

Computer Architecture and System Programming

Summary of the course in autumn 2009 by Thomas Gross and Stefan Freudenberger

Stefan Heule

2010-01-24

Table of Contents

1	Data representation	5
1.1	Bits and bytes	5
1.1.1	Basics	5
1.1.2	Byte ordering	5
1.1.3	Strings in C	5
1.1.4	Machine-level code representation	5
1.1.5	C – bit-wise and logical operations	6
1.2	Integers	6
1.2.1	Encoding integers	6
1.2.2	Numeric ranges	6
1.2.3	Casting / unsigned vs. signed	7
1.2.4	Addition	7
1.2.5	Multiplication	8
1.2.6	Properties of integer arithmetic	8
1.3	Floating point numbers	8
1.4	Arrays	8
1.4.1	Basic principle	8
1.4.2	Nested arrays	9
1.4.3	Multi-level arrays	9
1.4.4	Dynamic nested arrays	9
1.5	Structures and Unions	9
1.5.1	Concept of structres	9
1.5.2	Unions	9
2	Machine-level programming	10
2.1	Basics	10
2.1.1	Assembly programmer’s view	10
2.1.2	Assembler	10
2.2	Instructions	10
2.2.1	Moving data	10
2.2.2	Address computation instruction	11
2.2.3	Some arithmetic operations	11
2.2.4	Condition codes	11
2.3	IA32 Stack	11
2.3.1	Push/pop	11
2.3.2	Procedure control flow	12
2.3.3	IA32/Linux stack frame	12
2.3.4	Register saving conventions	12
3	Sequential processors	13
3.1	Instruction set architecture	13
3.1.1	Assembly language view	13
3.1.2	Layer of abstraction	13
3.1.3	Instruction set architecture, ISA	13
3.2	Y86 processor	13
3.2.1	Processor state	13
3.2.2	Instruction encoding	13
3.2.3	Registers	13
3.2.4	Instructions	14
3.3	Sequential implementation	14
3.3.1	Stages	14
3.3.2	Limitations	14
3.4	Pipelined implementation	14

3.4.1	Problems.....	14
3.4.2	Adapting SEQ hardware to PIPE	14
3.4.3	Dynamic nops	14
3.4.4	Data forwarding.....	15
3.4.5	Instruction reordering	15
4	Code Optimization	16
4.1	Optimizing compilers	16
4.1.1	Limitations of compilers	16
4.2	Machine independent optimizations	16
4.2.1	Code motion	16
4.2.2	Reduction in strength	16
4.2.3	Make use of registers	16
4.2.4	Share common sub-expressions.....	16
4.3	Optimization blockers	16
4.4	Machine dependent optimizations	17
4.4.1	Pointer code	17
4.4.2	Loop unrolling.....	17
4.4.3	Parallelism / parallel unrolling.....	17
4.4.4	Branch	17
5	The memory hierarchy.....	18
5.1	Storage technologies.....	18
5.1.1	Random access memory (RAM)	18
5.1.2	Nonvolatile memory	18
5.1.3	Disks.....	18
5.1.4	Locality	19
5.2	Memory hierarchy.....	20
5.2.1	Caches	20
5.3	Linux memory layout.....	20
6	Cache memories	21
6.1	Generic organization of caches	21
6.1.1	Summary of cache parameters	21
6.2	Types of caches	22
6.2.1	Direct mapped cache.....	22
6.2.2	Set associative cache	22
6.2.3	Multi-level caches.....	22
6.3	Cache write policies	22
6.3.1	Bypass cache.....	22
6.3.2	Write-back.....	22
6.3.3	Write-through cache	23
6.4	Cache performance metrics	23
7	Linking.....	24
7.1	Basics.....	24
7.1.1	Translating code into executables.....	24
7.1.2	What does a linker do?.....	24
7.1.3	Why do we need linkers?	24
7.2	Executable and linkable format ELF	24
7.2.1	File format	24
7.3	Relocating symbols and resolving external references	25
7.3.1	Definitions	25
7.3.2	Linker's symbol rules	25
7.4	Static libraries (archives)	25
7.4.1	Linker's algorithm for resolving external references.....	25

7.4.2	Disadvantages	25
7.5	Shared libraries	25
8	Exceptional control flow	26
8.1	Exceptions	26
8.1.1	Asynchronous exceptions (interrupts)	26
8.1.2	Synchronous exceptions.....	26
8.2	Processes and context switching	27
8.2.1	Context switching	27
8.2.2	C and processes	27
8.2.3	Zombies	27
8.3	Signals	28
8.3.1	Concept	28
8.3.2	Sending and receiving	28
8.3.3	Signal implementation	28
8.3.4	Signal handlers	29
8.4	Process groups	29
8.5	Nonlocal jumps.....	29
9	Measuring program performance	30
9.1	Challenge.....	30
9.2	Interval counting	30
9.3	Program profiling	30
9.4	Cycle counters.....	30
10	Virtual Memory.....	31
10.1	Motivation.....	31
10.2	Basics.....	31
10.3	Address translation	31
10.4	Caches and virtual memory.....	31
10.5	Translation lookaside buffer TLB.....	32
10.6	Multi-level page tables.....	32
11	Dynamic memory allocation	33
11.1	Explicit allocation	33
11.1.1	Motivation and goals.....	33
11.1.2	Peak memory utilization.....	33
11.1.3	Fragmentation	33
11.2	Implementation of explicit allocators	34
11.2.1	Issues	34
11.2.2	Implicit lists.....	34
11.2.3	Explicit lists	34
11.2.4	Segregated free list.....	35
11.3	Implicit memory management.....	35
11.3.1	Memory as a graph.....	35
11.3.2	Mark and sweep collecting.....	36

1 Data representation

1.1 Bits and bytes

1.1.1 Basics

- All modern computer systems use **base two** to represent numbers, text, or any other kind of information. The binary systems has some advantages:
 - o Easy to store with bi-stable elements.
 - o Reliably transmittable on noisy and inaccurate wires.
- The smallest piece of information is the **bit**, but generally, information is encoded in **bytes** or groups of bytes. One byte contains 8 bits.
- The whole memory organization is byte-oriented. Programs can refer to what is known as virtual addresses. Conceptually, they are a very large array of bytes, but the implementation is somewhat complicated (see later chapters).
- Every machine has a **word size**, the nominal size of integer-valued data, including addresses. To-day standard personal computers use either 32 or 64 bits, whereby the former limits addresses to 4 GiB.
- Even though the word size of a machine is fixed, multiple data formats are supported. Usually this includes both fractions and multiples of the word size, and is always an integral number of bytes.

1.1.2 Byte ordering

Multiple bytes within multi-byte words can be ordered in different fashions. The two most popular to-day are

- Big endian: Least significant byte has lowest address.
- Little endian: Least significant byte has highest address.

0x100	0x101	0x102	0x103
01	23	45	67
0x100	0x101	0x102	0x103
67	45	23	01

On the right-hand side, one can see the representation of the value 0x1234567 at address 0x100.

1.1.3 Strings in C

Strings in C are represented as character array, each character encoded in ASCII format. All strings are zero-terminated, which means that the final character is 0 (0x00, as opposed to 0x30 for the character "0"). Because the data is organized in single byte quantities, byte ordering is not an issue. Therefore, text-files are generally platform independent, except for different conventions of line termination character(s).

1.1.4 Machine-level code representation

- Programs are encoded as sequence of instructions, each representing a simple operation (arithmetic operation, read/write memory, conditional branch).
- Instructions are encoded as bytes or sequences thereof
 - o Reduced instruction set computer (RISC), like Alpha, Sun, PowerPC
 - o Complex instruction set computer (CISC), most modern personal computers

- There are different instruction types and encodings for different machines, mostly not binary compatible.

