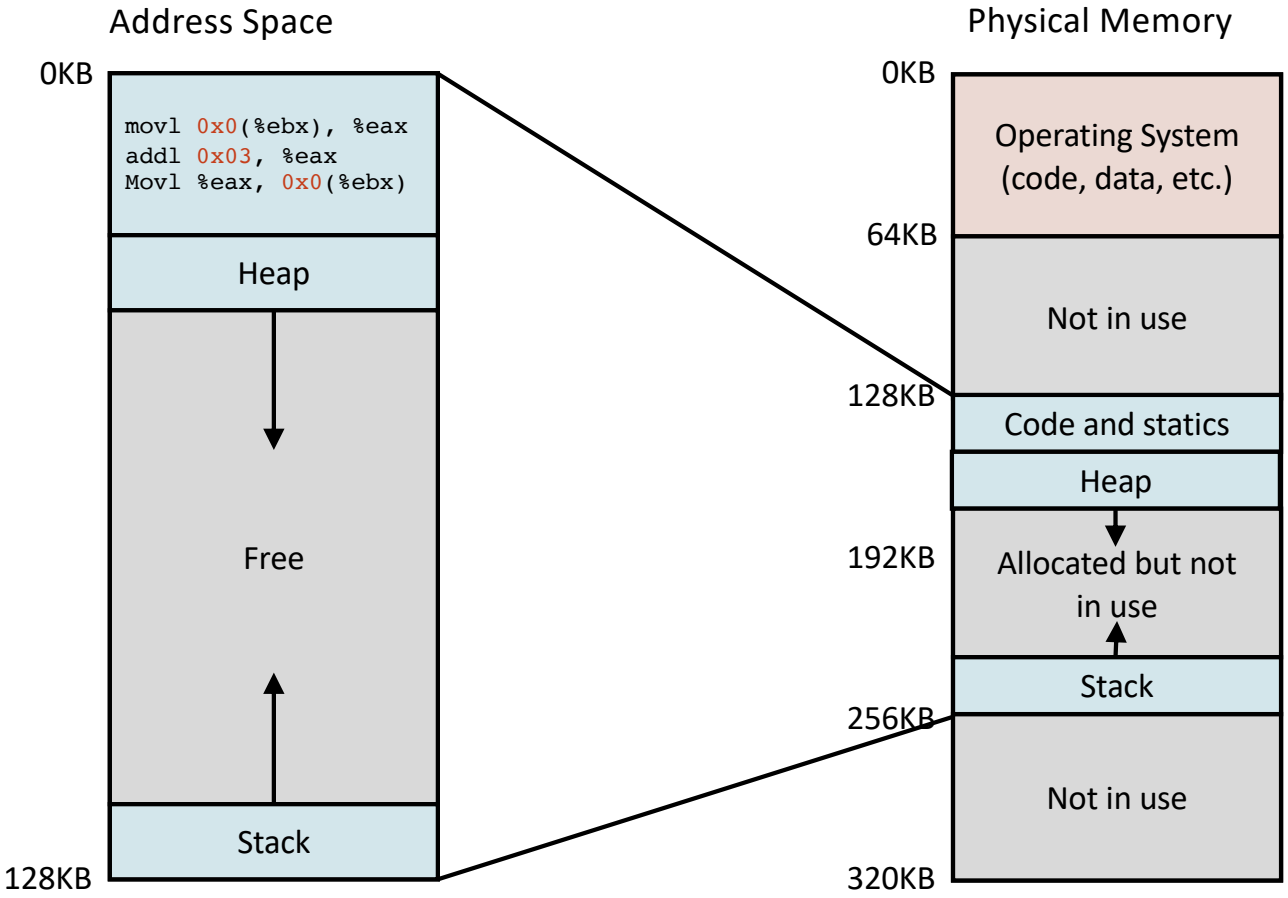
Efficient address translation requires additional hardware support – **hardware-based address translation**



Rube Goldberg's "Self Operating Napkin" [source]

*"All problems in computer science can be solved by another level of **indirection**."*
- Butler Lampson

