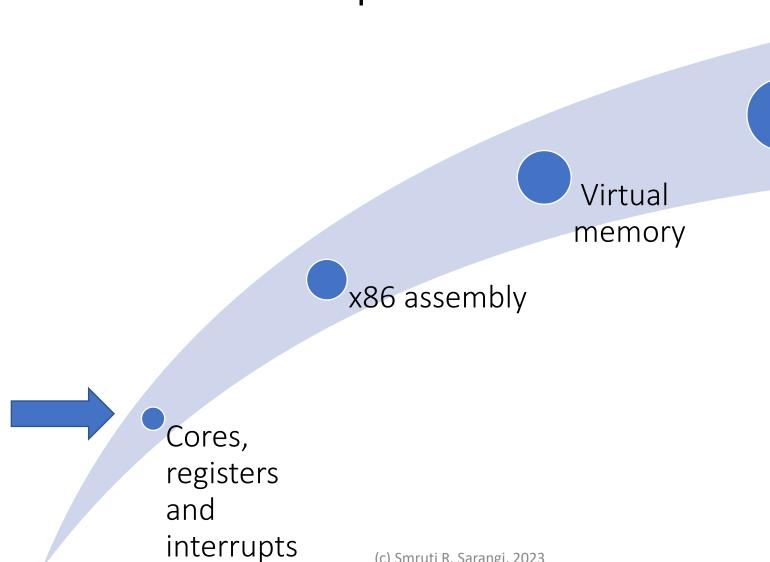
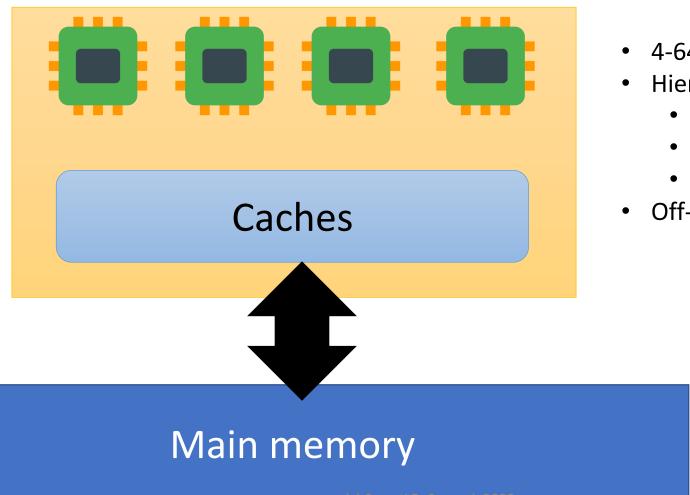


Outline of this Chapter



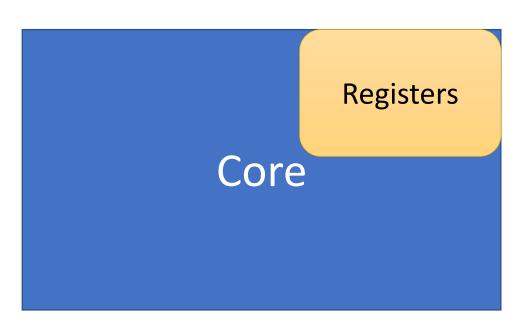
1/0

Design of a Modern Multicore Chip



- 4-64 cores
- Hierarchy of caches
 - Ist level: i-cache and d-cache
 - IInd level: L2 cache (several MBs)
 - IIIrd level: L3 cache (MBs)
- Off-chip main memory

Cores and Registers



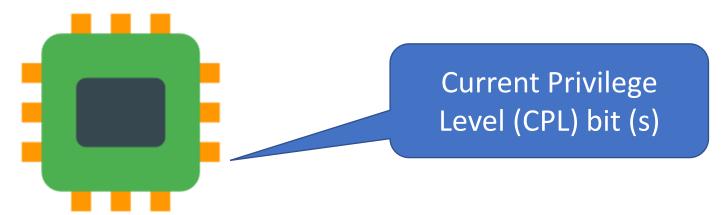
- Every core has a set of registers named storage locations
- They number from 8-32. They can be accessed very quickly (fraction of a clock cycle).
- Most of the CPU's operations are performed usually on registers.
- RISC machines first load memory values into registers and then perform operations on them
- CISC machines can have one source operand from memory (lower register usage)

What does the register set of a typical machine look like?

- General purpose registers: These registers can be used by all programs
- The rest are all privileged registers. They can only be used in certain processor modes (e.g., by the OS, hypervisor, etc.)
 - Hidden registers: Example \rightarrow flags register. This stores the result of the last comparison.
 - Control registers: Use them to enable/disable certain processor features
 - Debug registers: Debug hardware and system software
 - I/O registers

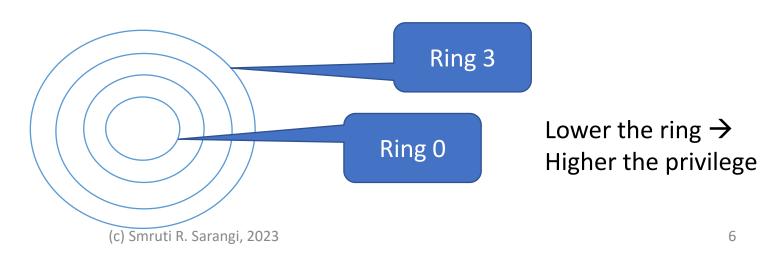
Current Privilege Level

Current Privilege Level



In general CPL=0 indicates the privileged OS mode and CPL=1 indicates user mode

Extended to Rings



Privileged and Non-Privileged Instructions

- There are two kinds of instructions privileged and non-privileged
- Non-privileged instructions
 - Regular instructions
 - When they access privileged registers, several things can happen
 - An exception may be generated
 - There will be some effect
 - There will be no effect at all (fully silent)
- Privileged instructions
 - Can access all registers
 - Exceptions are not generated

Interrupts, Exceptions, and System Calls

Interrupts

• An interrupt is an externally generated event whose main job is to draw the attention of the CPU. They are mainly generated by I/O devices and other chips in the chipset.

Exceptions

 An exceptional condition in a program such as accessing an illegal memory address or dividing by zero. There is a need to suspend the program and take some action to rectify the situation.

