Systems Programming

Some preliminary facts about C

- file.c
- C preprocessor (cpp) introduce all file marked with # and turns into file.i.
- Compiler generates file.s.
- Assembler generates file.o.
- Linker merges object files in a single executable file.
- Executable file.

Suppose we write a C program as two files p1.c and p2.c. We compile with:

```
unix> gcc -O1 -o p p1.c p2.c
```

- Invoking *gcc* invokes the Gcc C compiler.
- *-O1* indicates level one optimization. In general, increasing the level of optimization makes the final program run faster, but at a risk of increased compilation time and difficulties running debugging tools on the code.

If we write

```
unix> gcc -O1 -S code.c
```

we don't compile, we just get the .s file.
Instead if we write:

```
unix> gcc -O1 -c code.c
```

we bot compile and generate the .o file.

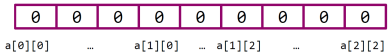
Type	Size (in bytes)
char	1
short	2
int	4
float	4
double	8
long	8

```
#include <stdint.h>

int8_t          a;
int16_t         b;
int32_t         c;
int64_t         d;

uint8_t         x;
uint16_t        y;
uint32_t        z;
uint64_t        w;
```

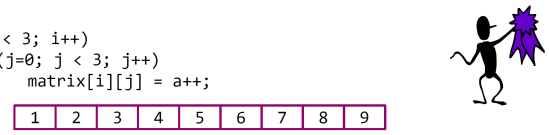
Signed integers, precise size in bits

Unsigned integers,

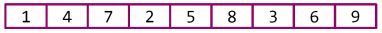
```
int a[3][3];
```




```
int a = 1;
for (i=0; i < 3; i++)
        for (j=0; j < 3; j++)
                matrix[i][j] = a++;
```




```
int a = 1;
for (i=0; i < 3; i++)
        for (j=0; j < 3; j++)
                matrix[j][i] = a++;
```



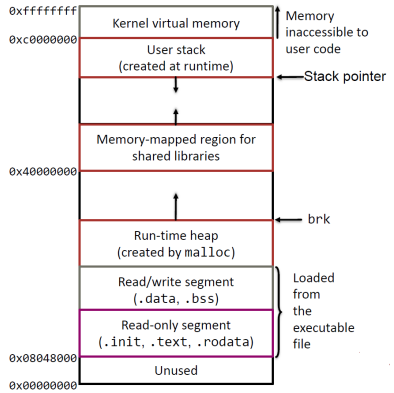
Left shift: fill with zeros on the right.
Logical right shift: fill with zeros on the left.
Arithmetical right shift: replicate most significant bit on the left.
Integers in C can be represented in two ways:

- **Unsigned:** $\sum_{i=0}^{n} x_i \cdot 2^i$
- **Signed:** $-x_n \cdot 2^n + \sum_{i=0}^{n-1} x_i \cdot 2^i$

Some useful rules to remember are:

- Casting between signed and unsigned: bit pattern is maintained but reinterpreted.
- Expression containing signed and unsigned: cast to unsigned.
- $-x =\backsim x + 1$

The OS gives each process an address space, which contains virtual memory. When the OS loads a program it creates an address space, it inspects the executable file to see what's in it and it lazily copies regions of the file into the right place in the address file.