1.1.5 C – Bit-wise and logical operations

- Various **bit-level operations** are available in C, and can be applied to any “integral” data type (like long, int, short or char). The arguments are viewed as bit vectors, and the operation is applied bit-wise.
 - `&` bit-wise AND
 - `|` bit-wise OR
 - `^` bit-wise XOR
 - `~` bit-wise NOT
- There are also three **logical operations**, which can be applied to any numeric data type. 0 is viewed as `false`, anything else is take as `true`. The return value of such an operation however is always 0 or 1, and lazy-evaluation applies.
 - `&&` logical AND
 - `||` logical OR
 - `!` logical NOT
- And last, there are two **shift operations**, both left and right. The left shift, `x << y`, shifts `x` by `y` positions to the left, filling the free bits with zeros. For the right shift on the other hand, there are two possible variants: the logical shift (filling with 0's) and the arithmetic shift (replicate the sign bit).

1.2 Integers

1.2.1 Encoding integers

- **Unsigned**

$$\sum_{i=0}^{w-1} x_i \cdot 2^i$$

- **Two's complement**

$$-x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

1.2.2 Numeric ranges

- Unsigned values
 - $UMin = 0$ with representation `000...0`
 - $UMax = 2^w - 1$ with representation `111...1`
- Two's complement values
 - $TMin = -2^{w-1}$, binary `100...0`
 - $TMax = 2^{w-1} - 1$, binary `011...1`
- Note that signed values have an asymmetric range, because $|TMin| = TMax + 1$.

- One can define two mapping $B2U(x)$ and $B2T(x)$ that convert a binary sequence x to its integer representation (either signed or unsigned). For nonnegative values: $B2U(x) = B2T(x)$
- Obviously these mappings are bijective, and thus one can define their inverse $U2B(n)$ and $T2B(n)$.

1.2.3 Casting / unsigned vs. signed

- C allows conversation (both implicit and explicit) from signed to unsigned and vice versa. The result has the same bit representation, just the interpretation changes. The conversation can be written as:

$$T2U(n) = B2U(T2B(n))$$

- In C, by default constants are considered to be signed integers. If one needs an unsigned constant, there is the “U” suffix as in `4294967259U`.
- If expressions are mixed (both signed and unsigned), signed values are implicitly casted to unsigned. This also holds for comparison operators, and thus some surprising results might occur: $0 > -1$, but $0U < -1$.
- Unsigned numbers should not just be used because a variable is non-negative. On some machines, C compiler might generate less efficient code, and it is easy to make some nasty mistakes. On the other hand, when performing modular arithmetic unsigned integers are the right choice. The same holds if the extra bit worth of range is really needed.

1.2.4 Addition

- For any two’s complement integer x it holds: $\sim x + 1 == -x$. This follows from the observation that $x + \sim x = -1 = 111\dots 1$.
- **Unsigned addition** is a function that implements modular arithmetic:

$$s = UAdd_w(u, v) = u + v \bmod 2^w$$

- Unsigned addition forms an **Abelian group**, with the following properties:
 - o closed: $0 \leq UAdd_w(u, v) \leq 2^w - 1$
 - o commutative: $UAdd_w(u, v) = UAdd_w(v, u)$
 - o associative: $UAdd_w(t, UAdd_w(u, v)) = UAdd_w(UAdd_w(t, u), v)$
 - o additive identity: $UAdd_w(u, 0) = u$
 - o additive inverse: $UAdd_w(u, UComp_w(u)) = 0$ with $UComp_w(u) = 2^w - u$
- **Two’s complement addition** has identical bit-level behavior as unsigned addition. The numbers are added, dropping the highest order carry bit and treating what’s left as two’s complement integer.

$$TAdd_w(u, v) = \begin{cases} u + v + 2^w, & u + v < TMin_w \text{ (negative overflow)} \\ u + v, & TMin_w \leq u + v \leq TMax_w \\ u + v - 2^w, & u + v > TMax_w \text{ (positive overflow)} \end{cases}$$

- Two's complement addition is isomorphic to unsigned addition with

$$TAdd_w(u, v) = U2T(UAdd_w(T2U(u), T2U(v)))$$

And forms also a group, with the following properties

- o closed, commutative, associative, 0 is additive identity
- o every element has an additive inverse

$$TComp_w(u) = \begin{cases} -u, & u \neq TMin_w \\ TMin_w, & u = TMin_w \end{cases}$$

1.2.5 Multiplication

- **Unsigned multiplication** behaves modular just like addition:

$$UMult_w(u, v) = u \cdot v \bmod 2^w$$

- In **two's complement multiplication**, the product could require up to $2w$ -bits, but in the result the high order w bits are discarded. And as with addition, signed and unsigned multiplication have the same bit-level behavior.
- Multiplication with a power of 2 can be done with shifts: $u \ll k$ gives $u \cdot 2^k$
- The same can be done for division of unsigned numbers: $u \gg k$ gives $\left\lfloor \frac{u}{2^k} \right\rfloor$
- Division for signed numbers is slightly more complicated, because the same code as for unsigned numbers would give wrong results for negative number. Instead of rounding towards zero, rounding occurs towards $-\infty$. For negative x we would want $\left\lceil \frac{x}{2^k} \right\rceil$, but compute it as $\left\lfloor \frac{x+2^k-1}{2^k} \right\rfloor$, or in C: $(x + (1 \ll k) - 1) \gg k$

1.2.6 Properties of integer arithmetic

- Unsigned multiplication with addition forms a commutative ring with the following properties:
 - o addition is commutative group
 - o closed under multiplication
 - o multiplication is commutative
 - o multiplication is associative
 - o 1 is the multiplicative identity
 - o Multiplication distributes over addition
- Two's complement arithmetic is isomorphic to the unsigned arithmetic.
- While integer arithmetic obeys ordering properties, both unsigned and two's complement arithmetic do not!

1.3 Floating point numbers

- Floating point numbers are stored in the IEEE standard 754.

1.4 Arrays

1.4.1 Basic principle

- $T \ A[L]$: Array of data type T and length L (contiguously allocated region of $L \cdot \text{sizeof}(T)$ bytes)
- C does not check bounds, so out of range behavior is implementation-dependant.

1.4.2 Nested arrays

- Row-major ordering of all elements is guaranteed
- `T A[R][C]`: Array of data type `T`, `R` rows and `C` columns (contiguously allocated region of `R*C*sizeof(T)` bytes)
- Address of `A[i][j]` is `A+(i*C+j)*sizeof(T)`
- Advantages: C compiler handles doubly subscripted arrays, and the generated code is very efficient
- Disadvantages: Only works if you have a fixed array size

1.4.3 Multi-level arrays

- Array of arrays: `type* arr[] = {arr1, arr2, arr3};`
- Accessing an element requires two memory reads, a first one to get the starting address of the row array, and a second to access the element within this row.

1.4.4 Dynamic nested arrays

- Advantages: Can create matrix of arbitrary size.
- Disadvantages: Index computation explicitly, and accessing single elements is costly (multiplication)

1.5 Structures and Unions

1.5.1 Concept of structures

- Contiguously allocated region of memory
- Refer to members within structures by names
- Members may be of different types
- Alignment
 - o Offset within structure must satisfy element's alignment requirement
 - o Overall structure placement: Initial address and structure length must be multiple of `k` bytes, where `k` is the largest alignment of any element.