System calls

There are specialized instructions in ISAs to generate a "dummy exception". They lead to a suspension of the program's execution and invocation of an OS routine. This mechanism can be used to pass data to the OS such that it can perform a service for the user program. Such a convoluted OS function call mechanism is known as a system call sarangi, 2023

System calls in x86-64 Linux

• See /usr/include/asm/unistd_64. h (286 systems calls defined)

mov \$<sys call number>, %rax
syscall

- Move the system call number to the rax register
- Issue the "syscall" instruction
- Another option: int \$0x80 (software interrupt)

All are handled in the same manner

The idea is to knock on the doors of the operating system to draw its attention.

- 1. Either rely on a hardware interrupt that the CPU uses as a pretext to invoke an OS routine
- 2. OR, treat a fault/exception in the program as an interrupt
- 3. OR, generate a software interrupt yourself to ask the OS to do some work for you



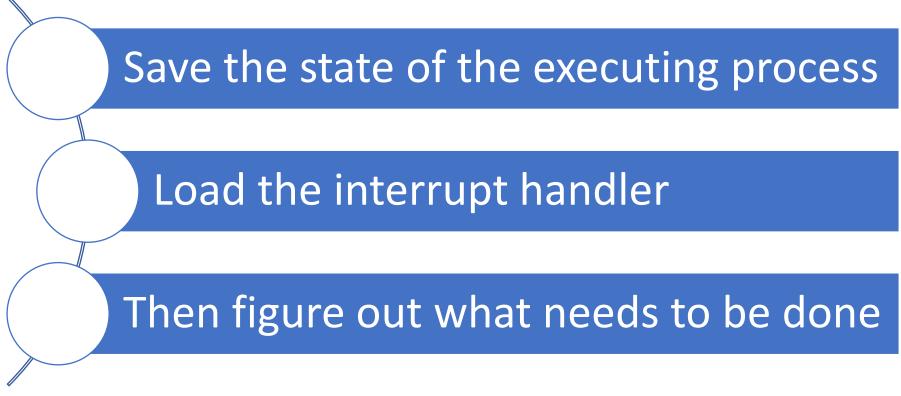


Kind of a weird way of drawing the OS's attention. The only way to talk to the fireman is by setting your house on fire.

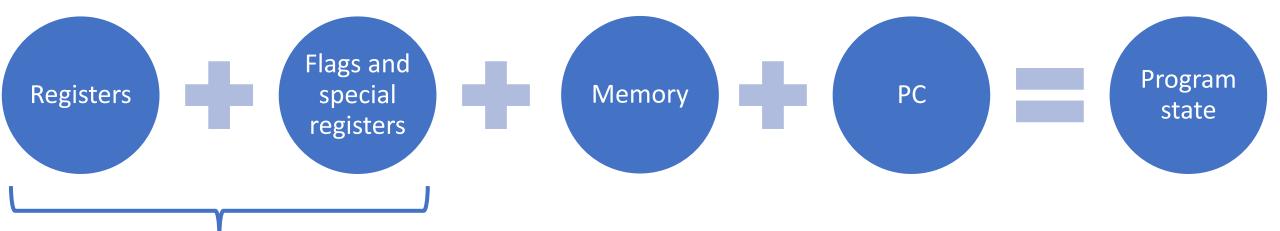
(c) Smruti R. Sarangi, 2023

Now what does the CPU do?

• Something has happened, the OS needs to take a look



Saving the State: Context Switch

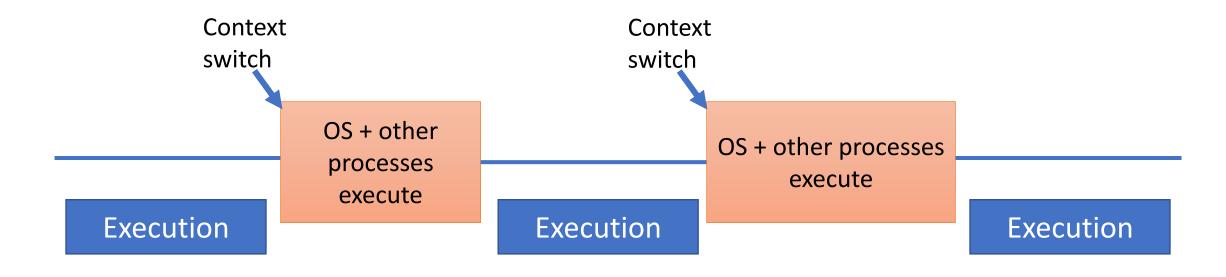


Register state

- Store the register state somewhere
- The memory of the process should remain untouched
- Store the PC of the last executed instruction
- Then do other work
- Later on, restart the process from exactly the same point



Lifecycle of a Process

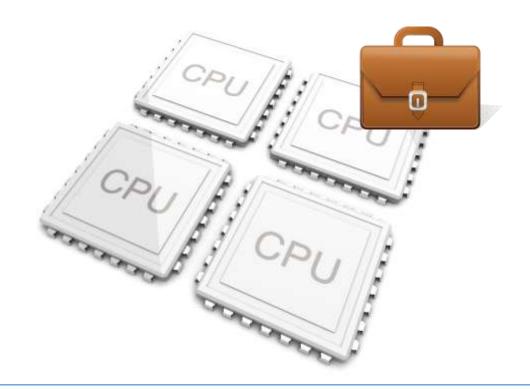




The process has no way of knowing that it is being swapped out and being swapped in

It is agnostic to context switches

How does the OS see a process?





The OS treats the process as a suitcase containing some state. It can be seamlessly moved from core to core on a multi-core CPU. It can be suspended at will and brought back to life at a future point of time without the process knowing.

A Question to Ponder About?

?

What if there are no interrupts, exceptions, and system calls?



In principle, an application can continue to run forever.

It will never get swapped out and will continue to monopolize the processor.





Have an external time chip on the motherboard. Generates a dummy interrupt (timer interrupt) once every *jiffy*



A jiffy = 1 ms (as of today)

Guaranteed source of interrupts

Why do you need a guaranteed source of interrupts?

