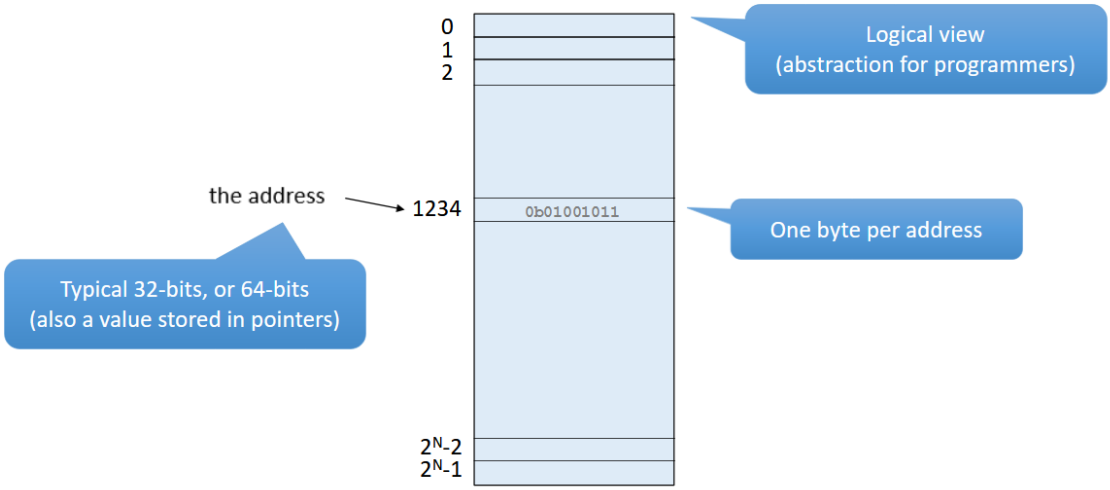
Memory

- Definition
  - Each memory is organized into memory cells – bits
  - Bits are grouped into words of fixed length
    - 1, 2, 4, 8, 16, 32, 64, and 128 bits
  - Each word can be accessed by a binary address
    - N bits (usually 32 or 64 on contemporary architectures)
    - We can store 2^N words in the memory
  - Today, the 8-bit word is used exclusively
    - Byte

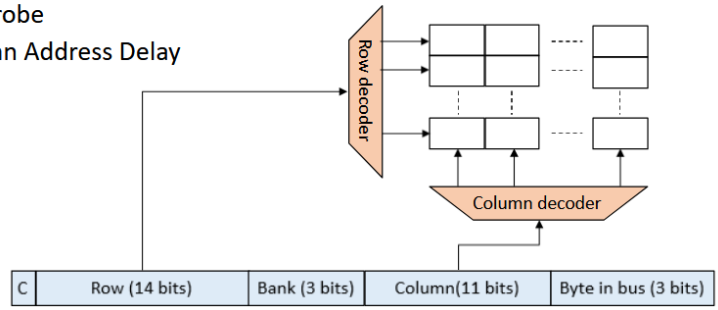
This is the basis for addressing, but many architectures have "native" support only for larger words (e.g., 32 bit)

Memory – address space



Memory – physical view

- 2D array
  - Row x column
  - Select, access, deselect row
  - Timing
    - CAS (tCL) – Column Access Strobe
    - tRCD – Row Address to Column Address Delay
    - tRP – Row Precharge
    - RAS (tRAS) – Row Active Time

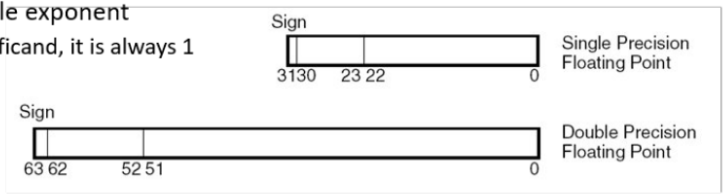


Data representation – integers

- Unsigned numbers
  - Simple binary representation of a number
  - Usual sizes
    - 1, 2, 4, 8 bytes
  - Represented range
    - $[0; 2^N-1]$
- Signed numbers
  - Two's complement
    - Bitwise negation + 1
    - One 0
    - Compatible with unsigned arithmetic
  - Asymmetric range
    - $[-2^{N-1};2^{N-1}-1]$

Data representation – floats

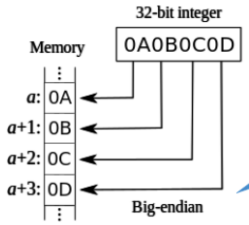
- IEEE 754 floating point numbers
  - Hidden bit convention
    - Memory representation for single-precision, double-precision
    - Use the smallest representable exponent
      - Hide the leading bit of significand, it is always 1
  - Exponent (unsigned)
    - Bias (SP=127, DP=1023)
    - Special values
  - Value
    - $V = (-1)^{sign} \times significand \times 2^{exponent-bias}$



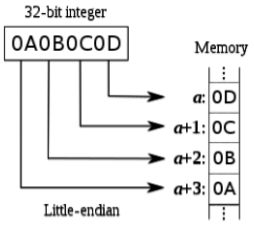
CPUs may internally use different representation (more suitable for circuits, but more redundant)

Data representation- Endianness

- How to store multi-byte numbers?
- Big endian
  - MSB first, LSB last
  - PowerPC, …
- Little endian
  - LSB first, MSB last
  - Intel (x86)
- Example
  - Store 32-bit number 0x0A0B0C0D