1.5.2 Unions

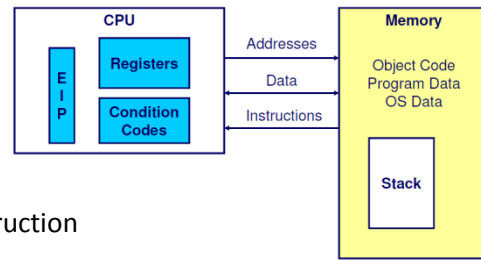
- Overlay union elements and allocate according to largest element

2 Machine-level programming

2.1 Basics

2.1.1 Assembly programmer's view

- Programmer-visible state
 - o EIP (Program counter): Address of next instruction
 - o Register file: Heavily used program data
 - o Condition codes: Store status information about most recent arithmetic operation
 - o Memory: byte addressable array containing code, user data, some OS code, the stack, etc.
- Turning C into object code
 - o C source files (p1.c) are compiled by the compiler (gcc)
 - o Asm files (p1.s) are then assembled by the assembler (as)
 - o Object files (p1.o) are linked with static libraries (.a) by the linker (ld)
 - o The resulting program (p) can then be executed
- Assembly characteristics
 - o Only minimal data types are available: Integer data of 1,2 or 4 bytes (data, pointers) or floating point data of 4, 8 or 10 bytes. No arrays or structures.
 - o Primitive operations: Arithmetic operations, operations to transfer data between memory and registers, transfer control operations.
- Disassembling object code
 - o `objdump -d p`
 - o useful tool for examining object code



2.1.2 Assembler

- Intel/Microsoft format
 - o `lea eax, [ecx+ecx*2]`
 - o `sub esp, 8`
 - o Constants not preceded by \$, denote hex with h at the end
 - o Addressing format shows effective address computation
- GAS/Gnu format
 - o `lea (%ecx,%ecx,2), %eax`
 - o `subl $8, %esp`
 - o Operand size indicated operator suffix

2.2 Instructions

2.2.1 Moving data

- `movl src, dest`
 - o move 4-byte ("long") word
- Operand types
 - o **Immediate**: constant integer data (e.g. `$0x400`, `$-43`)

- **Register:** one of eight integer registers (e.g. %eax, %esp)
- **Memory:** four consecutive bytes of memory, various address modes.
- Important: It is not possible to move data between two memory locations with one instruction. Any other combination is possible.
- Addressing modes
 - $D(Rb, Ri, S) \leftrightarrow Mem[Reg[Rb] + S * Reg[Ri] + D]$
 - D: constant displacement, encoded with 1, 2 or 4 bytes
 - Rb: Base register (any of the eight integer registers)
 - Ri: Index register (any, except for %esp)
 - S: Scale, only 1, 2, 4 or 8

2.2.2 Address computation instruction

- `leal src, dest`
 - `src` is address mode expression
 - Sets `dest` to address denoted by expression `src`
- Uses
 - Computing addresses without doing memory references (pointer arithmetic)
 - Computing simple arithmetic expressions

2.2.3 Some arithmetic operations

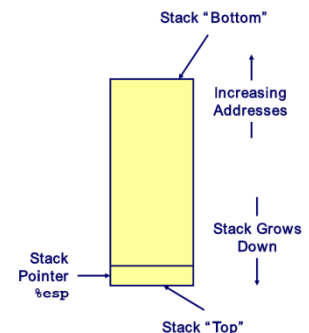
- `addl, subl, imull, sall, sarl, shrl, xorl, andl, orl`
- `incl, decl, negl, notl`

2.2.4 Condition codes

- Single bit registers
 - CF: carry flag
 - ZF: zero flag
 - SF: sign flag
 - OF: overflow flag
- Implicitly set by arithmetic operations like `addl`, not set by `leal` however.
- Explicitly set by compare instruction `cmpl` or `testl`.
- Reading condition codes is possible with `setX` instructions.

2.3 IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register %esp indicates lowest stack address, i.e. the address of the top element



2.3.1 Push/pop

- `pushl src`
 - Fetch operand at `src`, decrement %esp by 4 and write operand to the address given by %esp.

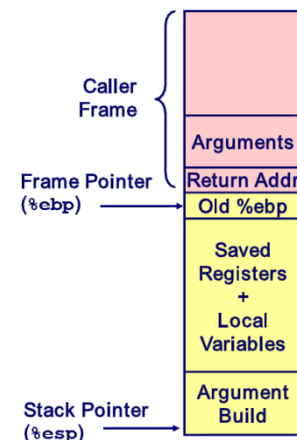
- `popl dest`
 - o Read operand at address given by `%esp`, increment `%esp` by 4 and write to `dest`.

2.3.2 Procedure control flow

- Procedure call: `call label`
 - o Push return address (address of the instruction beyond `call`) on stack and jump to `label`.
- Procedure return: `ret`
 - o Pop address from stack and jump to that address.
- Stack-based languages (such as C, Java) support recursion. The state for a given procedure must be stored somewhere (arguments, local variables, return pointer).
- Stack is allocated in **frames**, containing local variables, return information, and some temporary space.
- These frames are allocated in a so-called “set-up” code when entering a procedure and deallocated at the end.
- Two special pointers help stack management:
 - o Stack pointer `%esp` indicates stack top
 - o Frame pointer `%ebp` indicates start of current frame

2.3.3 IA32/Linux stack frame

- Current stack frame (“top” to bottom)
 - o Parameters for function about to call
 - o Local variables
 - o Saved register context
 - o Old frame pointer
- Caller stack frame
 - o Return address (pushed by call-instruction)
 - o Arguments for this call



2.3.4 Register saving conventions

- **Caller save** (`%eax`, `%edx`, `%ecx`)
 - o Caller saves temporary in its frame before calling (callee can freely use this registers)
- **Callee save** (`%ebx`, `%esi`, `%edi`)
 - o Callee saves temporary in its frame before using (caller is guaranteed that contents stay the same after a procedure call as before)
- Special: `%ebp` (base pointer), `%esp` (stack pointer) and `%eax` (return value).

3 Sequential processors

3.1 Instruction set architecture

3.1.1 Assembly language view

- Processor state (registers, memory, ..)
- Instructions and how they are encoded

3.1.2 Layer of abstraction

- Above: How to program machine (processor executes instructions in a sequence)
- Below: What needs to be built (various speedup tricks, like parallel execution)

3.1.3 Instruction set architecture, ISA

- Contract between software side (compilers, assemblers) and processor architects

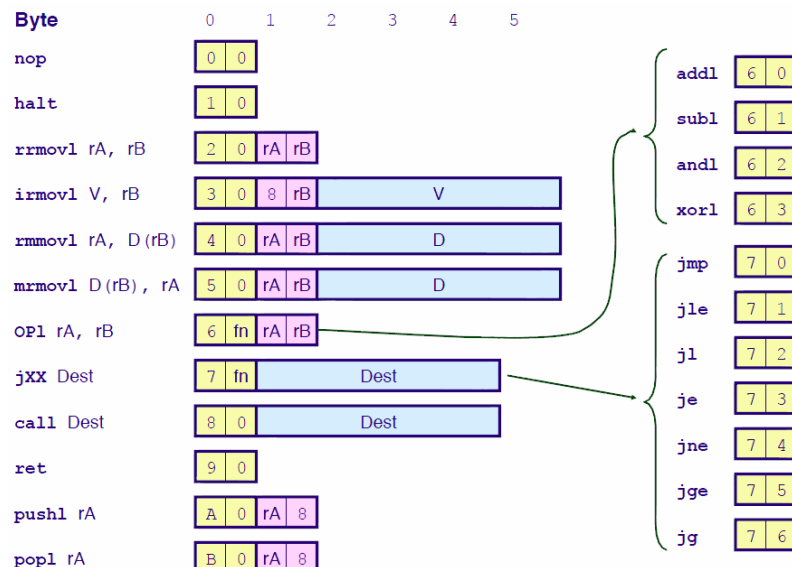
3.2 Y86 processor

3.2.1 Processor state

- Program registers: same 8 registers as with IA32, each 32 bits
- Condition codes: OF, CF, ZF, SF
- Program counter: indicates address of next instruction
- Memory: byte-addressable storage array, little-endian

3.2.2 Instruction encoding

- 1 to 6 bytes of information, can determine length of instruction from first byte.