- This is to allow the OS to periodically execute and make process scheduling decisions.
- Otherwise, the OS may never get a chance to execute. For it to execute, it needs to be invoked by any one of the three mechanisms: interrupts, exceptions or system calls.
- The latter two are not relevant here. We need to thus generate timer interrupts.

Relevant Kernel Code

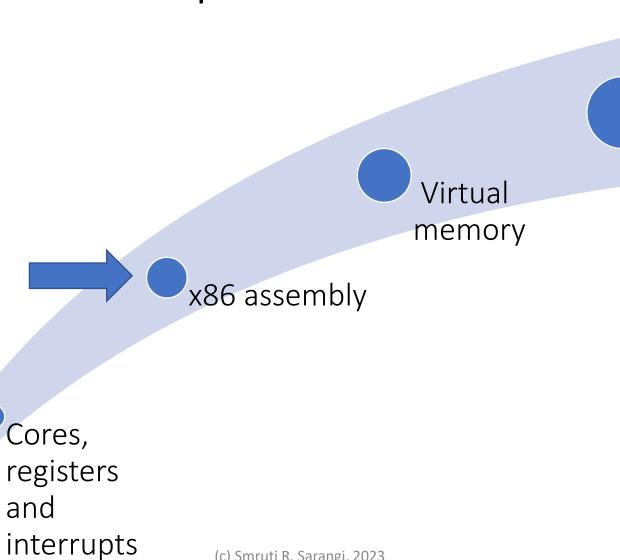
https://elixir.bootlin.com/linux/v6.1.2/source/include/linux/jiffies.h



extern unsigned long volatile jiffies;

The jiffy count is incremented once every time there is a timer interrupt. The interval is determined by the compile-time parameter HZ. If HZ = 1000, then it means that the timer interrupt interval = 1 ms

Outline of this Chapter

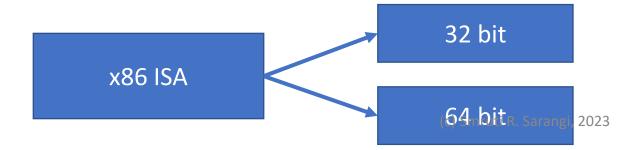


1/0

Introduction to Assembly Languages

A large part of the kernel code is written in assembly language

- ?
 - We need low-level control of the hardware.
 - Speed and efficiency
 - Limited code size
 - Specify the exact code that will be executed



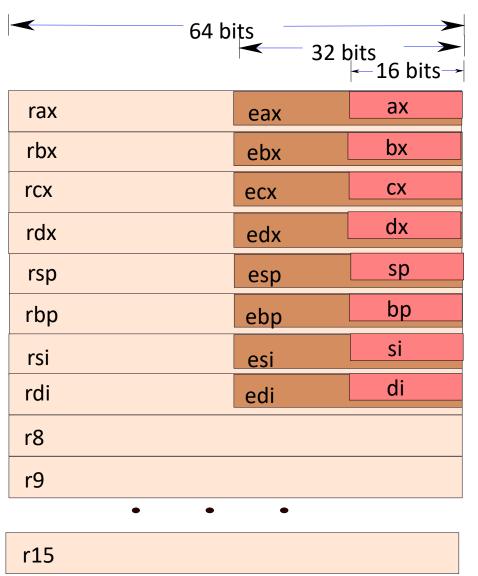
View of Registers

- * Modern Intel machines are still ISA compatible with the arcane 16-bit 8086 processor
- * In fact, due to market requirements, a 64-bit processor needs to be ISA compatible with all 32-bit, and 16-bit ISAs
- * What do we do with registers?
- * Do we define a new set of registers for each type of x86 ISA? **ANSWER**: **NO**

View of Registers – II

- * Consider the 16-bit x86 ISA It has 8 registers: ax, bx, cx, dx, sp, bp, si, di
- * Should we keep the old registers, or create a new set of registers in a 32-bit processor?
- * NO Widen the 16-bit registers to 32 bits.
- * If the processor is running a 16-bit program, then it uses the lower 16 bits of every 32-bit register.

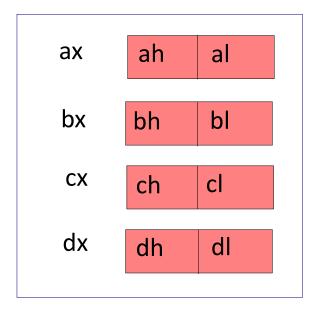
View of Registers – III



8 registers 64, 32, 16 bit variants

The 64-bit ISA has 8 extra registers r8 - r15

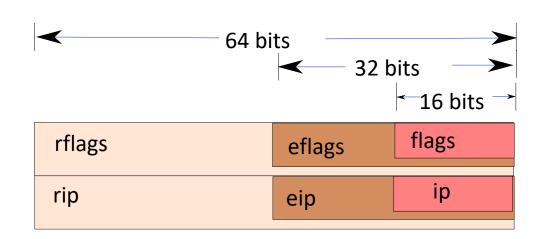
x86 can even Support 8-bit Registers



* For the first four 16-bit registers

- * The lower 8 bits are represented by al, bl, cl, dl
- * The upper 8 bits are represented by ah, bh, ch, dh

x86 Flags Registers and PC



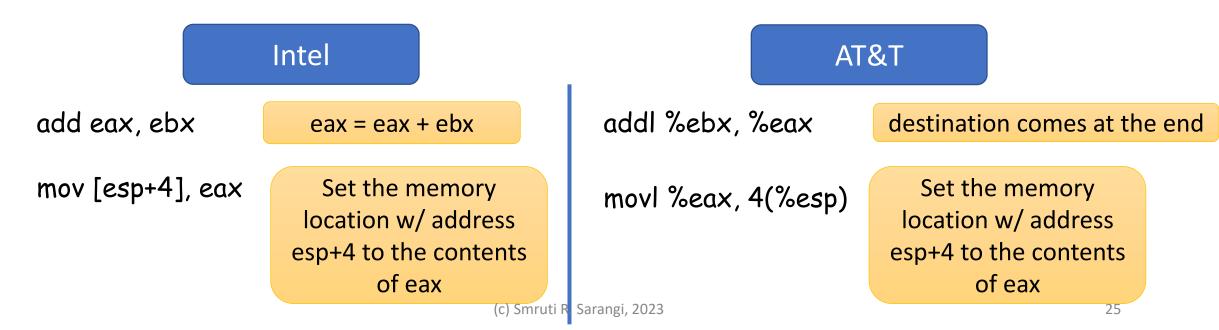
Fields in the flags register

| Field | Condition | Semantics |
|-------|------------|------------------------------------|
| OF | Overflow | Set on an overflow |
| CF | Carry flag | Set on a carry or borrow |
| ZF | Zero flag | Set when the result is a 0, or the |
| | | comparison leads to an equality |
| SF | Sign flag | Sign bit of the result |