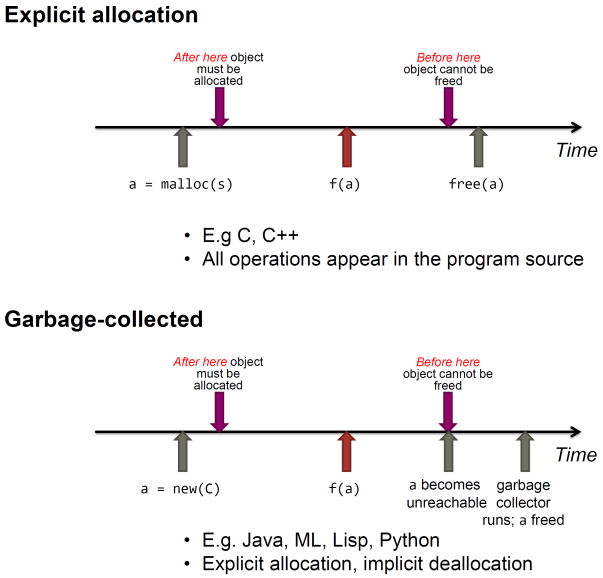
Pointers are a very useful feature in C: they save the address of another variable (i.e. they point to another variable).
So far memory is allocated statically (e.g. the counter is allocated when the program is loaded and is deallocated when program exits) and variables are automatically allocated. Sometimes we want memory which persists across multiple function calls, but not for the whole lifetime of the program. This kind of memory may be too big to fit on the stack and hence is allocated when explicitly requested by the programmer with commands such as malloc (creates a block of desired size and returns a pointer to it or NULL of failure), calloc (same as malloc, but sets the memory to zero) and must be freed with the free command.
Two important commands (used for example to implement exceptions) are:

- **setjmp(env):** saves the current stack environment in env; returns 0.

- **longjmp(env, val):** causes a return to the point saved by env; this time the point where there is env returns val.

Implementing Dynamic Memory Allocation

Memory allocators can be **explicit** (e.g. malloc() and free () in C) or **implicit** (e.g. Java, where freeing is done by a garbage collector.



When implementing memory allocation we have two performance goals:

- **Throughput:** number of completed requests per unit of time.
- **Peak memory utilization:** maximize the ratio between the highest aggregate payload and the current heap size.
- Those goals are often conflicting.

Poor memory utilization can be cause by fragmentation, which comes in two ways:

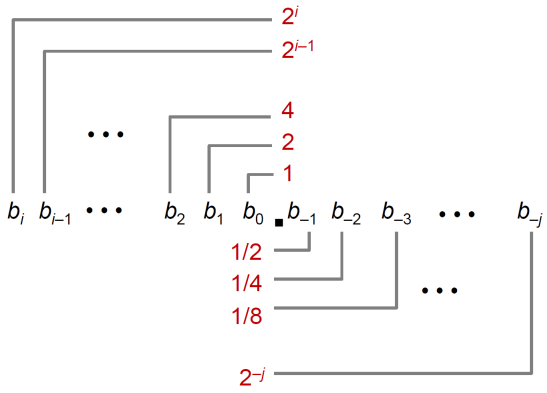
- **External fragmentation:** there is enough aggregate heap memory, but no single free block is large enough. This means that in a book there are some free pages and, if you sum them, there is enough space for what you want to write but, if you want to write sequentially, there is no enough space.
- **Internal fragmentation:** for a given block we have payload ¡ block size (for example because of padding for alignment purposes). This means that in a page of a book we have written only half of the page.

To keep track of free blocks there are several techniques:

- **Implicit free list:** for each block, at the beginning, we save the length of the free space and whether is allocated or not (with a flag bit). When we want to find a free block we can either use next fit (when you find a free spot large enough, use it) or best fit (scan the whole list and pick the best spot). We can also use coalescing: if we free a block and the previous/ next block is also free, we join them.
- **Explicit free list:** we use an explicit list among the free blocks using pointers.
- **Segregated free list:** different free lists for different size classes.
- **Blocks sorted by size:** can use a balanced tree with pointers within each free block.

Floating Point

Fractional binary numbers: with this form you can't express every number; multiplication/ division are easy (simply shift to the left/ right).

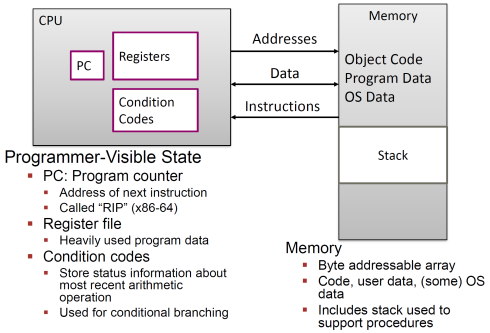


Floating point: 1 sign bit; exp exponent bits; M mantissa bits. Examples: floating points (32 bits, 8 bits exponent, 23 bits mantissa); double (64 bits, 11 bits exponent, 52 bits mantissa).

