II. Memory Isolation

Step 2: Address Translation

Implement a function mapping \( \langle \text{pid, virtual address} \rangle \) into physical address.

Enables:
- Isolation
- Relocation
- Data sharing
- Multiplexing
- Non-contiguity

Virtual

\[ \pi_i \]

0xA486D4

Physical

0x5E3A07
Isolation

At all times, functions used by different processes map to disjoint ranges — aka “Stay in your room!”
Relocation

The range of the function used by a process can change over time
Relocation

The range of the function used by a process can change over time — “Move to a new room!”
Map different virtual addresses of distinct processes to the same physical address — (“Share the kitchen”)

$p_i$

0x4D26A

$p_j$

0x19AF3

0x5E3A07
Data Sharing

Map different virtual addresses of distinct processes to the same physical address — ("Share the kitchen")
Multiplexing

Create illusion of almost infinite memory by changing domain (set of virtual addresses) that maps to a given range of physical addresses — ever lived in a studio?
Multiplexing

The domain (set of virtual addresses) that map to a given range of physical addresses can change over time.
The domain (set of virtual addresses) that map to a given range of physical addresses can change over time.
The domain (set of virtual addresses) that map to a given range of physical addresses can change over time.
Multiplexing

The domain (set of virtual addresses) that map to a given range of physical addresses can change over time.
More Multiplexing

At different times, different processes can map part of their virtual address space into the same physical memory — (change tenants)
More Multiplexing

At different times, different processes can map part of their virtual address space into the same physical memory — (change tenants)
(Non) Contiguity

Contiguous virtual addresses can be mapped to non-contiguous physical addresses...
(Non) Contiguity

...and non-contiguous virtual addresses can be mapped to contiguous physical addresses
A simple mapping mechanism: 
Base & Bound

Hardware to the rescue!
A simple mapping mechanism: Base & Bound

CPU

Logical addresses

Memory Exception

≤

yes

no

≤

500

Bound Register

1000

Base Register

Physical addresses

p’s physical address space

MAX_{sys}

0

1000

1500
On Base & Limit

- **Contiguous Allocation:** contiguous virtual addresses are mapped to contiguous physical addresses

- Isolation is easy, but sharing is hard
  - *Say* I have many copies of Safari open...
    - I may want them to share the same code, or even the same global variables

- And there is more...
  - *Hard to relocate*
    - Addresses are absolute and may be stored in registers or on the stack (a return address)
Giving control back to the Hardware to the rescue!
III. Timer Interrupts

- Hardware timer
  - can be set to expire after specified delay (time or instructions)
  - when it does, control is passed back to the kernel

- Other interrupts (e.g. I/O completion) also give control to kernel
Interrupt Management

Interrupt controllers implements interrupt priorities:

- Interrupts include descriptor of interrupting device
- Priority selector circuit examines all interrupting devices, reports highest level to the CPU
- Controller can also buffer interrupts coming from different devices
Interrupt Management

**Maskable** interrupts

- can be turned off by the CPU for critical processing

**Nonmaskable** interrupts

- indicate serious errors (power out warning, unrecoverable memory error, etc.)
Types of Interrupts

Exceptions
- process missteps (e.g. division by zero)
- attempt to perform a privileged instruction
  - sometime on purpose! (breakpoints)
- synchronous/non-maskable

Interrupts
- HW device requires OS service
  - timer, I/O device, interprocessor
- asynchronous/maskable

System calls/traps
- user program requests OS service
- synchronous/non-maskable
Interrupt Handling

Two objectives

- handle the interrupt and remove the cause
- restore what was running before the interrupt
  - kernel may modify saved state on purpose

Two “actors” in handling the interrupt

- the hardware goes first
- the kernel code takes control by running the interrupt handler
Review: stack (aka call stack)

```c
int main(argc, argv){
    ...
    f(3.14)
    ...
}

int f(x){
    ...
    g();
    ...
}

int g(y){
    ...
}
```

- Arguments (3.14)
- Return address
- Saved FP (main)
- Local variables
- Saved registers
- Scratch space
- User stack
- PC/IP
- SP
- FP

Stack frames for:
- `main()`
- `f()`
- `g()`
A Tale of Two Stack Pointers

(it was the best of stacks...)

- Interrupt handler is a program: it needs a stack!
  - so, each process has two stacks pointers:
    - one when running in user mode
    - a second one when running in kernel mode, to support interrupt handlers

- Why not use the user-level stack?
  - user SP cannot be trusted to be valid or usable
  - user stack may not be large enough, and may spill to overwrite important data
  - security:
    - e.g., kernel could leave sensitive data on stack
Handling Interrupts: HW

- On interrupt, hardware:
  - sets supervisor mode (if not set already)
  - disable *(masks)* interrupts *(partially privileged)*
  - pushes PC, SP, and PSW of user program on interrupt stack
  - sets PC to point to the first instruction of the appropriate interrupt handler
    - depends on interrupt type
    - interrupt handler specified in *interrupt vector* loaded at boot time
Handling Interrupts: SW

- We are now running the interrupt handler!
  - IH first pushes the registers’ contents (needed to run the user process) on the interrupt stack
    - need registers to run the IH
    - only saves necessary registers (that’s why done in SW, not HW)