3.2.3 Registers

- Each register has a 4-bit ID (from 0 to 7: eax, ecx, edx, ebx, esi, edi, esp, ebp; ID 8 is used to denote "no register")

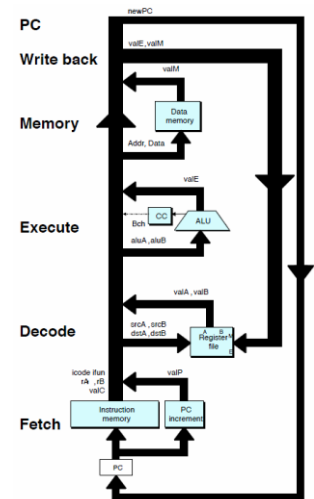
3.2.4 Instructions

- Arithmetic operations: `addl, subl, andl, xorl`
- Move operations: `rrmovl, irmovl, rmmovl, mrmovl`
- Jumps: `jmp, jle, jl, je, jne, jge, jg`
- Stack operations: `pushl, popl`
- Subroutine calls: `call, ret`
- Miscellaneous: `nop, halt`

3.3 Sequential implementation

3.3.1 Stages

- Fetch: Read instruction from memory
- Decode: Read program registers
- Execute: Compute value or address
- Memory: Read or write data
- Write Back: Write program registers
- PC: Update program counter



3.3.2 Limitations

- Too slow to be practical
- Hardware units only active for a fraction of clock cycle (much idle time)

3.4 Pipelined implementation

3.4.1 Problems

- Register overhead: The deeper the pipeline, the bigger the overhead of loading registers
- Data dependencies: Operations may depend on result of preceding ones
- Predictions may be wrong

3.4.2 Adapting SEQ hardware to PIPE

- Stages in SEQ occur in sequence, one at a time
- Adding pipeline register: Between two stages registers are needed to hold intermediate values
- All values pass through all stages, even when not explicitly needed
- Feedback paths: Branch information (jump taken or not, target address), etc.
- Safe prediction strategy
 - o Instructions that don't transfer control (next PC is `valP`)
 - o Call and unconditional jumps (next PC is `valC`)
 - o Conditional jumps/return instructions: don't try to predict

3.4.3 Dynamic nops

- If instructions follow too closely after one that writes registers, dynamically insert nops. This nops are called bubbles. The instruction that causes this is called stalling instruction.
 - o Stalling instruction is held back in decode stage
 - o Following instruction stays in fetch state

- Bubble injected into execute stage

3.4.4 Data forwarding

- Registers aren't written until completion of write-back stage, but source operands read from register file in decode stage.
- Trick: Pass values directly from generating instruction to decode stage (needs to be available at end of decode stage)
- Forwarding sources: valE or valM

3.4.5 Instruction reordering

- Reordering instructions can help reduce data dependencies and therefore lead to speed-up.

4 Code Optimization

4.1 Optimizing compilers

- Provide efficient mapping of program to machine: Register allocation, code selection and reordering, eliminating minor inefficiencies.
- Don't usually improve asymptotic efficiency.
- Have difficulties overcoming so-called "optimization blockers" (e.g. potential memory aliasing or procedure side-effects).

4.1.1 Limitations of compilers

- Operate under fundamental constraints
 - o Must not cause any change in the program behavior under any possible condition.
 - o Often prevents it from making optimizations that would only affect behavior under pathological condition.
- Most analysis is performed only within procedures; whole program analysis is too expensive in most cases.
- Most analysis is based only on static information; difficult to anticipate run-time inputs.
- When in doubt, compilers must be conservative.

4.2 Machine independent optimizations

4.2.1 Code motion

- Move frequently executed code that always has the same result (e.g. moving code outside of loops).
- Compilers do a good job with array code and simple loop structures.

4.2.2 Reduction in strength

- Replace costly operations with simpler ones (e.g. shift instead of multiplication, or instead of $i*n$ in a loop `tmp += n`)

4.2.3 Make use of registers

- Reading and writing registers is much faster than reading or writing of memory.
- Limitation: Compilers are not always able to determine whether variable can be held in register, e.g. with the possibility of aliasing.

4.2.4 Share common sub-expressions

- Reuse portions of expressions.
- Compilers are often not very sophisticated in exploiting arithmetic properties.

4.3 Optimization blockers

- Procedure calls
 - o Procedures may have side effects (alter global state)
 - o Function may not return the same value each time for given arguments (depends on other parts of the global state)

- Linker may overload with different version (compiler cannot even look at most procedures)
- Warning: Compiler treat procedure calls as black boxes, so they are weak in optimizing in and around them.
- Aliasing: Two different memory references may specify a single location.
 - Easy to happen in C, since address arithmetic is allowed.
 - Get in habit of introducing local variables (your way of telling the compiler that there is no aliasing)

4.4 Machine dependent optimizations

4.4.1 Pointer code

- Sometimes it is faster to use pointer code.

4.4.2 Loop unrolling

- Combine multiple iterations into a single loop body
- Amortizes loop overhead across multiple iterations
- Finish extras at the end

4.4.3 Parallelism / parallel unrolling

- Accumulate in multiple variables, and accumulate in the end.
- Note: Combining operation must be associative and commutative.
- This uses the pipelining of some functional units and removes data dependencies
- Limitations:
 - Need lots of registers, only 6 usable integer registers are available, and they are also needed for pointers, loop conditions, etc.

4.4.4 Branch

- Branches are expensive. Misprediction on a Pentium III costs about 14 cycles.
- Best to avoid at all, e.g. by using conditional moves.

5 The memory hierarchy

5.1 Storage technologies

5.1.1 Random access memory (RAM)

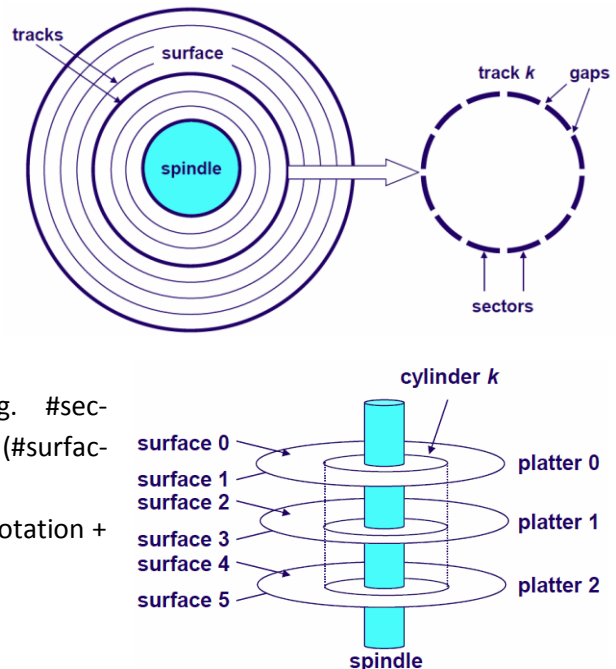
- The basic storage unit is a **cell** (one bit), and RAM is packaged as a chip. Multiple chips form a memory.
- Static RAM (SRAM)
 - o Each cell is build by a six-transistor circuit
 - o Retains value as long as its powered
 - o Relatively insensitive to disturbances such as electrical noise
 - o Very fast, very expensive
- Dynamic RAM (DRAM)
 - o Each cell stores bit with a capacitor and a transistor
 - o Values must be refreshed every 10-100 ms
 - o Sensitive to disturbances
 - o Slower, but a lot cheaper
- Conventional DRAM Organization
 - o $d \times w$ DRAM: dw total bits organized as d supercells of w bits
 - o Read: Row access strobe (RAS) selects a row, and this row is copied from DRAM to a row buffer. Then the column access strobe (CAS) selects a column, and the corresponding supercell is copied from the buffer to the data line.