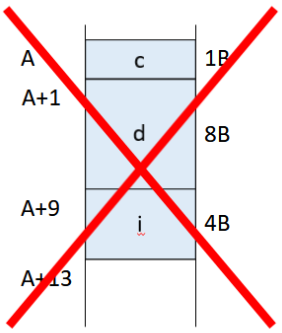
Btw. big endian is used in TCP/IP protocols

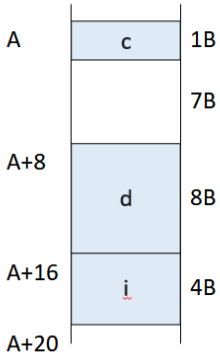


Data alignment – inner padding

- Modern CPUs require data in memory aligned to their size
  - E.g., integer (4B) must have address aligned to 4
  - Structure is aligned to largest data type available on CPU (e.g., 16B)

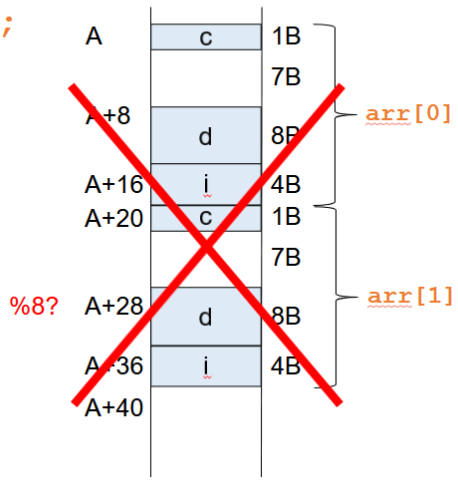
```
struct dem {
  char c;
  double d;
  int i;
};
```

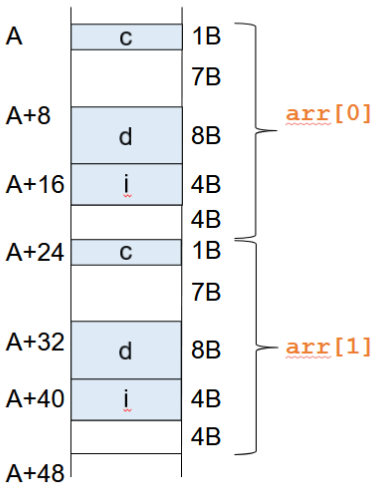




Data alignment – outer padding

```
dem arr[2];
```





Instruction

- Simple command to the CPU
  - Binary encoding (CPU), assembler (programmers)

MIPS32 Add Immediate Instruction

001000	00001	00010	0000000101011110
OP Code	Addr 1	Addr 2	Immediate value

Fixed vs. variable lengths

Equivalent mnemonic: `addi $r1, $r2, 350`

- Operands
  - Depending on ISA – max 1, 2, or 3 (some may be implicit)
  - Register, immediate value

Instruction cycle

- Sub-steps performed in every instruction

- Load an instruction from the address stored in IP register
- Decode the instruction
- Load operands
- Execute the operation
- Store the result
- Increment IP

Some steps may be skipped for some instructions

Instruction classes

- Load instructions
  - Memory -> register
  - Take a long time to execute, important to detect soon (fetch data ahead)
- Store instructions
  - Register/immediate -> memory
- Move instruction
  - Between registers
  - x86-64 also between registers and memory
    - Difficult to implement efficiently in HW

Instruction classes

- Arithmetic and logic instructions
  - `+, -, <<, >>, &, |, ^, ~`
  - `*, /, %`

No funny stuff like `"str" * 4`

- Jumps
  - Unconditional × conditional
  - Direct × indirect × relative

Tests required (==, <, ...)

- Call, return
- ...

Higher-level code structures

```
if (a < 3) b = 4; else c = a << 2;
```

```
...
jge [a], 3
store [b], 4
jmp
load r1, [a]
shl r1, 2
store [c], r1
...
```

This is just a symbolic abstract assembler (not an actual one)

Higher-level code structures

```
for (int i = 0; i < 5; ++i) a[i] = i;
```

```
mov r1, 0
jge r1, 5
load r2, [a]
add r2, r2, r1
store [r2], r1
add r1, r1, 1
jmp
...
```

Actually, the offset needs to be multiplied by `sizeof(a[0])`, but you got the point...

MIPS – instructions

- Arithmetic
  - `add $rd,$rs,$rt`
    - R[rd] = R[rs] + R[rt]
  - `addi $rd,$rs,imm16`
    - R[rd] = R[rs] + signext(imm16)
  - `sub $rd,$rs,$rt`
  - `subi $rd,$rs,imm16`

These are only symbolic placeholders, real example could look like `add $r2,$r4,$r5`

Immediate value, 16 bit number encoded directly in the instruction itself

Signed extension to 32bits (to match size of the registers)

ISA comparison

MIPS	x86
`add $t1,$t1,$t0`	`add eax,ebx`
`addi $t1,$t1,1`	`add eax,1`
	or `inc eax`
`add $t2,$t0,$t1`	`mov eax,ebx`
	`add eax,ecx`