- $exp = 1...1$, $frac = 0...0$: infinity
- $exp = 1...1$, $frac \neq 0...0$: NaN
- $exp = 0...0$ (denormalized values): E=-Bias+1, where Bias=$2^{e-1} - 1$, M=0.mantissa bits; number=$(-1)^s \cdot M \cdot 2^E$
- $exp \neq 0...0$ and $exp \neq 1...1$ (normalized values): E=exp-bias, where bias=$2^{e-1} - 1$; M=1.mantissa bits; number=$(-1)^s \cdot M \cdot 2^E$

x86

The **ISA** is the parts of a processor design that one needs to understand to write assembly code. The **microarchitecture** is the concrete implementation of the ISA in the hardware. Here we summarizing 64-bit x86.



Let's take a look at an example program:

C code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated x86 assembly

```
sum:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    -24(%rbp), %eax
    movl    -20(%rbp), %edx
    addl    %edx, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

Obtain with command

```
gcc -O -S code.c
```

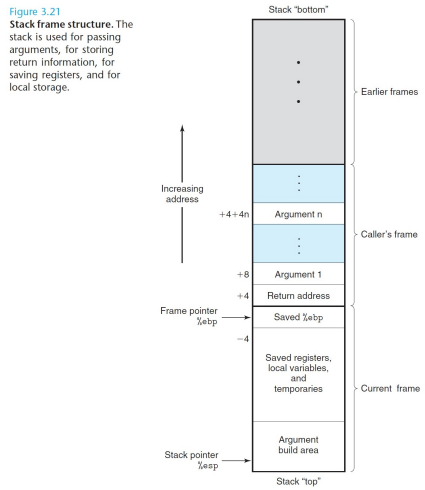
Produces file `code.s`

Some compilers use single instruction "leave"

Systems Programming

This example is useful to illustrate how procedures calls work:

A procedure call involves passing both data (in the form of procedure parameters and return values) and control from one part of a program to another. In addition, it must allocate space for the local variables of the procedure on entry and deallocate them on exit. Most machines, including IA32, provide only simple instructions for transferring control to and from procedures. The passing of data and the allocation and deallocation of local variables is handled by manipulating the program stack.



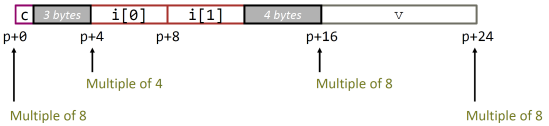
When a method is called the internal command *call* pushes the return address (i.e. the address of the caller + 4) in the stack. Afterwards there is a jump to the beginning of the new procedure. The first instruction *pushq %rbp* pushes the base pointer (i.e. the address where the previous function begins) into the stack. The following instruction *movq %rsp, %rbp* saves the stack pointer into the base pointer, such that the next function that will be called will save this address in the stack. In such a manner we have a linked list of the beginning addresses of all functions. After the procedure is over there is a command *popq %rbp* which takes the value where the stack pointer is pointing to (i.e. the base pointer of the previous function) and update the base pointer with this value (in facts, we are leaving the procedure and hence the next procedure that will be called will need to have the right version of the base pointer). Afterwards the instruction *ret* pops the return address from the stack and jumps to this address.

Structures are contiguously-allocated regions of memory and contains different elements of different types. The elements are **aligned** depending on the type (e.g. char 1 byte, short 2 byte, int and float 4 bytes, double and pointers 8 bytes).

Satisfying alignment with structures

- Example (under Windows or x86-64):
  - K = 8, due to double element

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Optimizing compilers

The compiler is your friend!
The efficiency of a program goes beyond the asymptotic complexity of the algorithm. A lot of other factors are involved:

- Constants ($n \neq 100n$).
- Coding style (unnecessary procedure calls, unrolling, reordering).
- Algorithm structure (locality, instruction level parallelism).
- Data representation.

Hence we have to optimize at multiple levels in order to be efficient (e.g. algorithm, data representation, procedures, loops). Compilers can help you to optimize your code in order to run faster on the machine. For example in *gcc* there are several different compiler flags which gives you different performances (from O1 to O3 you can pass from 200 to 30 cycles).
Some important things to remember about compilers:

- Compilers are good at mapping program to a machine. For example register allocation, code selection and ordering, dead code elimination, eliminating minor inefficiencies.
- Compilers are not good at improving asymptotic efficiency and at overcoming "optimization blockers"(e.g. potential memory aliasing, potential procedure side-effects).

- **If in doubt, the compiler is conservative!**

Now, let's take a look to some examples of compiler behaviours.
Code motion

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni+j] = b[j];
```