5.1.2 Nonvolatile memory

- Volatile memory (like DRAM and SRAM) loses its information if powered off.
- Generic name for nonvolatile memory is read-only memory (ROM).

5.1.3 Disks

- Disks consist of **platters**, each with two **surfaces**. Each surface consists of concentric rings called **tracks**. Each track consists of **sectors** separated by **gaps**.
- Multiple tracks on different platters build a **cylinder**.
- The capacity is the maximum number of bits that can be stored.
 - o $\text{Capacity} = (\text{\#bytes/sector}) \times (\text{avg. \#sectors/track}) \times (\text{\#tracks/surface}) \times (\text{\#surfaces/platter}) \times (\text{\#platters/disk})$
- Disk access time: $T_{\text{access}} = T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}}$

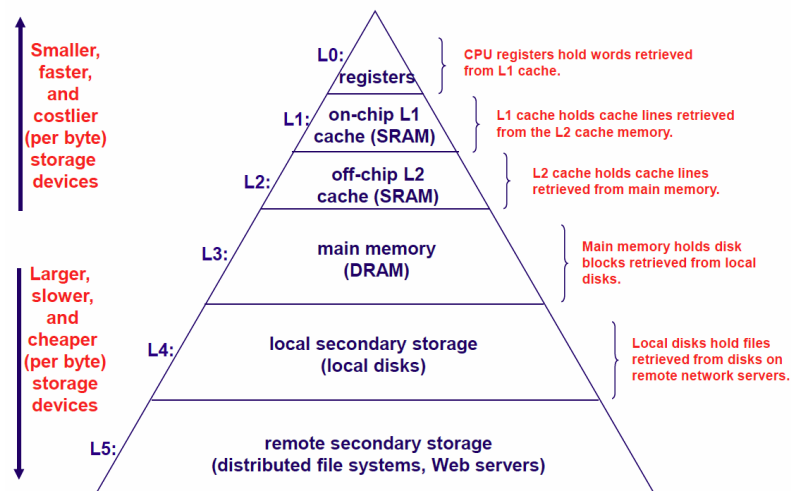


- Seek time: Time to position heads over cylinder containing target sector (e.g. 9ms)
- Rotational latency: Time waiting for first bit of target sector to pass under r/w head
- Transfer time: Time to read the bits in the target sector
- Important: First bit in a sector is the most expensive, because the access time is dominated by the seek time and the rotational latency.
- Logical disk blocks
 - Modern disks present a simpler abstract view of the complex sector geometry. The set of available sectors is modeled as a sequence of b-sized logical blocks
 - The mapping between logical blocks and the actual (physical) sectors is maintained by the hardware/firmware
 - Allows controller to set aside some spare cylinders for each zone.
- Reading a disk sector
 - CPU initiates a disk read by writing a command, logical block number and a destination memory address to a port (address) associated with the disk controller.
 - Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.
 - When the DMA transfer completes, the disk controller notifies the CPU with an interrupt.

5.1.4 Locality

- Programs tend to reuse data and instructions they have used recently, or that were recently referenced themselves
 - **Temporal locality:** recently referenced items are likely to be referenced in the near future.
 - **Spatial locality:** items with nearby addresses tend to be referenced close together in time.

5.2 Memory hierarchy

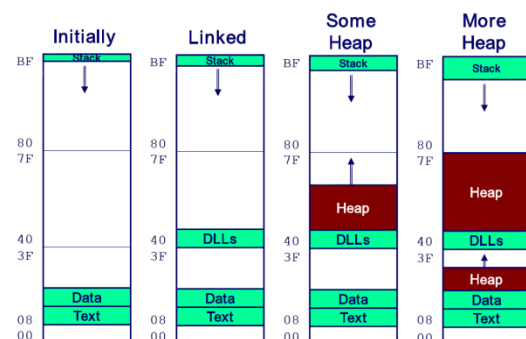


5.2.1 Caches

- A smaller but faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Data between levels is copied in block-sized transfer units. If a program needs an object d , which is stored in some block b , there are two possible cases:
 - o Cache hit: Program finds b in the cache at level k .
 - o Cache miss: b is not at level k , so the level k cache must fetch it from level $k+1$.
- Types of cache misses:
 - o Cold (compulsory) miss: Occurs because the cache is empty
 - o Conflict miss: Most caches limit blocks at level $k+1$ to a small subset of block positions at level k . Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block
 - o Capacity miss: Occurs when the set of active cache blocks (working set) is larger than the cache.

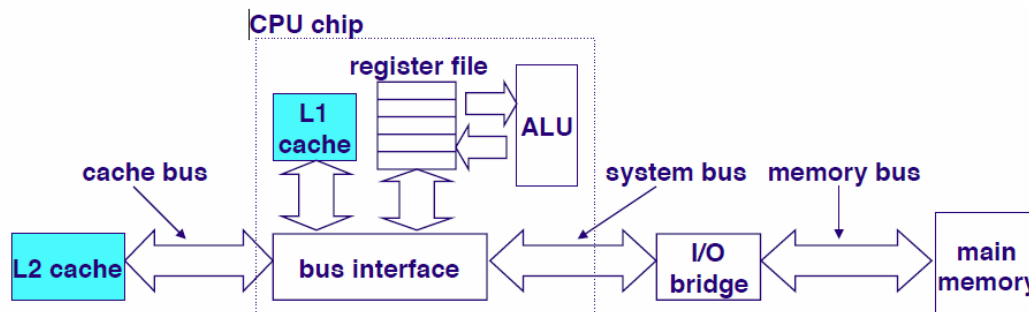
5.3 Linux memory layout

- Stack: Runtime stack (8MB Limit)
- Heap: Dynamically allocated storage (malloc, etc)
- DLLs: Dynamically linked libraries (e.g. printf)
- Data: Statically allocated data (e.g. strings in code)
- Text: Executable machine instruction, read-only



6 Cache memories

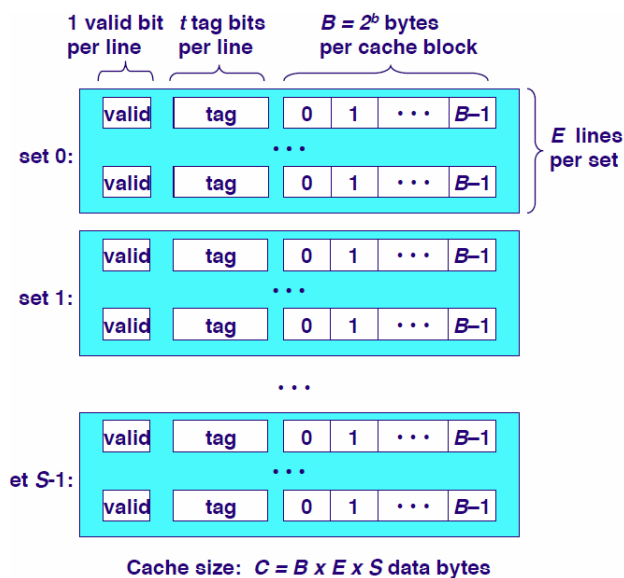
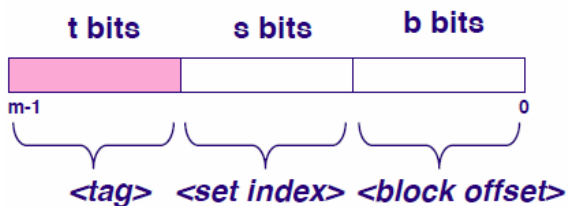
The typical bus structure looks as follows:



6.1 Generic organization of caches

- A cache is an array of **sets**
- Each set contains one or more **lines**.
- Each line holds a **block of data**.
- The word at address A is in the cache, if the tag bits in one of the valid lines in set <set index> match <tag>.