MIPS – instructions

- Logic operations
  - `and $rd,$rs,$rt` `andi $rd,$rs,imm16`
  - `or $rd,$rs,$rt` `ori $rd,$rs,imm16`
  - `xor $rd,$rs,$rt` `xori $rd,$rs,imm16`
  - `nor $rd,$rs,$rt`
    - R[rd] = R[rs] and/or/xor zeroext(imm16)
  - No **not** instruction, use `nor $rd,$rs,$rs`
- Shifts
  - `sll` / `slr` `$rd,$rs,shamt`
    - R[rd] = R[rs] << / >> shamt
  - `sra $rd,$rs,shamt`

Arithmetic shift (keeps the sign)

ISA comparison

MIPS	x86
`nor $t1,$t2,$t2`	`mov eax,ebx`
	`not eax`
`sll $t1,$t1,3`	`shl eax,3`

MIPS – instructions

- Memory access
  - `lw $rd,imm16($rs)`
    - R[rd] = M[R[rs] + signext32(imm16)]
  - `sw $rt,imm16($rs)`
    - M[R[rs] + signext32(imm16)] = R[rt]
  - `lb $rd,imm16($rs)`
    - R[rd] = signext32(M[R[rs] + signext32(imm16)])
  - `lbu $rd,imm16($rs)`
    - R[rd] = zeroext32(M[R[rs] + signext32(imm16)])
  - `sb $rt,imm16($rs)`
    - M[R[rs] + signext32(imm16)] = R[rt]
- Moves
  - `li $rd,imm32`
    - R[rd] = imm32
    - Pseudo-instruction, translates to `lui` and `ori`
      - Load upper immediate
  - `move $rd,$rs`
    - R[rd] = R[rs]

ISA comparison

MIPS

```
lw   $t1,1234($t0)
sw   $t1,1234($t0)
lb   $t1,1234($t0)
li   $t1,5678
move $t1,$t0
```

x86

```
mov eax,[ebx+1234]
mov [ebx+1234],eax
mov al,[ebx+1234]
mov eax,5678
mov eax,ebx
```

MIPS – instructions

- Jumps
  - `j addr`
    - PC = addr
  - `jr $rs`
    - PC = R[rs]
  - `jal addr`
    - "Jump and link"
    - R[31] = PC+4
    - PC = addr

Slightly different instruction format – the address is a 26bit immediate value

`jal` instruction (like all other instructions) takes 4 bytes in memory

ISA comparison

MIPS

```
    j   label
    jr  $ra

label:
    jal fnc
```

Placeholder that marks a place in the code, the actual address is computed by a compiler

x86

```
    jmp label
    jmp [ebx]

label:
    call fnc
```

This is double indirection!!! Register **ebx** holds address where is the pointer that is loaded and used as target address for the jump!

Return address stored to stack!

MIPS – instructions

- Conditional jumps
  - `beq $rs,$rt,addr`
    - If R[rs] == R[rt] then PC = addr else PC = PC+4
  - `bne $rs,$rt,addr`
    - Analogical (not equal)
- Testing
  - `slt` / `sltu $rd,$rs,$rt`
    - If R[rs]<R[rt] then R[rd] = 1 else R[rd] = 0
  - `slti` / `sltiu $rd,$rs,imm16`
    - If R[rs] < signext/zeroext(imm16) then R[rd] = 1 else R[rd] = 0

`sltu` is unsigned version of `slt`

Only lesser-than test, greater-than is created by swapping operands

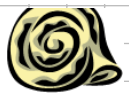
ISA comparison

MIPS	x86
`beq $t0,$t1,label`	`cmp eax,ebx` `jz label`
`slt $t2,$t1,$t0` `bne $t2,$zero,label`	`cmp eax,ebx` `jl label`
`slti $t2,$t1,5` `bne $t2,$zero,label`	`cmp eax,5` `jl label`

Universal compare (result stored in flags)

Jump decision based on flags

Code examples



- Simple for-loop

```
for (int i = 0; i < N; i++)
{
  A[i] = 42;
}
```

```
    addi  $t0, $gp, 28      # $t0 <- address of A
    lw    $t1, 4($gp)       # fetch N
    move  $t2, $zero        # i = 0
    ori   $t3, $zero, 42    # $t3 = 42
    j     cond              # go to condition
body:
    sll   $t4, $t2, 2       # i*4 -> offset
    add   $t4, $t4, $t0     # A+i*4
    sw    $t3, 0($t4)       # A[i] = 42
    addi  $t2, $t2, 1       # i = i+1
cond:
    slt   $t4, $t2, $t1     # are we there yet?
    bne   $t4, $zero, body  # no, we're NOT done
```

Code examples

```
i = N*5+3;
```

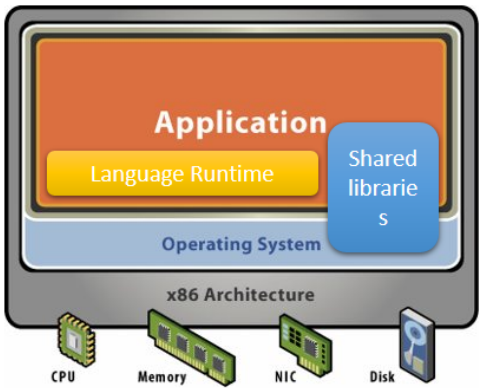
```
lw    $t0, 4($gp)     # fetch N
ori   $t1, $zero, 5   # 5
mult  $t0, $t0, $t1   # N*5 (fake)
addi  $t0, $t0, 3     # N*5 + 3
sw    $t0, 0($gp)     # i = ..
```

Multiplication is more complex as the result needs to be stored in 2 registers (but we have simplified it here)

```
lw    $t0, 4($gp)     # fetch N
sll   $t1, t0, 2      # N*4
add   $t1, $t1, t0    # N*4+N
addi  $t1, $t1, 3     # N+3
sw    $t1, 0($gp)     # i = ...
```

Code can be optimized...