Memory Relocation Example



Software-Based Translation Method

The purpose of a **loader** is to load the binary program on disk into the process memory

Some early loaders also had the job of translating all addresses found in instructions from virtual to physical locations

Translation is performed once (**statically**) before the process begins execution

Disadvantages: the loader needs to be trusted code (or no memory protection) and relocation after the process starts is costly

Base and Bounds Translation Method

The **base and bounds** method requires two CPU registers

base – points to start of process in physical memory

bounds – points to maximum legal address for process

When instruction is executed, all addresses translated by hardware

```
physical address = virtual address + base
```

If physical address > bounds, access is illegal, trap to kernel

Allows **dynamic relocation** or process memory

MMU

The hardware responsible is the **Memory Management Unit (MMU)**

It is typically part of the CPU but sits between the core and the address buss

Translates all addresses between CPU and main memory

Example

Instruction in code:

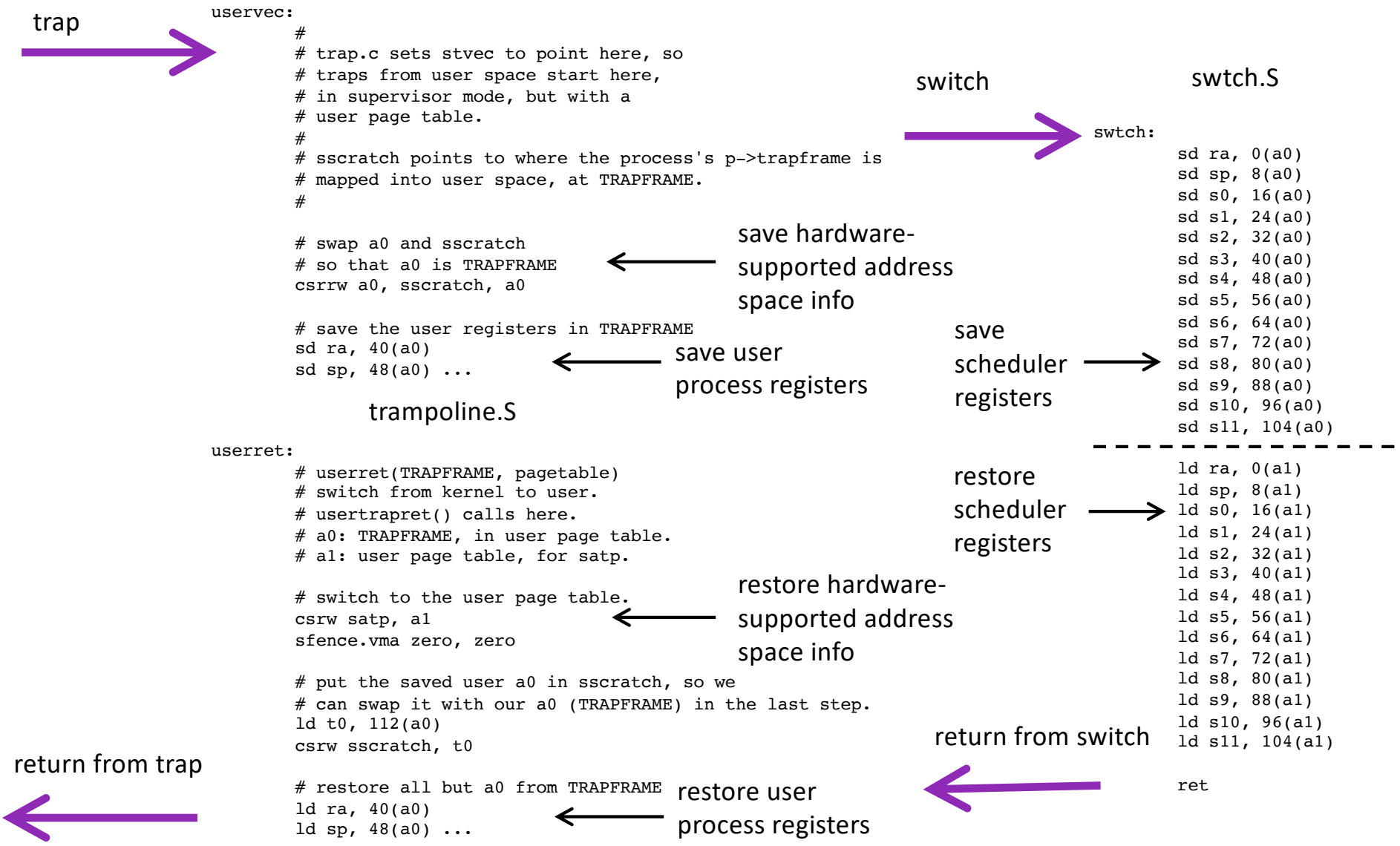
```
128: mov 1000, %eax
```

1. Program Counter (PC) is incremented to 128
2. CPU begins fetching instruction by reading from address 128
3. MMU translates 128 to 32,896 and memory is read
4. CPU decodes instruction and requests a read from address 1000
5. MMU translates 1000 to 33,768 and memory is read
6. CPU finishes execution of instruction

Example

OS @ boot (kernel mode)	Hardware	(No Program Yet)
initialize trap table		
	remember addresses of...	
	system call handler	
	timer handler	
	illegal mem-access handler	
	illegal instruction handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table		
initialize free list		

OS @ run (kernel mode)	Hardware	Program (user mode)
To start process A:		
allocate entry		
in process table		
alloc memory for process		
set base/bound registers		
return-from-trap (into A)		
	restore registers of A	
	move to **user mode**	
	jump to A's (initial) PC	
		Process A runs
		Fetch instruction
	translate virtual address	
	perform fetch	
		Execute instruction
	if explicit load/store:	
	ensure address is legal	
	translate virtual address	
	perform load/store	