Address A:



6.1.1 Summary of cache parameters

Fundamental parameters	
Parameter	Description
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size in bytes
$m = \log_2 M$	Number of physical (main memory) address bits

Derived quantities	
Parameter	Description
$M = 2^m$	Maximum number of unique memory addresses
$s = \log_2 S$	Number of set index bits
$b = \log_2 B$	Number of block offset bits
$t = m - (s+b)$	Number of tag bits
$C = B \times E \times S$	Cache size in bytes not including overhead

6.2 Types of caches

6.2.1 Direct mapped cache

- Simplest cache: Exactly one line per set.
- Access
 - o Set selection: Use the set index bits to determine the set of interest
 - o Line matching: Check the valid and tag bits
 - o Word selection: Then extract the word

6.2.2 Set associative cache

- Multiple lines per set
- Access
 - o Set selection: Use the set index bits to determine the set of interest
 - o Line matching: Check the tag bits in each valid line
 - o Word selection: Then extract the word

6.2.3 Multi-level caches

- There are two options to organize caches: There is the option to have separate caches for data and instructions, or for one, unified cache.

6.3 Cache write policies

The trend goes towards write-back caches; though they are more complicated, overall they are better.

6.3.1 Bypass cache

- If a requested block is not at level k (i.e. a miss), the data can be written directly to level k+1.
- Hurts spatial locality, because the next read misses.

6.3.2 Write-back

- If a requested block is not at level k (i.e. a miss), the cache fetches the block from level k+1, and writes the data (may have to replace another block).
- A write operation to level k may incur two transfers between level k and k+1
 - o Old (dirty) line must be saved
 - o Block that contains the data item must be read

6.3.3 Write-through cache

- In order to avoid dirty lines, it is possible to keep level $k+1$ always up to date: A write to level k is passed on to level $k+1$ and level k keeps a copy).
- Now each write must wait for the transfer to level $k+1$ to complete.

6.4 Cache performance metrics

- **Miss rate:** Fraction of memory references not found in the cache
- **Hit time:** Time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
- **Miss penalty:** Additional time required because of a miss

7 Linking

7.1 Basics

7.1.1 Translating code into executables

- A **compiler driver** such as gcc coordinates all steps in the translation and linking process. It invokes the preprocessor (cpp), the compiler (cc1), the assembler (as) and the linker (ld).

7.1.2 What does a linker do?

- A linker merges multiple relocatable (.o) object files into a single executable object file that can be loaded and executed by the loader
- To do this, the linker resolves external references (references to symbols that are defined in other object files): It relocates the symbols from their relative locations in the .o files to the new absolute positions in the executables and updates all references to reflect their positions

7.1.3 Why do we need linkers?

- Modularity
 - o Programs can be written as a collection of smaller source files, rather than a monolithic mass.
 - o Can build libraries of common functions (e.g. math libraries, standard C library)
- Efficiency
 - o Time: Only recompile changed files.
 - o Space: Libraries of common functions can be aggregated into a single file, and yet executables contain only code for the functions they actually use.

7.2 Executable and linkable format ELF

- Standard binary format for object files. One unified format for
 - o Relocatable object files .o
 - o Executable object files
 - o Shared object files .so

7.2.1 File format

- Elf header (Magic number, type (.o, exec, .so), machine, byte ordering, etc)
- Program header table: Page size, virtual addresses memory segments, segment sizes, etc.
- .text section: Code
- .data section: Initialized (static) data
- .bss section: Uninitialized (static) data. Has a section header, but occupies no space
- .symtab section: Symbol table with procedure and static variable names
- .rel.text section: Relocation info for .text section (Addresses of instruc-

ELF header
Program header table (required for executables)
.text section
.data section
.bss section
.symtab
.rel.text
.rel.data
.debug
Section header table (required for relocatables)

tions that will need to be modified in the executable and instructions on how to do so)

- .rel.data, .debug, .rodata, .line, .strtab, etc sections

7.3 Relocating symbols and resolving external references

7.3.1 Definitions

- **Symbols** are lexical entities that name functions and variables.
- Each symbol has a **value** (typically a memory address).
- Code consists of symbol **definitions** and **references**.
- References can be either **local** or **external**.
- Program symbols are either strong or weak:
 - o **strong**: procedures and initialized globals
 - o **weak**: uninitialized globals

7.3.2 Linker's symbol rules

1. A strong symbol can only appear once.
2. A weak symbol can be overridden by a strong symbol of the same name. That is, references to weak symbols resolve to the strong symbol.
3. If there are multiple weak symbols, the linker can pick an arbitrary one.

7.4 Static libraries (archives)

- Commonly used functions (like math, I/O, memory management) can be put in so-called **static libraries**. The linker tries to resolve unresolved external references by looking for the symbol in one or more archives. The linker selectively includes only the .o files in the archive that are actually needed.
- **Archivers** allow you to create static libraries, build from multiple object files.

7.4.1 Linker's algorithm for resolving external references

- Scan .o files and .a files in the command line order. During the scan, keep a list of currently unresolved references. As each new .o or .a object file is encountered, try to resolve each unresolved reference in the list against the symbols in the object file.
- In any entries in the unresolved list are still left at the end of the scan, throw an error.
- Problem: Command line order matters.

7.4.2 Disadvantages

- Potential duplicating lots of common code in executable files on a file system.
- Potential for duplicating lots of code in the virtual memory space of many processes
- Minor bug fixes of system libraries require each application to explicitly relink.

7.5 Shared libraries

- Dynamic link libraries, DLLs
- Dynamic linking can occur when executable is first loaded and run, or also after the program has begun.
- Shared libraries routines can be shared by multiple processes.

8 Exceptional control flow

- Mechanisms for exceptional control flow exists at all levels of a computer system
- Low level mechanism
 - o Exceptions: Change in control flow in response to a system event
 - o Combination of hardware and OS software
- Higher level mechanism
 - o Process context switch, signals, nonlocal jumps
 - o Implemented by OS software or C language runtime library (nonlocal jumps)

8.1 Exceptions

- An exception is a transfer of control to the OS in response to some event (i.e. change in the processor state)
- Each type of event has a unique exception number k , which is an index into a jump table, the so-called **interrupt vector**. Jump table entry k point to a function (**exception handler**) and is called each time exception k occurs.

8.1.1 Asynchronous exceptions (interrupts)

- Caused by events external to the processor and are indicated by setting a processor's interrupt pin.
- Handler return to "next" instruction.
- Examples
 - o I/O interrupts: hitting ctrl-c, arrival of a packet from a network or of a data sector from disk
 - o Hard reset interrupt: hitting the reset button
 - o Soft reset interrupt: hitting ctrl-alt-delete

8.1.2 Synchronous exceptions

- Caused by events that occur as a result of executing an instruction
- Traps
 - o Intentional
 - o e.g. system calls, breakpoint traps
 - o returns control to "next" instruction
- Faults
 - o Unintentional but possibly recoverable
 - o e.g. page faults (recoverable), protection faults (unrecoverable)
 - o Either re-execute faulting ("current") instruction or aborts
- Aborts
 - o Unintentional and unrecoverable
 - o e.g. parity error, machine check
 - o Aborts current program

8.2 Processes and context switching

- A process is an instance of a running program and provides the program with two key abstractions:
 - o Logical control flow: Each program seems to have exclusive use of the CPU.
 - o Private address space: Each program seems to have exclusive use of main memory
- The state consists of the memory image, the register values and the program counter

8.2.1 Context switching

- Processes are managed by a shared chunk of OS code, called the **kernel**. The kernel however is NOT a separate process, but rather runs as part of some user process
- Control flow from one process to another happens via a **context switch**.