Operating system – role



- Abstract machine
  - Presented by kernel API
    - System calls
    - Wrapped in C libraries
  - Hide HW complexity/diversity
- Resource manager
  - All HW managed by OS
  - Sharing HW among applications
    - Allocation (memory)
    - Time-sharing (CPU)
    - Abstraction (disk, network)

CPU modes

- User mode
  - Available to all application
  - Limited (e.g., read-only) or no access to some resources
    - The system registers, instructions
- Kernel (system) mode
  - More privileged (all registers and instructions are available)
  - Used by OS or by only part of OS
  - Full access to all resources
- The transition between the modes (especially user -> kernel)
  - Syscall (user instruction), jumps to the explicit kernel entry point

Kernel mode example

- Print string to std. output
  - Note that requires kernel call

```
int main() {
  printf("Hello");
  ...
}
```

```
movq   rdx, @len
movq   rcx, @@msg
movq   rbx, 1
movq   rax, 4
syscall

...

sysret
```

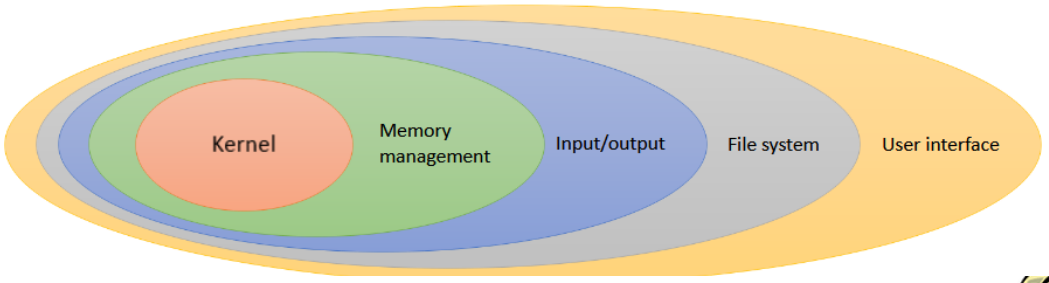
x86 64bit

Jumps to address stored in IA32_LSTAR register whilst switching to ring 0 (kernel mode)

Return and switch to ring 3

Architecture – layered

- Evolution of monolithic system
  - Organized into hierarchy of layers
  - Layer n+1 uses exclusively services supported by layer n
  - Easier to extend and evolve



Devices

- Terminology
  - Device - “a thing made for a particular purpose”
  - Device controller
    - Handles connected devices electrically (signals “on wires”, A/D converters)
    - Devices connected in a topology
  - Device driver
    - SW component (piece of code), part of OS (module, dynamically loaded)
    - Abstract interface to the upper layer in OS
    - Specific for a controller or a class/group of controllers
  - BIOS/UEFI
    - Basic HW interfaces that allow to enumerate and initialize devices on boot

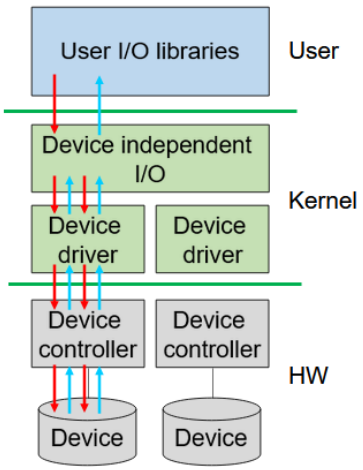
Device communication

- How CPU performs I/O operations
  - Specialized instructions
    - `in al, dx`
    - `out al, dx`
  - Memory-mapped devices
    - Device operating structures are mapped at fixed addresses (at boot)
    - Basic memory operations are translated by HW into I/O operations
      - Reads and writes may trigger some device functions
      - E.g., writing at a specific offset may in fact send a command to the device

Old x86 input-output instructions
(dx holds the IO port number associated with the device)

Device handling

1. Application issues an I/O request
2. Language library makes a system call
3. The kernel decides, which device is involved
4. The kernel starts an I/O operation using a device driver
5. The device driver initiates an I/O operation on a device controller
6. Device does something
7. The device driver checks for the status of the device controller
8. When data are ready, transfer data from the device to the memory
9. Return to any kernel layer and make other I/O operations fulfilling the user request
10. Return to the application



Example – PIO disk handling

Port offset	Function
0	Data register
1	Error register
2	Sector count register
3	Sector number register
4	Cylinder low register
5	Cylinder high register
6	Drive/head register
7	Status/Command register

Ports relative to the device base address operated by in/out instructions

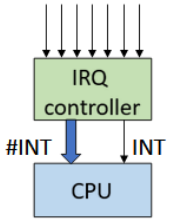
- Reading sectors (simplified)
  1. Check status/error
  2. Set count register
  3. Set address (sector and cylinder)
  4. Send reading command **`0x20`**
  5. Read the status register until the operation completes (or fails)
  6. Read the data register 256 times (based on the size of the data)
  7. Possibly repeat steps 5 & 6