		(A runs...)
	Timer interrupt	
	move to **kernel mode**	
	jump to handler	
Handle timer		
decide: stop A, run B		
call `switch()` routine		
save regs(A)		
to proc-struct(A)		
(including base/bounds)		
restore regs(B)		
from proc-struct(B)		
(including base/bounds)		
return-from-trap (into B)		
	restore registers of B	
	move to **user mode**	
	jump to B's PC	
		Process B runs
		Execute bad load
	Load is out-of-bounds;	
	move to **kernel mode**	
	jump to trap handler	
Handle the trap		
decide to kill process B		
deallocate B's memory		
free B's entry		
in process table		


trap
uservec:
#
trap.c sets stvec to point here, so
traps from user space start here,
in supervisor mode, but with a
user page table.
#
sscratch points to where the process's p->trapframe is
mapped into user space, at TRAPFRAME.
#
swap a0 and sscratch
so that a0 is TRAPFRAME
csrrw a0, sscratch, a0
save hardware-supported address space info
save the user registers in TRAPFRAME
sd ra, 40(a0)
sd sp, 48(a0) ...
save user process registers
trampoline.S
userret:
userret(TRAPFRAME, pagetable)
switch from kernel to user.
usertrapret() calls here.
a0: TRAPFRAME, in user page table.
a1: user page table, for satp.
switch to the user page table.
csrw satp, a1
sfence.vma zero, zero
restore hardware-supported address space info
# put the saved user a0 in sscratch, so we
# can swap it with our a0 (TRAPFRAME) in the last step.
ld t0, 112(a0)
csrw sscratch, t0
# restore all but a0 from TRAPFRAME
ld ra, 40(a0)
ld sp, 48(a0) ...
restore user process registers
return from trap
switch
swtch.S
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)
save scheduler registers
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)
restore scheduler registers
ret
return from switch

Exception Handling

Memory access out of bounds results in a trap

OS typically terminates process

Hardware Requirements

Hardware Requirement	Common Implementation
Privilege mode	Kernel mode
Base and bounds registers	
Translate virtual address	**MMU** intercepts all addresses between CPU and bus
Privileged instructions to update base and bounds	Write to registers (kernel mode)
Privileged instructions to register exception handlers	Write to interrupt vector table (kernel mode)
Ability to raise exceptions	**Trap** when read out of bounds