8.2.2 C and processes

- `int fork(void)`
 - o Creates a new process (child process) that is identical to the calling process (parent)
 - o Returns 0 to the child and the child's pid to the parent
 - o The parent and child both run the same code and start with the same state, but each has its own copy
- `void exit(int status)`
 - o Exits a process with exit code status (normally 0).
 - o `atexit(f)` can be used to register functions to be executed upon exit
- `int wait(int *child_status)`
 - o Suspends the current process until one of its children terminates
 - o Return value is pid of the child process and `child_status` will point to some status information
 - o If multiple children completed, they will be taken in an arbitrary order
- `waitpid(pid, &status, options)`
 - o Waits for a specific process
- `int execl(char *path, char *arg0, char* arg1, ..)`
 - o Loads and runs executable at path with args `arg1,arg2,...` `arg0` will be the name of the process. Return -1 on error, and doesn't return otherwise.

8.2.3 Zombies

- When a process terminates, it still consumes some resources (various tables are maintained by the OS)
- **Reaping**
 - o Performed by the parent on terminated child
 - o Parent is given the exit status information and the kernel discards the process
 - o If a parent doesn't reap a child, it will be reaped by the init process. Thus, explicit reaping is only needed for long-running processes.

8.3 Signals

8.3.1 Concept

- A signal is a small message that notifies a process that an event of some type has occurred in the system
 - o Sent from the kernel (sometimes at the request of another process) to a process
 - o Different signals are identified by a small integer ID
 - o The only information in a signal is its ID and the fact that it arrived.

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt from keyboard (ctrl-c)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

8.3.2 Sending and receiving

- Signals are **sent** by the kernel to a destination process by updating some state in the context of the destination process.
- The kernel sends a signal for one of the following **reasons**:
 - o The kernel has detected a system event such as divide-by-zero or the termination of a child.
 - o Another process has invoked the kill system call to explicitly request the kernel to send a signal to the destination process
- **Receiving** a signal. A destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal:
 - o Ignore the signal
 - o Terminate the process
 - o Catch the signal by executing a user-level function called a **signal handler**.
- A signal is **pending** if it has been sent, but not yet received. Note that there can be at most one pending signal of any particular type. Signals are NOT queued, if a process has a pending signal of type k, any subsequent signals of the same type are discarded.
- A process can **block** the receipt of certain signals. Blocked signals can be delivered, but will not be received until the signal is unblocked.
- A pending signal is received at most once.

8.3.3 Signal implementation

- The kernel maintains `pending` and `blocked` bit vectors in the context of each process.
- Before the kernel passes control to a process (after returning from an exception handler), it computes `pnb = pending & ~blocked`, and passes control to the next instruction of p if `pnb == 0`. Otherwise, it chooses the least nonzero bit k and forces process p to receive the signal k. This is repeated for all nonzero k in pnb.

8.3.4 Signal handlers

- Every signal type has a predefined **default action**, one of:
 - o The process terminates
 - o The process terminates and dumps core
 - o The process stops until restarted by a SIGCONT signal
 - o The process ignores the signal
- The signal function modifies the default associated with the receipt of signal `signum`:
`handler_t* signal(int signum, handler_t *handler)`
 - o `handler` can be either `SIG_IGN` (ignore this signal), `SIG_DFL` (restore default action) or an address of a signal handler.

8.4 Process groups

- Every process belongs to exactly one process group.
- `getpgrp` and `setpgid` can be used to view and change the process group of a process.

8.5 Nonlocal jumps

- Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location.
 - o Useful for error recovery and signal handling
- `int setjmp(jmp_buf jb)`
 - o Must be called before `longjmp` and identifies a return point for a subsequent `longjmp`.
 - o Called once, and returns one or more times
 - o Implementation: Remember where you are by storing the current register context, stack pointer, and PC value in the jump buffer `jb`.
 - o Returns 0 the first time.
- `void longjmp(jmp_buf jb, int i)`
 - o Return from the `setjmp` remembered by the jump buffer `jb` again, and return `i` this time.
 - o Called once, but never returns.
 - o Implementation: Restore register context from the jump buffer, set `%eax` (return value) to `i` and jump to the location indicated by the PC stored in the jump buffer.
- Limitations: Works within stack discipline, so only long jumps to environments of functions that have been called but not yet completed are possible.

9 Measuring program performance

9.1 Challenge

- How much time does program X require?
 - o CPU time: How many total seconds are used when executing X? Used for most applications, with a small dependence on other system activities.
 - o Wall clock time: How many seconds elapse between the start and completion of X? Depends on system load, I/O times, etc.
- “Time” on a computer system for a specific process can be divided into three categories:
 - o User time: Time executing instructions in the user process.
 - o System time: Time executing instructions in kernel on behalf of the user process.
 - o Some other user’s time.

9.2 Interval counting

- This technique is used by the OS to measure runtimes using the interval timer: Two counts per process are maintained, one for the user time and one for the system time.
- Each time there is a timer interrupt, increment the counter for the executing process (user time if running in user mode, and system time if running in kernel mode).
- Accuracy
 - o If the timer interval is α , a single process segment measurement can be off by $\pm \alpha$. Therefore it is not possible to give an error bound, as the actual time could be consistently over- or underestimated.
 - o In average, over/underestimates tend to balance out if the run time is sufficiently large.

9.3 Program profiling

- In unix, there is `prof` and `gprof` to get detailed information about runtimes (per function).

9.4 Cycle counters

- Most modern systems have build in registers that are incremented every clock cycle. They are very fine grained and are maintained as part of the process state.
- Special assembly instruction to access this counter.
- Measuring with cycle counter: Get the current value of the cycle counter, compute something and get the new value.

10 Virtual Memory

10.1 Motivation

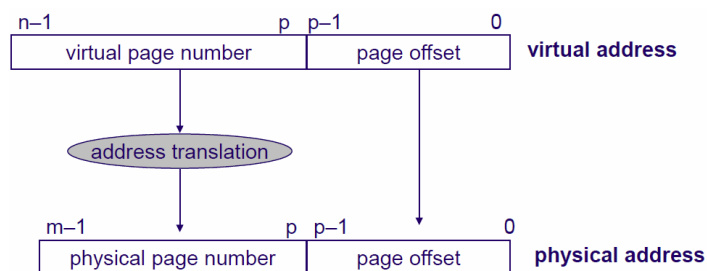
- Use physical DRAM as a cache for the disk: Address space of a process or the sum of address spaces of multiple processes can exceed physical memory size.
- Simplify memory management: Multiple processes are resident in main memory, each with its own address space.
- Provide protection: One process can't interfere with another, and user processes cannot access privileged information

10.2 Basics

- On a system with virtual memory, the CPU only generates virtual addresses, which then are translated by the hardware to physical addresses via an OS-managed lookup table (**page table**).
- The page table is organized in pages, whereby each table has an entry, even if it is not currently in memory.
- If an object is on disk rather than in memory, a so-called **page fault** occurs. The OS exception handler is invoked and moves the data into memory. A page fault triggers a series of actions:
 - o Processor signals the controller: Read block of length p starting at disk address X and store the information at memory address Y .
 - o Read occurs: Direct memory access (DMA), done by the I/O controller
 - o I/O controller signal completion: Interrupt processor, OS resumes suspended process

10.3 Address translation

- Parameters:
 - o $P = 2^p$ page size (bytes)
 - o $N = 2^n$ virtual address limit
 - o $M = 2^m$ physical address limit
- The page offset bits do not change as a result of the address translation.
- Each process has its own (set of) page table(s), and the VPN forms an index into the page table (points to a page table entry).
- The page table entry PTE provides information about the page:
 - o Valid bit: If true, the page is in memory, otherwise on disk
 - o Access rights

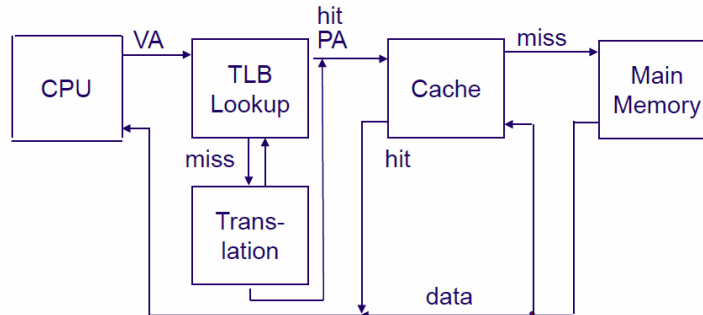


10.4 Caches and virtual memory

- Most caches are "physically addressed", that is, accessed by physical addresses
 - o This allows multiple processes to have blocks in the cache at the same time and to share pages
 - o Caches don't need to be concerned with protection issues.
- Address translation is preformed before cache lookup. But this could also involve a memory access itself (of the PTE). Of course, page table entries also become cached.