Strength reduction $16x \to x << 4$
Common subexpressions: reuse portions of expressions in order to reduce the complexity of the computations.
An example where you have to help the compiler is the following:

```
void lower(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

This loop is $\mathcal{O}(n^2)$ because the *strlen* function takes linear time on the size of the input. A natural thing to do in this case would be to save the length of the input before entering the loop, but the compiler will not do this kind of optimization in this case. In facts, the *strlen* function might have side effects and hence the compiler can not do this kind of code motion. In general, the compiler threats procedure calls as black boxes that can not be analysed. We have to help the compiler to help us!

Architecture

The programmer assumes that his program is translated in a sequence of instruction and that those instructions are executed in order one after the other. In practice things are a little bit different. A first important observation is that instructions can be decomposed in micro instructions. For example:

- Fetch
- Decode
- Execute
- Memory stage
- Write back

This allows to pipeline instructions (i.e. one instruction is in the fetch stage while another one is in the execute stage). Other tricks used to improve performance are:

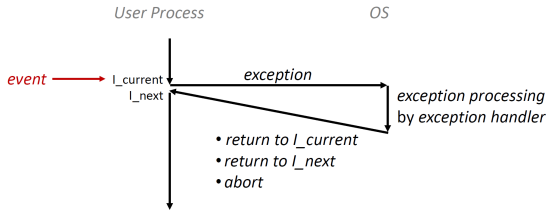
- **Out of order execution:** data dependencies limit the performance of a program. For example if instruction 2 depends on instruction 1, but instruction 3 is independent of both other instructions, then the order 1-3-2 would be more convenient. Out of order execution is useful in those cases: instructions are executed out of order and then reordered with Tomasulo Algorithm.
- **Superscalar processors:** it is possible to do more operations at the same time (e.g. more additions, load or stores at the same time).
- **Branch predictor:** when there are branches (e.g. if statements), the program doesn't know what will be the next instruction. Branch predictors are methods to predict which instruction will be the next one. When branch predictors are used the computer begins to execute the possible next instruction speculatively. If the guess was right there is a better performance, otherwise there will be a penalty time to undo the wrong instruction.

Some other important concepts in computer architecture are:

- Frequency of the processor (usually in GHz): number of cycles every second.
- Latency: time to execute an instruction.
- Throughput: number of instructions executed every second when the pipeline is fully operative.
- Time to execute operations on array: $n \cdot CPE + overhead$.

Exceptions

Processors do only one thing: from startup to shutdown, a CPU simply reads and executes a sequence of instructions, one at a time. This sequence is the CPU's **control flow**. Up to now we can change the control flow with jumps and function's call/ return. However there are more events in which the control flow should be modified, e.g. division by zero, user hits Ctrl-C, ... This means that the system needs mechanisms for **exceptional control flow**.
An **exception** is a transfer of control to the OS in response to some event.