- * Similar to the classical flags registers in RISC ISAs
- * It has 16-bit, 32-bit, and 64-bit variants
- * The PC is known as the IP (instruction pointer)

Intel and AT&T Formats

- The same code can be written in two different formats: Intel and AT&T
- The Linux kernel and the toolchain uses the AT&T format (which is older and often not preferred by modern developers)



Floating-point Registers

FP register stack



- * x86 has 8 (80 bit) floating-point registers
 - * st0 st7
 - * They are also arranged as a stack
 - * st0 is the top of the stack
 - * We can perform both register operations, as well as stack operations

Memory Addressing Mode

address = disp(base, index, scale)

address = base + index*scale + disp



- * x86 supports a base, a scaled index and an offset (known as the displacement)
- * Each of the fields is optional

Examples

-32(%eax, %ebx, 0x4)

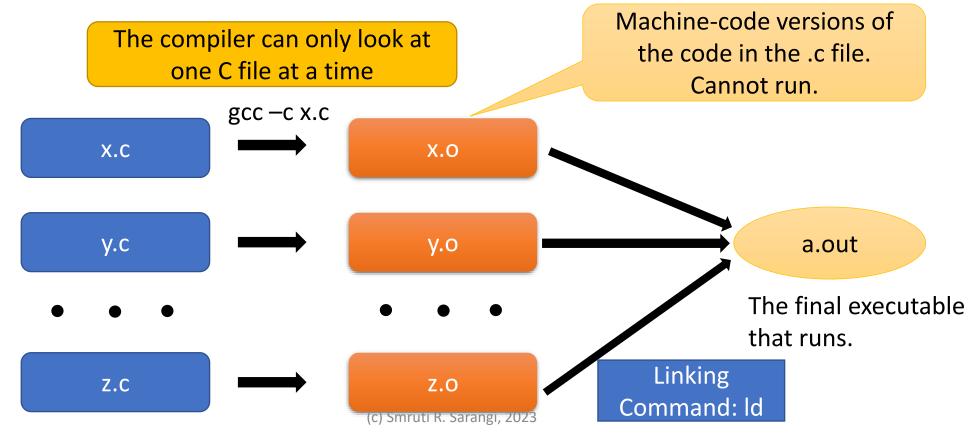
(%eax, %ebx)

Assembly code for computing a factorial

```
movl
              $1, %edx
    mov1
             $1, %eax
.L2:
    imull
             %eax, %edx
                               edx = edx * eax
    addl
             $1, %eax
                                  eax++
    cmpl $11, %eax
                                Exit condition
    jne .L2
```

Brief Detour: Compiling and Linking

- Any large project comprises 100s of C/C++ files.
- How do we create a single executable out of them?



What does a .o file contain?

- Sections (objdump -x <file.o>)
 - .text (the machine instructions for the C code)
 - .data (initialized data)
 - .bss (uninitialized data)
 - .rodata (read-only data)
 - .comment
- The symbol table (objdump --syms <file.o>)
 - The list of all the symbols that are used (both defined and undefined)
- The relocation table (objdump --reloc <file.o>)
 - The symbols that need to be resolved at link time

How are developers supposed to collaborate?

Defines the factorial function

factorial.c

Uses the factorial function defined in factorial.c

prog.c

How will this happen?

- Compilers take a look at each C file individually
- When the compiler is trying to convert prog.c to prog.o, it needs some information about the factorial function.



What is the signature of the factorial function?

Answer: int factorial (int), Arguments and return value type.

A few follow-up questions

Where does prog.c get the signature of the factorial function from?

Bad way

Define it in prog.c before the function is used. extern int factorial (int)



We cannot change the signature later. This will clutter the C code. If many C files are using this function, then there is a lot of code replication.

Good way

Define the signature in a header file: factorial.h

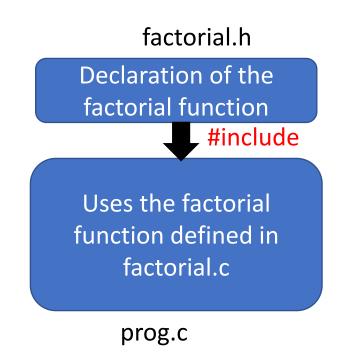
- Just include factorial.h to get the signature: #include <factorial.h>
- The pre-processor simply copy-pastes the contents of the header file to the relevant place in the C file.
- Great idea for quickly exporting signatures to any C file that is interesting in using functions defined in another C file.

New Structure

Symbol table

Defines the factorial function

factorial.c



- What is the address of the factorial function in prog.o?
 - It is unresolved.
 - There is an entry in the relocation table (in the .o file) that says that the symbol "factorial" is undefined.
- At link time, the symbol is substituted with its real address
 - The call instruction that calls factorial finally points to the correct address address of the first byte of the factorial function in memory

Static Linking

 Along with functions defined in other C files, a typical executable calls a lot of functions that are defined in the standard C libraries (essentially large .o files). Examples: printf, scanf, time, etc.

• Do we bundle all of them together?

Check if all the functions are bundled or not The size of the binary is quite large because the code of the entire library is included in a.out Add the code to a.out gcc -static test.c Idd a.out not a dynamic executable du -h a.out 892K a.out (c) Smruti R. Sarangi, 2023

```
#include <stdio.h>

int main(){
   int a = 4;
   printf ("%d",a);
}
```

Why is a out so large?