Device intercommunication

- Polling
  - CPU actively checks device status change (like in previous example)
- Interrupt
  - The device notifies the CPU that it needs attention
  - CPU interrupts the current execution flow
  - IRQ (interrupt request) handling
  - CPU has at least one pin for requesting an interrupt
- DMA (Direct Memory Access)
  - Transfer data to/from a device without CPU attention
  - DMA controller
  - Scatter/gather

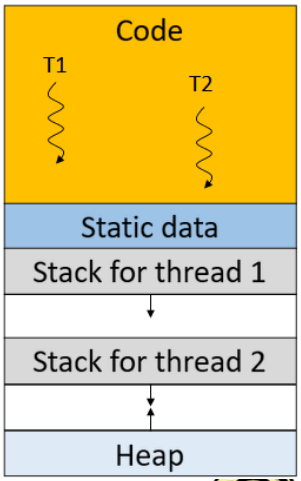
Interrupt request handling

- What happens, when an interrupt occurs?
  - CPU decides the source of the interrupt
    - Predefined or IRQ controller
  - CPU gets the address of the interrupt handler
    - Fixed (defined by ISA)
    - Interrupt table (array of pointers of handlers for individual interrupts)
  - The current stream of instructions is interrupted, CPU begins execution of interrupt handler's instructions
    - Usually **between** instructions (current instruction must be complete or rollback)
    - A privilege switch usually happens, the interrupt handler is part of a kernel
    - An essential part of the CPU state (at least IP) must be saved (e.g., to a special register)
  - The interrupt handler saves the (rest of the) CPU state
  - The Interrupt handler does something useful (its job)
  - The interrupt handler restores the CPU state
  - CPU continues with the original instruction stream



Processing

- Program
  - A passive set of instruction and data
  - Created by a compiler/ linker
- Process
  - An instance of a program created by OS
  - Program code and data
    - Process address space
  - The program is “enlivened” by an activity
    - Instructions are executed by CPU
  - Owns other resources
- Thread
  - One activity in a process
  - Stream of instructions executed by CPU
  - Unit of kernel scheduling
    - I.e., Holds CPU context
- Fiber
  - Lighter unit of scheduling
  - Cooperative scheduling
    - Running fiber explicitly yields



Process vs. Thread

- Process
  - Code (loaded in memory)
  - Memory space
  - Other system resources
    - File handles
    - Network sockets
    - Synchronization primitives
    - …
- Thread
  - Position in code (program counter)
  - Own stack (rest is shared)
  - Access to some system resources may require synchronization
  - CPU state
    - Must be saved when thread is removed from CPU core and reloaded when the thread resumes

Processing

- Scheduler
  - Part of OS
  - Uses scheduling algorithms to assign computing resources to scheduling units (CPU cores)
- Multitasking
  - Concurrent executions of multiple processes
- Multiprocessing
  - Multiple CPUs (cores) in one system
  - More challenging for the scheduler
  - Affinity

The main distinction between a thread switch and a process switch is that during a thread switch, the virtual memory space remains the same, while it does not during a process switch. Both types involve handing control over to the operating system kernel to perform the context switch. The process of switching in and out of the OS kernel along with the cost of switching out the registers is the largest fixed cost of performing a context switch.

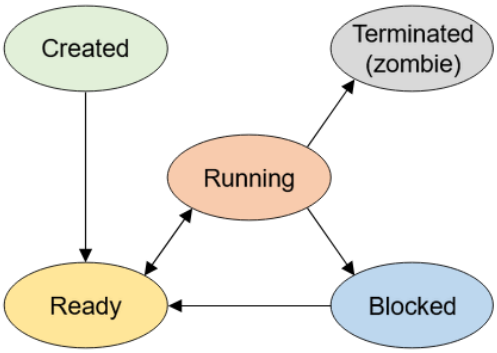
A more fuzzy cost is that a context switch messes with the processors cacheing mechanisms. Basically, when you context switch, all of the memory addresses that the processor "remembers" in its cache effectively become useless. The one big distinction here is that when you change virtual memory spaces, the processor's Translation Lookaside Buffer (TLB) or equivalent gets flushed making memory accesses much more expensive for a while. This does not happen during a thread switch.

Processing

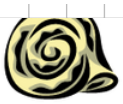
- Context
  - CPU (and possibly other) state of a scheduling unit
    - Registers (including PC, specialized vector registers)
    - Additional units (x87 coprocessor)
    - Virtual memory and address-space-related context
      - Page tables, TLB (will be covered later)
      - Memory caches are transparent (not part of the context, but may affect performance)
- Context switch
  - Process of storing the context of a scheduling unit (when suspended) and restoring the context of another scheduling unit (when resumed)
  - Quite costly (hundreds-thousands of instructions)

~~Unit of scheduling state~~ State of scheduling unit

- Created
  - Awaits admission
- Terminated
  - Until the parent process reads the result
- Ready
  - Wait for scheduling
- Running
  - CPU assigned
- Blocked
  - Wait for resources



