More Multiplexing

At different times, different processes can map part of their virtual address space into the same physical memory.

A simple mapping mechanism: Base & Bound

Contiguous Allocation: contiguous virtual addresses are mapped to contiguous physical addresses.

Isolation is easy, but sharing is hard.

And there is more...

- Hard to relocate
- We want heap and stack as far as as possible in virtual address space, but...
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 implement interrupt priorities:
  - Interrupts include descriptor of interrupting device
  - Priority selector circuit examines all interrupting devices, reports highest level to the CPU
  - More on this later...

Interrupt-driven I/O

- **Memory-mapped I/O**
  - Device communicate over the memory bus
  - I/O ops by dedicated device hardware correspond to reads/writes to special addresses
  - Devices appear as if part of the memory address space

**Maskable interrupts**
- Can be turned off by the CPU for critical processing

**Nonmaskable interrupts**
- Indicate serious errors (power out warning, unrecoverable memory error, etc.)
Interrupt-driven I/O

- **Memory-mapped I/O**
  - Device communicate over the memory bus
  - I/O ops by dedicated device hardware correspond to reads/writes to special addresses
  - Devices appear as if part of the memory address space

- **Interrupt driven ops with memory-mapped I/O**
  - CPU initiates device op (e.g., disk read): writes op descriptor to designated memory location
  - CPU continues its regular computation
  - Device asynchronously performs op; when op completes, interrupts the CPU
  - Could happen for each byte read!

From interrupt-driven I/O to DMA

- **Interrupt driven I/O**
  - Device ↔ CPU ↔ RAM
    - for \( (i = 1 \ldots n) \)
      - CPU issues read request
      - device interrupts CPU with data
      - CPU writes data to memory

- **Direct Memory Access**
  - Device ↔ RAM
    - for \( (i = 1 \ldots n) \)
      - CPU sets up DMA request
      - Device puts data on bus & RAM accepts it
      - Device interrupts CPU when done

From user mode to kernel mode...

- **Exceptions**
  - user program acts silly (e.g., division by zero)
  - attempt to perform a privileged instruction
    - sometime on purpose! (breakpoints)
  - synchronous

- **System calls/traps**
  - user program requests OS service
  - synchronous

- **Interruptions**
  - HW device requires OS service
    - timer, I/O device, interprocessor
  - asynchronous
...and viceversa

Resume process $p$ after exception, interrupt or syscall
- restore PC, SP, registers;
- toggle mode

Switch to different process $q$
- load PC, SP, registers from $q$'s PCB
- toggles mode

If new process
- copy program in memory,
- set PC and SP
- toggle mode

User-level upcall
- a sort of user-level interrupt handling

Making the transition: Safe mode switch

- Common sequences of instructions to cross boundary, which provide:
  - Limited entry
    - entry point in the kernel set up by kernel
  - Atomic changes to process state
    - PC, SP, memory isolation, mode
  - Transparent restartable execution
    - user program must be restarted exactly as it was before kernel got control

Interrupt vector

- Hardware identifies why boundary is crossed
  - trap?
  - interrupt (which device)?
  - exception?
- Hardware selects entry from interrupt vector
- Appropriate handler is invoked

Interrupt stack

- Pointed by privileged register
- Stores execution context of interrupted process
  - HW saves SP, PC
  - Handler saves remaining registers
- Stores handler's local variables
- One interrupt stack per process (or per thread!)
- Why not use the stack in user's space?
  - Reliability: user-level stack pointer may not be valid
  - Security: on a multiprocessor, another thread could modify return address for kernel, jumping to arbitrary code
Interrupt masking

- What if an interrupt occurs while running an interrupt handler?
  - Disable interrupts via privileged instruction
    - Overdramatic… it actually defers them
  - Just use the current SP of Interrupt stack

Mode switch on x86

1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack

Hardware performs these steps
Mode switch on x86

User-level Process

Registers

Kernel

Code

foo() {
    while(...) {
        x = x+1;
        y = y-2
    }
}

Stack

Hardware performs these steps:
1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack

EFLAGS

SS:ESP

CS:EIP

Other Registers:
EAX, EBX,
...

Interrupt Stack

Hardware performs these steps:
1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack
6. Save error code (optional)

EFLAGS

SS:ESP

CS:EIP

Error

Software handler performs this step
7. Handler pushes all registers on stack
1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack
6. Save error code (optional)
7. Transfer control to interrupt handler

Hardware performs these steps

Software (handler) performs this step
8. Handler pushes all registers on stack

Switching back

- From an interrupt, handler reverses all steps!
  - pops registers saved on the stack
  - executes instruction that restore PC, SP, and EFLAGS

- From exception and system call, increment PC on return
  - on exception, handler changes PC at the base of the stack
  - on system call, increment is done by hw when saving user level state