- This is because the code of all possible functions that can be invoked by a.out is added to it
- Imagine a program can possibly invoke 100 functions. However, in a realistic run, only 3 functions are invoked.
- There is no need to add the code of all the 100 functions to a.out.
- Add the code of functions to the process's address space on an on-demand basis
 - gcc test.c

Dynamic Linking

➤ Idd a.out a.out just has

pointers to

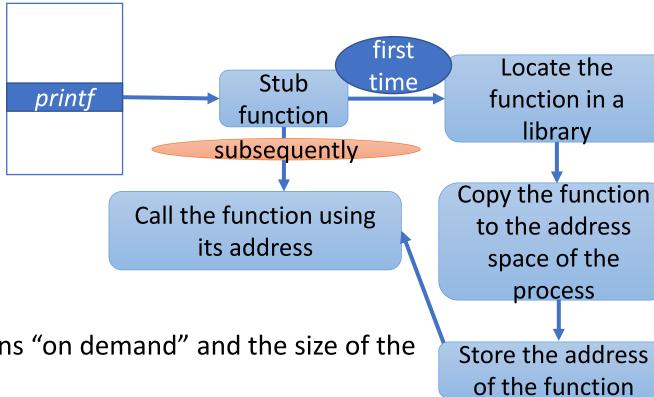
functions

 $linux-vdso.so.1 \Rightarrow (0x00007ffc51fd3000)$ libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f984d6a7000) /lib64/ld-linux-x86-64.so.2 (0x00007f984da71000)

du –h a.out 12K a.out

Ultrasmall

All about Dynamic Linking



- This means that we load functions "on demand" and the size of the binary remains small
- When we study virtual memory, we shall realize that copying the full code of the function is not necessary. A simple mapping can achieve the same. This means that only a single copy of *printf* needs to be resident in memory.

Example of Creating and Using a Static Library

Three files

factorial.c

```
#include "factorial.h"

int factorial (int val){
   int i, prod = 1;
   for (i=1; i<= val; i++) prod *= i;
   return prod;
}</pre>
```

prog.c

```
#include <stdio.h>
#include "factorial.h"

int main(){
   printf("%d\n", factorial(3));
}
```

factorial.h

```
#ifndef FACTORIAL_H
#define FACTORIAL_H
    extern int factorial(int);
#endif
```

Default

gcc factorial.c prog.c

> ./a.out

inefficient

Plain, old, simple, and

- gcc –c factorial.c –o factorial.o
- > gcc -c prog.c -o prog.o
- gcc factorial.o prog.o
- ./a.out 6

Compile .o files separately

In a large project we maintain a list of compiled .o files and compile as few files as possible when there is a new change. A Makefile and the make command automate this process. All that the programmer needs to type is make. The rules are in the Makefile.

Static and Dynamic Linking

- ➢ gcc ─c factorial.c ─o factorial.o
- > ar -crs factorial.a factorial.o
- gcc prog.o factorial.a
- > ./a.out

The *ar* command creates a library out of several .o files

factorial.a is a library that is statically linked in this case

- ➢ gcc −c −fpic −o factorial.o factorial.c
- gcc –shared –o libfactorial.so factorial.o
- gcc –L. prog.c -lfactorial
- export LD_LIBRARY_PATH=`pwd`
- > ./a.out

6

Generate position independent code

Create the shared library

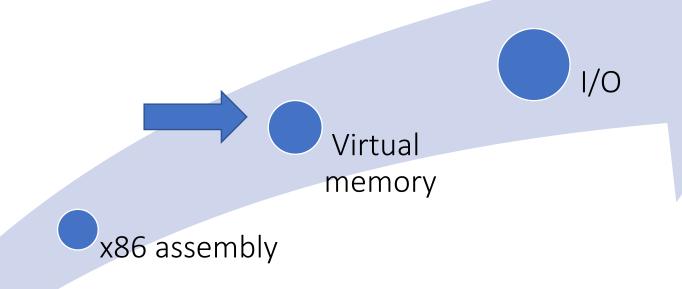
Create the executable. Reference the shared library.

Tell the system that the factorial library is in the current directory

Some Important Linux Commands

- Idd → Display the location of the shared libraries on your file system
- objdump → See all the contents of an object (.o) file
 Table, sections, machine instructions (also in the disassembled form)
- readelf → More expressive than objdump
- nm <file.o> \rightarrow List the symbols in object files
- *strip* <file.o> → Discard symbols from object files
- ranlib <archive> > Generate an index for an archive

Outline of this Chapter



Cores, registers and interrupts

Size and Overlap Problems



How does a process view memory?

Answer: One large array of bytes.





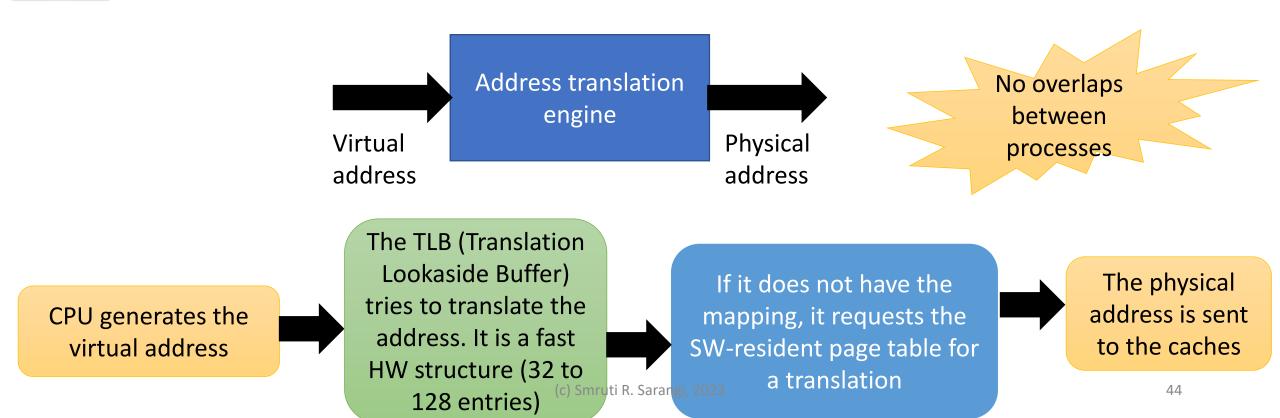
How do we ensure that the memory regions of two processes do not overlap?

Virtualize the Memory Divide memory into 4-KB chunks Map process's virtual Process 1 pages to physical frames Process 2

What is the key idea?



The program, compiler, and the runtime all see virtual addresses. This means that they assume that the process has one large, contiguous address space. The process can access any address at will.