Multitasking



- Cooperative
  - Unit of scheduling must explicitly and voluntarily yield control
  - All processes must cooperate
    - Special systems
  - Scheduling in OS reduced on starting the process and making context switch after the yield
    - OS does not initiate a context switch
- Preemptive
  - Each running unit of scheduling has an assigned **time slice**
  - OS needs some external source of **interrupt** (HW timer)
  - If the unit of scheduling blocks or is terminated before the time slice ends, nothing of interest happens
  - If the unit of scheduling consumes the whole time slice
    - interrupted by the external source
    - changed to READY state
    - OS will make a context switch

Scheduling

- Objectives
  - Maximize/optimize CPU utilization (based on the workload)
  - Fair allocation of CPU
  - Maximize throughput
    - Number of processes that complete their execution per time unit
  - Minimize turnaround time
    - The amount of time taken by a process to finish
  - Minimize waiting time
    - Time a process waits in the READY state
  - Minimize response time
    - Time to respond in interactive applications

Scheduling – priority

- Priority
  - A number expressing the importance of the process
  - Unit of scheduling with greater priority should be scheduled before (or more often than) unit of scheduling with lower priority
  - The priority of the process is the sum of a static priority and dynamic priority
    - Static priority
      - Assigned at the start of the process
        - Users' hierarchy or importance
    - Dynamic priority
      - Adding fairness to the scheduling
      - Once in a time, the dynamic priority is increased for all READY units of scheduling
      - The dynamic priority is initialized to 0 and is reset to 0 after the unit of scheduling is scheduled for execution

Scheduling algorithms – non-preemptive

- First Come, First Serve (FCFS)
  - Single FIFO queue
  - A process enters the queue on the tail, the head process is running on the CPU
  - Afterward, there is removed from the queue
- Shortest Job First
  - Maximizes throughput
  - Expected job execution time must be known in advance
- Longest Job First

Scheduling algorithms – preemptive

- Round Robin
  - Like FCFS (but preemptive)
  - Single queue
  - Each unit of scheduling has an assigned time slice
  - If the unit of scheduling consumes the whole time slice or is blocked, it will be assigned to the tail of the queue

Scheduling algorithms- preemptive

- Completely fair scheduler (CFS)
  - Implemented in Linux kernel
    - Currently the default scheduler
  - SUs are stored in a red-black tree
    - Indexed by their total execution time (called *virtual runtime*)
    - One tree per CPU core
  - *Maximum execution time*
    - A time slice calculated for each unit
    - Total waiting time divided by the current number of processes
      - The longer it waits, the greater
- Scheduling algorithm
  - The leftmost node in the RB tree is selected (lowest virtual runtime)
  - If the process completes its execution, it is removed from the schedule
  - If the process reaches its *maximum execution time* or is somehow stopped or interrupted, it is reinserted into the tree with a new time key
    - Actual time spent on the CPU is added to the virtual runtime
- Virtual runtime decays over time

Virtual memory

- Basic concepts
  - All memory accesses from instructions work with a **virtual** address
    - Virtual address space
    - Even instruction fetch
  - Operating memory provides **physical** memory
    - Physical address space
    - Always 1-dimensional
    - The memory controller uses physical addresses
  - Translation mechanism
    - Implemented **in HW** (MMU embedded in CPU)
    - Translates a virtual address to a physical address
    - The translation (mapping) **may not exist -> exception (fault)**
    - Two basic mechanisms – segmentation, paging

Virtual memory

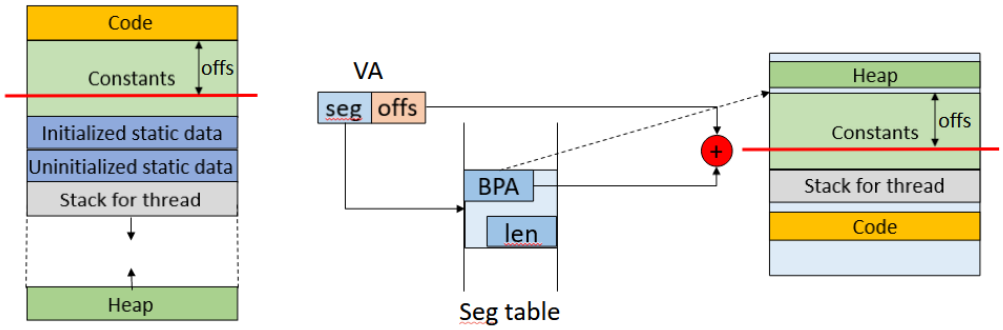
- Why?
  - More address space
    - VAS can be larger than PAS (an illusion of having large memory)
      - Today, IA-32 can have larger PAS than VAS
    - Add secondary storage as a memory backup/swap
    - This is no longer the primary reason today
  - Security
    - Process address space separation
    - "Separation" of logical segments in a process address space (read-only, executable, …)
  - Specialized (advanced) operations
    - Memory-mapped I/O (e.g., memory-mapped file)
    - Controlled memory sharing

Segmentation

- Concepts
  - Virtual (process) address space divided into logical segments
    - Segments are numbered
    - may have different sizes
  - Virtual address has two parts
    - [segment number; segment offset]
    - Offsets 0-based for each segment
  - Segment table (translation data structure)
    - In memory, for each process
    - Stores base physical address, length, and attributes for each segment
    - Indexed by the segment number
    - Segment fault (if translation or validation of access fails)