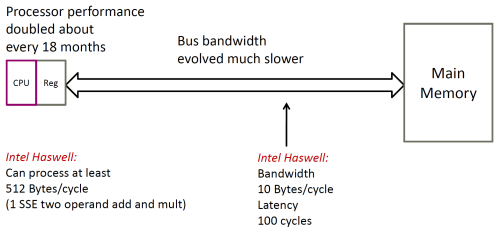
Each type of event has an exception number k, which is an index into the exception table. If exception k occurs, the code pointer by the k-th entry in the exception table is executed. When an exception occurs, there is a switch from **user-mode** to **kernel-mode**. There are several possibilities of the consequences of an exception. Imagine that we are executing an instruction curr and than the exception occurs. It is possible that, after the exception handler has finished its job, we resume from

the instruction which comes after curr, or we do the instruction curr again or that it is impossible to resume after the exception and hence we have an abort.
There are two main types of exceptions:

- **Synchronous exceptions:** caused by events that occur as result of executing an instruction.
  - Traps: system calls, breakpoints, ... Returns control to next instruction.
  - Faults: unintentional, but possibly recoverable. Examples are page faults, protection faults, floating point exceptions. Either re-execution of faulting instruction or abortion.
  - Aborts: unintentional and unrecoverable, e.g. machine check.
- **Asynchronous exceptions:** cause by events external to the processor. Some examples are I/O interrupts (Ctrl-C,arrival of a packet from the network, arrival of data from a disk) and reset interrupts.

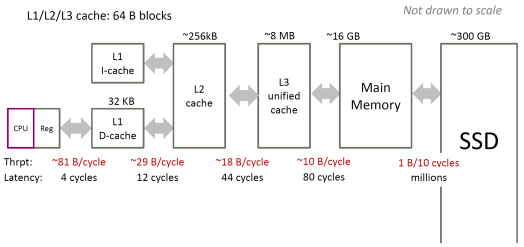
Caches

Problem: Processor-memory bottleneck



Solution: Caches

The memory of a computer is organized in a hierarchical way. The general rule is: **smaller means faster**. On the top of the hierarchy there are the registers (very small and very fast), then there are the caches (of different sizes, but should be a lot faster than memory, use SRAM technology), then there is the main memory and then the disk (which use DRAM technology).



The principle of the memory hierarchy is to take advantage of locality to improve performance: when data in block b is needed, we first check, whether b is in the cache. If b is in the cache we are over and we saved the time to search b in lower (and slower) level. Otherwise we spend some time to take b from one of the lower levels and then we save b in the cache. In facts, if we think that we are going to need b also in the close future, having b cached will save query time for the next time we will need it. The **placement policy** determines where in the cache the block will be stored. But what do we do if we want to save b in the cache but the slot designed by the placement policy is occupied? What if the cache is completely full? We need a **replacement policy** which determines which block gets evicted.
Caches improve performance because we exploit two kind of localities:

- **Temporal locality:** recently referenced items are likely to be referenced in the near future (example, the i variable of a for loop).
- **Spatial locality:** items with nearby addresses tend to be referenced close together in time (e.g. sequential array access).

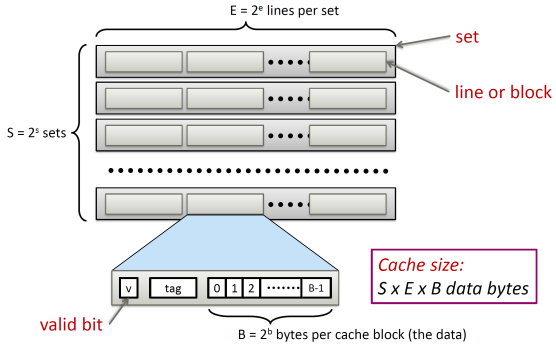
When we talk about caches it is useful to do some performance considerations:

- **Miss rate:** $\frac{\text{misses}}{\text{accesses}} = 1 - \text{hit rate}$