What about the "size" problem?

Assume that a process wishes to access 3 GB of memory, but we only have 2 GB of main memory.

Answer:

Store 2GB in main memory and store the remaining 1GB on the hard disk or any other storage device. Let us refer to the "non-main memory" region as the swap space.

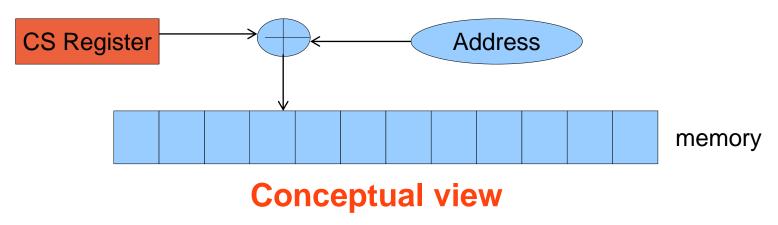
Use the virtual memory mechanism to manage the memory map.

If there is a TLB miss and the page table indicates that the frame is in the swap space, then bring it into the main memory first. The swap space could be on the local hard disk or could be on the hard disk of a remote machine.

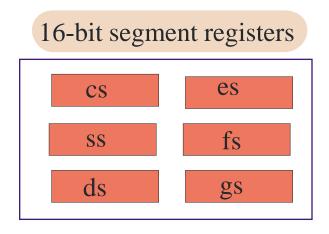
There could be several swap spaces as well.

View of Memory

- x86 follows a segmented memory model
 - Each address in x86 is actually an offset from the start of the segment.
 - For example, an instruction address is an offset in the code segment
 - The starting address of the code segment is maintained in a code segment (CS) register



Segmentation in x86



These registers are private to a CPU

- x86 has 6 different segment registers
 - Each register is 16 bits wide
 - Code segment (cs), data segment (ds), stack segment (ss), extra segment (es), extra segment 1 (fs), extra segment 2 (gs)
 - Depending upon the type of access, the CPU uses the appropriate segment register

Segmented vs Linear Memory Model

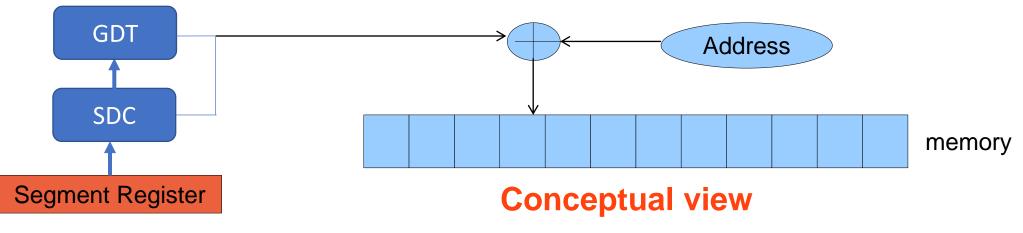
- In a linear memory model (e.g. RISC-V, ARM) the address specified in the instruction is sent to the memory system
 - There are no segment registers
- What are the advantages of a segmented memory model?
 - The contents of the segment registers can be changed by the operating system at runtime.
 - Can map the text section(code) to a dedicated part of memory, or in principle to other devices also (needed for security)
 - Stores cannot modify the instructions in the text section. REASON: Stores use
 the data segment, and instructions use the code segment

How does Segmentation Work

- * The segment registers nowadays contain an offset into a segment descriptor table
 - * Because 16 bits are not sufficient to store a memory address
- * Modern x86 processors have two kinds of segment descriptor tables
 - * LDT (Local Descriptor Table), 1 per process, typically not used nowadays
 - GDT (Global Descriptor Table), contains 8191 entries
 - * Each entry in these tables contains the starting address of the segment

Segment Descriptor Cache (similar to a TLB)

- Every memory access needs to access the GDT or LDT: VERY SLOW
- Use a segment descriptor cache (SDC) at each processor that stores a copy of the relevant entries in the GDT
 - Lookup the SDC first
 - If an entry is not there, send a request to the GDT
 - Quick, fast, and efficient



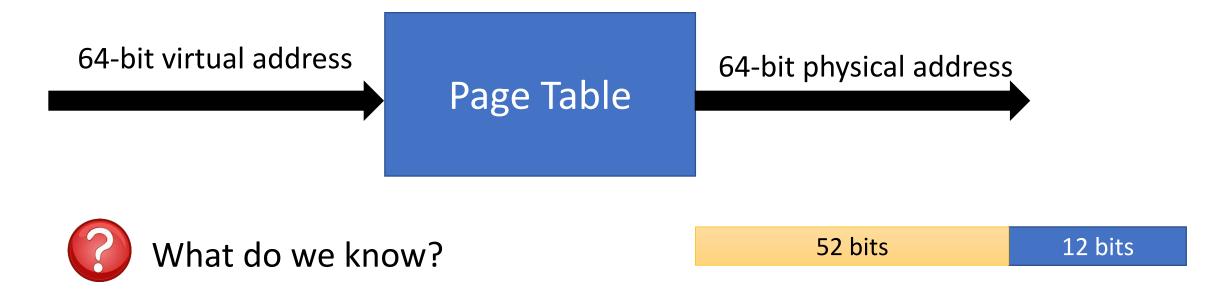
Segmentation in x86-64

- Nowadays with x86 (64 bits) only two segment registers are used: fs and gs
- The GDT is also not used. It has been replaced by MSRs (model specific registers)
- The Linux kernel uses the gs segment to store per-CPU data.
- The gcc compiler also uses them to store thread-local data