Segmentation

- Schema



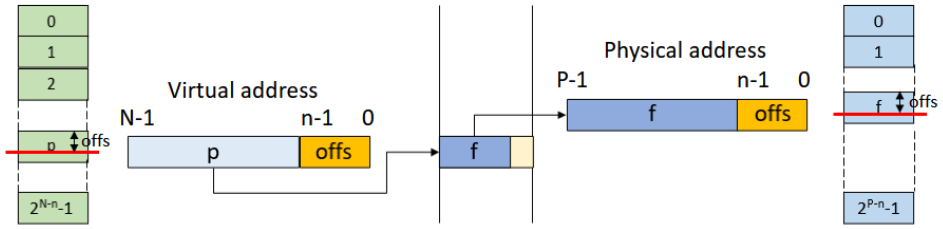
Paging

- Concepts
  - VAS divided into **equal** parts
    - Page, 2^n size
  - PAS divided into **equal** parts
    - Frame, equal size with page (i.e., one page fits exactly one frame)
  - VA 1-dimensional
  - Page table (translation data structure)
    - In memory, for each process
    - Indexed by a page number
    - Each entry contains a frame number and attributes (P)
    - Page fault

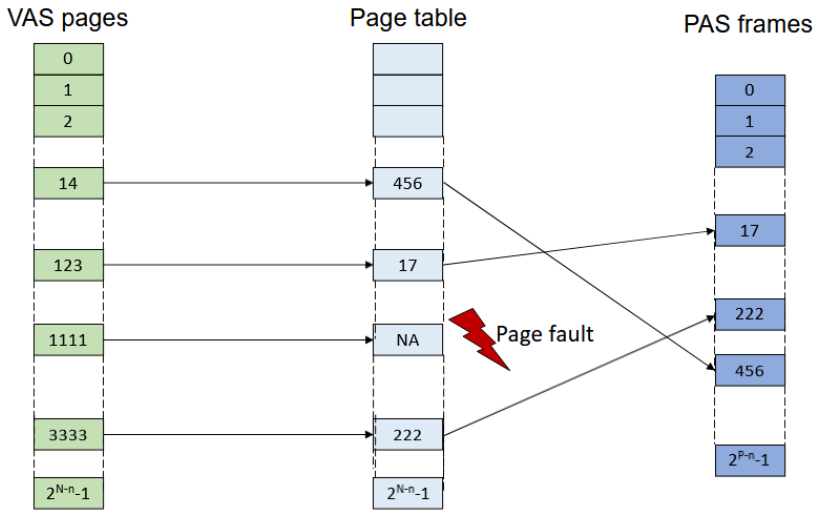
This is very important!

Paging

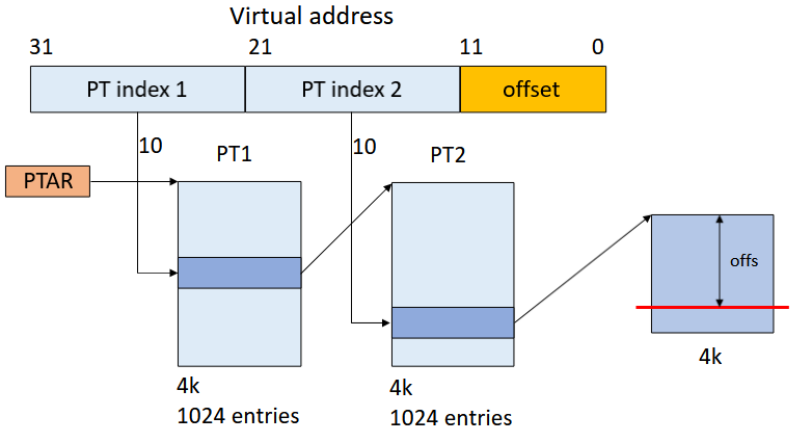
- Concepts



Page table – 1-level



Page table – 2-level



Example/Exercise

- Having the following code executed on IA-32
  - 32bit addresses
  - 2-level paging, 4 KiB pages
  - `sizeof(int) == 4`

```
const int* data = …
long long sum = 0;
for (int i = 0; i < 2000; ++i) {
  sum += data[i];
}
```

- Questions
  - How many (data) pages are read?
  - What is the minimal amount of page faults (optimistic scenario)
  - What is the maximal amount of page faults (pessimistic scenario)
  - How many **distinct** pages may cause a page fault in the worst case?
    - What if we copy the data?
    - What about the code?

The data block is 4 * 2000 = 8000B long, i.e., it can span 2 or 3 (4KiB) pages depending on the actual address of data pointer.

Optimistic and pessimistic scenarios are trick questions. The answers are 0 and infinity.

The block spans over 3 pages (worst case), so it is covered by two 2nd level page tables (worst case) and the first level is always in the memory (i.e., 5 distinct faults in total).

If we copy the data, the number will double (we need to count source and destination pages, unless there are some assumptions about from where to where the data are being copied).

The code may also cause some page faults. This will definitely be at least a few instructions, but clearly the loop does not require more than 4 KiB of code (it is simple enough), so it can span 2 pages at the worst + 2 (2nd level) page tables = 4 additional page faults.