10.5 Translation lookaside buffer TLB

- Small hardware cache in the MMU (memory management unit) that maps virtual page numbers to physical page numbers
- Contains complete page table entries for a small number of pages



10.6 Multi-level page tables

- The page table on modern systems would require huge amounts of memory itself. A common solution is multi-level page tables, e.g. a two-level table: The first table has 1024 entries, each of which points to a level-two page table. In the second level page table are again 1024 entries, each of which points to a page.

11 Dynamic memory allocation

11.1 Explicit allocation

11.1.1 Motivation and goals

- Constraints on application
 - o Can issue arbitrary sequence of allocation and free requests
 - o Free requests must correspond to an allocated block
- Constraints on allocators
 - o Can't control number or size of allocated blocks
 - o Must immediately respond to all allocation request (i.e. no reordering or buffering)
 - o Must allocate blocks from free memory (i.e. can place allocated blocks only in free memory)
 - o Must align blocks so they satisfy all alignment requirements (e.g. 8 byte alignment)
 - o Can manipulate and modify only free memory
 - o Can't move allocated blocks once they are allocation (i.e. no compaction)
- Primary goals
 - o Good time performance for `malloc` and `free` (ideally constant time, though not always possible)
 - o Good space utilization, minimize fragmentation
- Other goals
 - o Good locality properties: Structures allocated close in time should be close in space
 - o Robust: Can check that `free(p)` is on a valid object `p`, and can check that a memory reference is to an allocated space.
- Performance goals
 - o Maximize throughput (number of completed request per unit time)
 - o Maximize peak memory utilization

11.1.2 Peak memory utilization

- Given some sequence of malloc and free request R_0, R_1, \dots, R_{n-1}
- `malloc(p)` results in a block with a **payload** of `p` bytes.
- The **aggregate payload** P_k is the sum of currently allocated payloads
- The current heap size is denoted by H_k , and is assumed monotonically nondecreasing.
- The peak memory utilization is defined as $U_k = \frac{\max_{i \leq k} P_i}{H_k}$

11.1.3 Fragmentation

- Poor memory utilization is caused by fragmentation, coming in two forms: internal and external.
- **Internal fragmentation**
 - o For some block, the internal fragmentation is the difference between the block size and the payload size and is caused by overhead of maintaining the heap data structures.
 - o Depends only on the pattern of previous request, and thus is easy to measure
- **External fragmentation**

- Occurs when there is enough aggregate heap memory, but no single free block is large enough.
- External fragmentation depends on the pattern of future requests, and thus is difficult to measure.

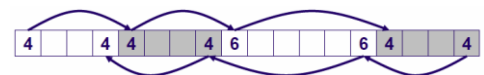
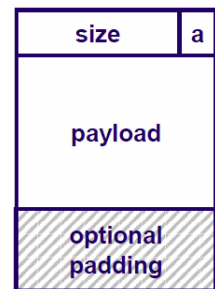
11.2 Implementation of explicit allocators

11.2.1 Issues

- How do we know how much memory to free?
 - The standard method is to keep the length of a block in the word preceding the block. This block is often called the header field
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a structure that is smaller than the free block?
- How do we pick a block to use for allocation? Many might fit.

11.2.2 Implicit lists

- Blocks are preceded by a header that stores the size (always a multiple of two), and one bit of information about whether the block is allocated or free.
- There are several possibilities on how to find a free block:
 - First fit: Search the list from the beginning and choose the first block that fits. Can take linear time in the total number of blocks, and in practice, it can cause “splinters” at the beginning of the list
 - Next fit: Like first fit, but the search begins from the location of the end of the previous search. Research suggests that fragmentation is even worse.
 - Best fit: Search the whole list, and choose the free block with the closest size that fits. This keeps fragmentation small, but will typically run slower than first fit.
- **Splitting:** Since the allocated space might be smaller than the free space, we might want to split the free block. But when freeing this block again, we need to check its neighbors to see, if they are free as well.
- **Join/coalesce** is done when freeing a block, and at least one of its neighbors is free, too. To join with the previous block, we need a way to traverse the list backwards. To that end, we replicate the size/allocated word at the bottom of the block (-> boundary tags).
- Summary
 - Not used in practice, allocating (worst case linear) is too slow.
 - Boundary tags and coalescing however are general concepts.



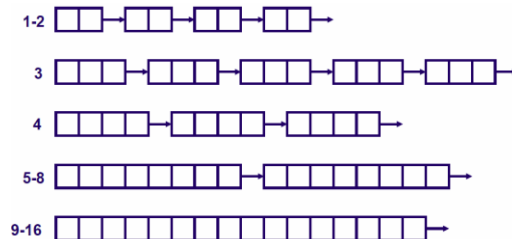
11.2.3 Explicit lists

- This approach uses the free space of free block for link pointers (typically doubly linked). The boundary tags however are still needed for coalescing. Note that the links are not necessarily in the same order as the blocks.

- Allocation: With this linked list, allocation is straight forward. The free block is split and the remaining free part will be in the list now.
- Freeing: Where in the free list do you put a newly freed block?
 - o LIFO: Insert the block at the beginning of the free list. This is simple and can be done in constant time, but studies suggest fragmentation is worse than with address ordered policy.
 - o Address ordered: Insert block so that free list blocks are always in address order, i.e. enforce $\text{addr}(\text{pred}) < \text{addr}(\text{curr}) < \text{addr}(\text{succ})$. This requires a search, but the fragmentation is better.
- Summary: Still linear time, but only in the number of free block instead of the total number of blocks.

11.2.4 Segregated free list

- There are different size classes, each with its own collection of blocks. Often there are classes for each small size, and for larger sizes a class for each power of 2.



- Simple segregate storage: Separate heap and free list for each size class, and no splitting.
 - o Allocate: If free list for size n is not empty, allocate the first block in the list (explicit or implicit). Otherwise get a new page, and create free list. Constant time.
 - o Free: Add the block to the free list again, and if the page is empty, return the page for use by another size (optional).
 - o Fast, but can fragment badly
- Segregated fits: Array of free lists, each one for some size class
 - o Allocate: Search appropriate free list for block size $m \geq n$. Split the block, and place the fragment in an appropriate list (optional). If no block is found, try next larger class.
 - o Free: Coalesce and place in appropriate list.
 - o Faster search than sequential fits.

11.3 Implicit memory management

- The **garbage collector** automatically reclaims heap-allocated storage; the application never has to free.
- Common in functional languages, scripting languages and modern object-oriented languages.

11.3.1 Memory as a graph

- Each block is a node in the graph, and each pointer is an edge.
- Locations not in the head that contain pointers into the heap are called root nodes (e.g. registers, stack, global variables)

- A node is **reachable** if there is a path from any root node to that node. Unreachable nodes are **garbage** (not used anymore).

11.3.2 Mark and sweep collecting

- Can build on top of malloc/free package, and allocate until you run out of space.
- Use extra mark bit in the header of each block, and do the following when running out of space:
 - o Mark: Start at root nodes and set mark bit on all reachable memory
 - o Sweep: Scan all blocks and free blocks that are not marked