Example

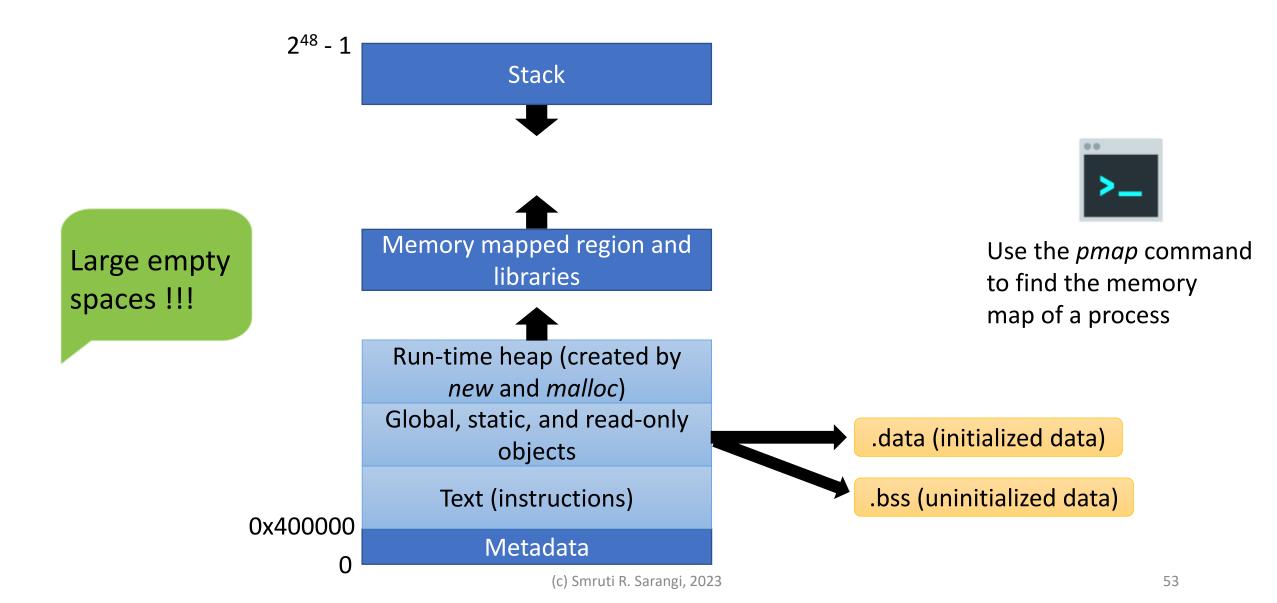
movl \$32, %fs:(%eax)

How is the Page Table Designed?



- We know that a page or a frame have a size of 4 KB (2¹² bytes)
- The virtual page or physical frame address is thus 52 bits
- We cannot possibly create a structure that has 2⁵² entries

Leverage a Pattern (Use a Memory Map)



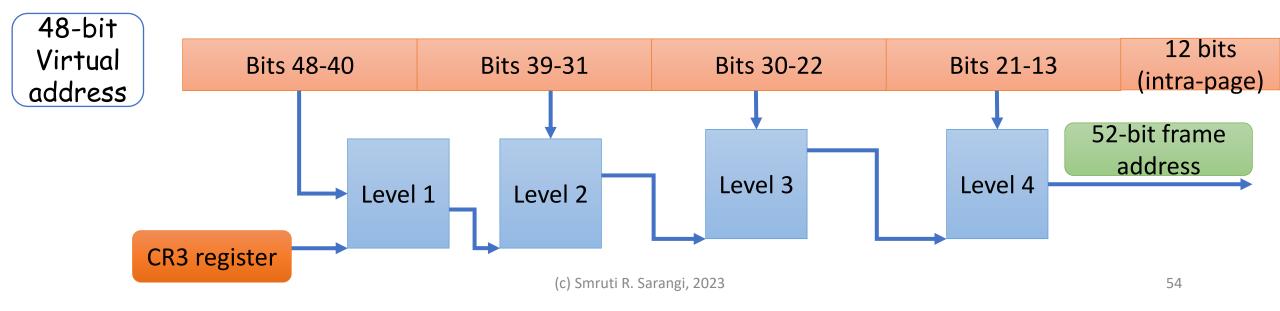
Design a Data Structure that Leverages the Structure of the Memory Map



- LSB bits have more randomness,
- MSB bits have less randomness
- The MSB bits determine the memory region

Multi-level Page Table

Consider the first (36 + 12) bits in the 64-bit x86 memory address



Page Protection and Information Bits

Each entry of the page table or TLB has additional page protection bits.

| Bit | Function |
|---------|---------------------------------------------|
| Present | Present in the main memory or not |
| RW | Set if the page can be written to |
| User | If the page can be accessed from user space |
| Dirty | Has a page been written to |



These bits can be used to make a page read-only. This is a vital security measure for pages that contain code.

Questions on Efficiency



Where does a page table save memory?

Very few entries in the Level 1 page table are full. This means that there are a few Level 2 page tables. There are slightly more Level 3 page tables and much more Level 4 page tables. The sparse structure of the memory map minimizes the number of page tables.



Do we access the page tables on a memory access?







What do we do then?

Answer: Use a HW structure to cache frequent mappings. It is the TLB (Translation Lookaside Buffer). We cache 32-128 entries. We can access in 1 cycle.

Memory Management

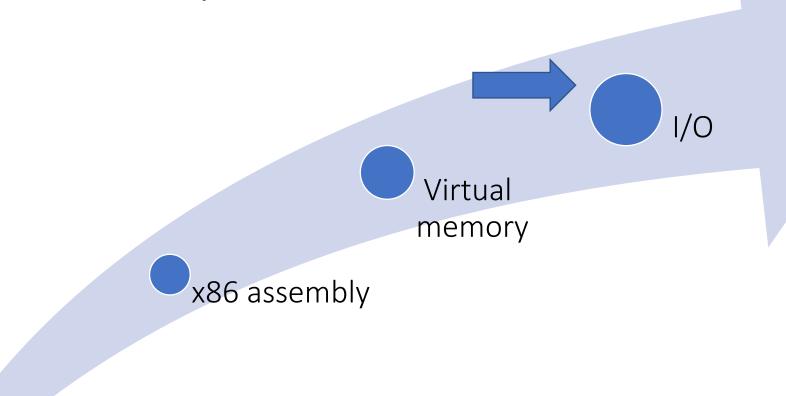
- Before accessing data, it needs to be present in the main memory, regardless of wherever it is stored.
- The main memory has finite size.
- If it is full, then we need to evict a frame to make space.