Paging – address translation

- Steps for address translation
  - Take the page number from VA and keep offset (separately)
  - Check TLB for mapping
    - If exists, retrieve the frame number, otherwise continue
  - Go through the page table
  - Update A(ccessed) and D(irty) bits in page table/TLB
  - Assemble PA by concatenating the retrieved frame number and the original offset from VA

- Go through the page table
  - Divide page number into multiple PT indices
  - Index 1st level PT
    - If there is no mapping for 2nd level PT, raise page fault exception
    - Retrieve PA for 2nd level PT and continue
  - Go through all levels of PTs
    - If there is no mapping in any PT level, raise page fault exception
    - If all PT levels are mapped, retrieve frame number
  - Save retrieved mapping to TLB

File

- File
  - Data organization unit
  - Collection of related information
    - Abstract stream of data (bytes)
      - Kernel does not understand file formats
    - Typically stored on secondary storage, but there are other possibilities
  - File identification
    - System uses numeric identifiers
    - File name and path – a named reference to the file identifier in organized tree structure
      - So that humans can find the files
      - Some parts of the file name may have special meaning (leading dot, extension)

File operations

- Std. lib in C

```c
#include <stdio.h>

FILE *fp = fopen("file.txt", "r");
if (!fp) { /* error */ }

fseek(fp, 42, SEEK_SET);
fgets(buf, 16, fp);

fclose(fp);
```

- POSIX

```c
#include <unistd.h>

int fd = open("file.txt", O_RDONLY);
if (fd == -1) { /* error */ }

lseek(fd, 42, SEEK_SET);
read(fd, buf, 16);

close(fd);
```

File operations

- Additional operations
  - Create, truncate, delete, flush, change attributes
- File handle
  - Process-specific sequentially assigned, kernel holds translation table
- Buffering
  - To increase performance, multiple levels (system, language runtime)
  - Sequential vs random access
- Alternatives
  - Memory mapping (will become more clear after memory management)
  - Async file I/O

It is no coincidence that **`stdin`**, **`stdout`**, and **`stderr`** file descriptors have handles 0, 1, and 2.

Race condition

- Race condition
  - Multiple threads accessing (updating) the same data in shared memory space
    - Cache coherency helps a little, but does not actually solve anything
    - Load-store architecture makes it more pronounced
  - The result of a computation depends on the sequence or timing of units of scheduling

```
class List {
  private:
    Node *root;

  public:
    void PushFront(Node *n) {
      n->next = root;
      root = n;
    }
};
```

Race condition

- Shared variable

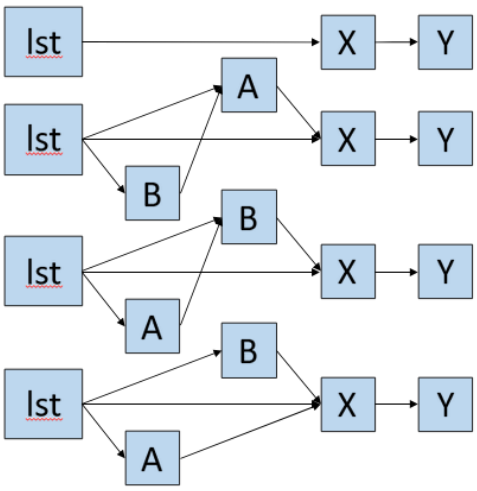
```
List lst;
```

- Thread 1

```
lst.PushFront(A);
```

- Thread 2

```
lst.PushFront(B);
```



Critical section

- Problem definition
  - Concurrent access to a shared resource can lead to the race condition or even to an undefined behavior
- Solution
  - Parts of the program, where the shared resource is accessed, need to be protected to avoid concurrent access
- Critical section
  - Protected section of the program
- Mutual exclusion
  - A critical section can be executed simultaneously by at most one unit of scheduling

Synchronization

- Process synchronization
  - Multiple units of scheduling do some form of a handshake at a certain point to make an agreement to a certain sequence of action
- Data synchronization
  - Keeping multiple copies of data in coherence with each other
  - Maintain data integrity
  - Usually implemented by process synchronization
- Problems with synchronization
  - Deadlock, starvation, overhead…

Synchronization primitives

- Synchronization primitives
  - Implement process synchronization (in OS)
  - Active
    - Instructions are executed during waiting for an access
      - Busy waiting (testing a condition in a loop)
  - Passive/blocking
    - The unit of scheduling is blocked until access is allowed
- Hardware support
  - Atomic instructions
    - Test-and-set (TSL), compare-and-swap (CAS)
    - Instruction semantics:

```
bool cas(T* var, T old, T newVal)
{
  if (*var != old) return false;
  *var = newVal;
  return true;
}
```

This is realized as **one instruction**!

Synchronization primitives

- Spin-lock
  - Busy waiting using TSL/CAS
  - Short latency, right for short waiting times
- Semaphore
  - Protected counter and a queue of waiting for US
  - Atomic operations UP and DOWN

```
void down() {
  if (counter > 0) --counter;
  else {
     queue.push(myUS);
     myUS.block();
  }
}
```

```
void up() {
  if (counter == 0 && !queue.empty()) {
    US = queue.popany();
    US.unblock();
  }
  else ++counter;
}
```

Synchronization primitives

- Mutex
  - Implements mutual exclusion (semaphore with counter = 1)
  - Atomic operations LOCK and UNLOCK (corresponding to UP and DOWN)
- Barrier
  - Multiple units of scheduling meet at the same time on the same barrier
- Specific programming language constructs
  - Monitor
    - Methods in an object executed with mutual exclusion
    - Possibility to wait on a certain condition
  - Java/C#
    - Keyword `synchronized`/`lock`

Creates a critical section