Which one 🕜

- There are different heuristics: least recently used, least frequently used in a given timeframe, most frequently used, FIFO, random, etc.
- An optimal solution exists: Evict the frame that will be used farthest in the future



Outline of this Chapter



Cores, registers and interrupts

The Motherboard and Chipset

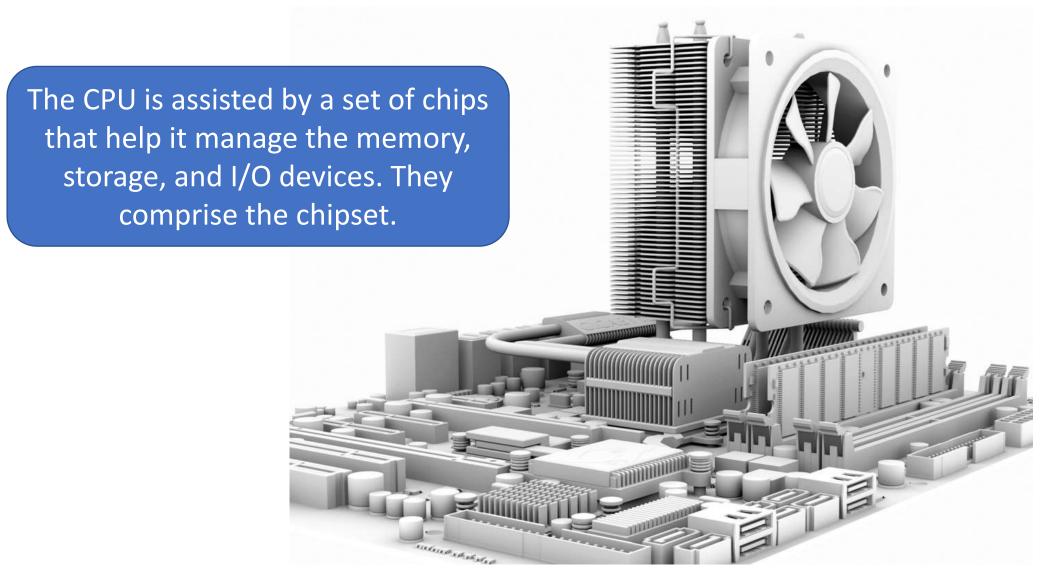
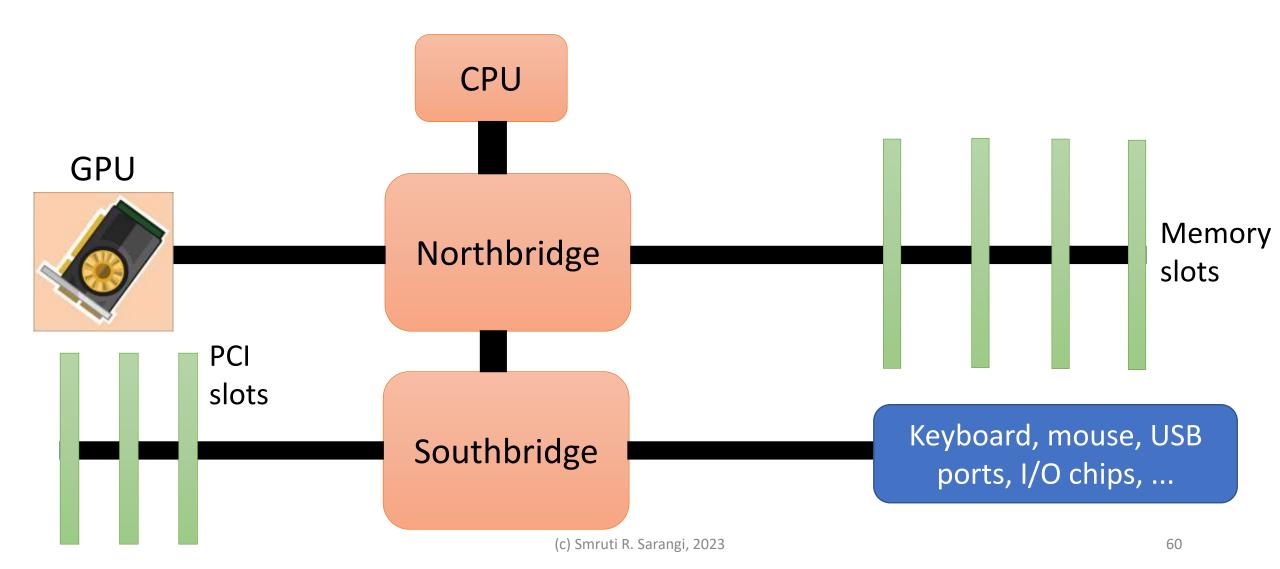


Diagram of the Motherboard



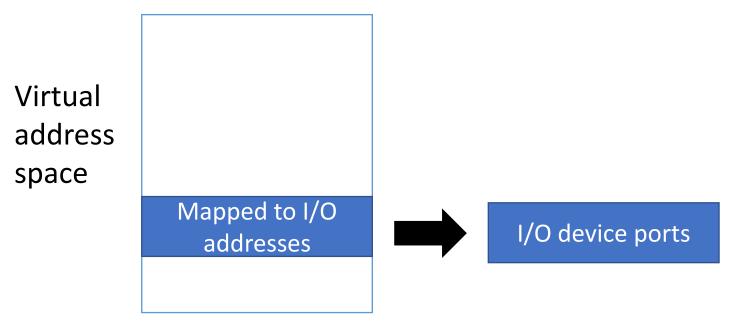
x86 I/O Instructions

- When the system boots, each I/O device is provided a 16-bit I/O address
- The OS can query this information and figure out the I/O addresses associated with each device.
- These are known as ports.
- It is possible to write a value to a port or read a value from it.
- x86 has dedicated in and out instructions



Scalability is an issue. We cannot read and write a lot of data in one go.

Memory Mapped I/O



- Use the virtual memory mechanism to map a portion of the virtual address space to I/O devices
- Use regular reads and writes to access I/O devices
- The system automatically routes memory traffic to the I/O devices (bulk transfers possible)

DMA (Direct Memory Access)

- Assume that a large amount of data (several MBs) needs to be transferred from the hard disk to the main memory.
- Why should the program involve itself in this process, if this can be outsourced to a separate chip – the DMA controller?
 - Just give it a pointer to the region in the storage device (hard disk in the case), and the main memory. It does the transfer on its own and interrupts the chip once done.





srsarangi@cse.iitd.ac.in