- **Hit time:** time to deliver a line in the cache to the processor (e.g. 1-2 cycles for L1, 5-20 cycles for L2).
- **Miss penalty:** additional time required because of a miss (e.g. 50-200 cycles for main memory, but this number tends to increase).

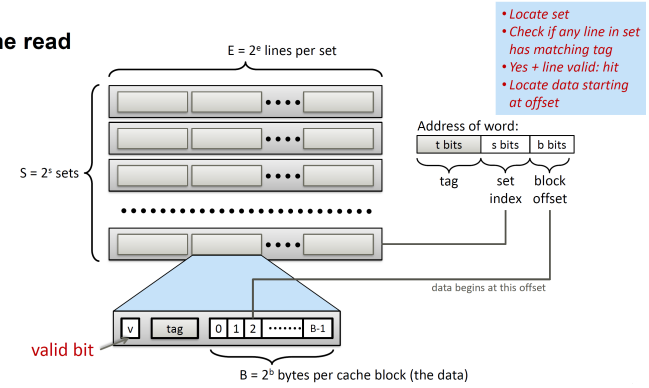
Cache misses can be of different types:

- **Cold (compulsory) misses:** occurs on first access to a block.
- **Conflict miss:** place in the cache where the block should go is occupied. Cache may be large enough, but multiple lines map to the same slot.
- **Capacity miss:** set of active cache blocks is larger than the cache.
- **Coherency miss:** occurs on multiprocessors systems.

General cache organization (S, E, B)



Cache read



Writes are more involved and we have different possible designs:

- **Write-hit:**

  - Write-through: write immediately to memory, such that memory is always consistent to the cache copy. This can be slow if the same line is written several times.
  - Write-back: defer write to memory. This needs a dirty bit in the cache to show that the copy is not consistent with memory. This can improve the performance but it is more complex.

- **Write-miss:**
  - Write-allocate: load into the cache, update line in the cache. Common with write-back caches.
  - No-write-allocate: writes immediately to memory. Seen with write-through caches.

If we have a direct mapped cache and we have a write miss there is only one line to evict. However, in case of associative caches we have to choice which block to evict. There are several strategies:

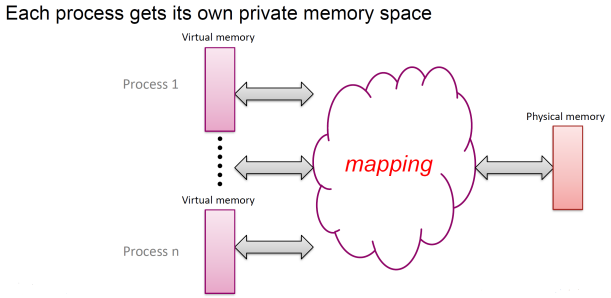
- **Random**
- **LFU:** evict the block used less frequently.
- **LRU:** evict the block used less recently.

When we design a cache we have to take several choices, let's see some trade-offs:

- LFU and LRU need a more involved hardware structure and hence introduce some overhead: we choose this policy replacement for low level of caches (i.e. where miss have a big cost).
- Cache size: if bigger we have a better hit rate, but also a higher latency to find the data.
- Block size: if we have big blocks we advantage programs with a good spatial locality. However, if we considered a cache with fixed size, we have less cache lines and this is bad for programs with a good temporal locality. Moreover size of the block and miss penalty are proportional.
- Associativity: high associativity means less conflict misses. However the cost of the replacement policy is high and hence the miss penalty gets bigger.

Memory

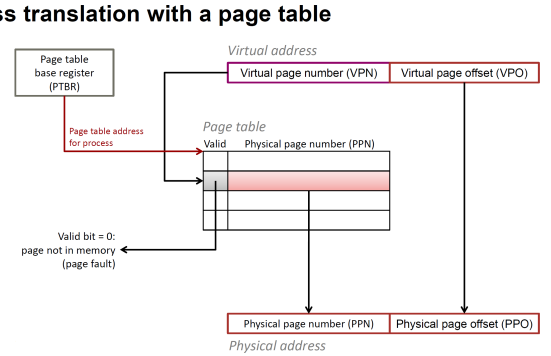
Programs refer to virtual memory addresses in order to provide a good abstraction for programmers. Conceptually it is possible to think to memory as a very large array of bytes, where each byte has it own address. This is only an approximation which is far from reality: memory is not homogeneous (there are several layers with different access times) and is not true that every process has its own private large memory. In reality the OS gives this illusion by mapping virtual addresses to physical memory.



When we consider the mapping between virtual and physical addresses we have to consider that:

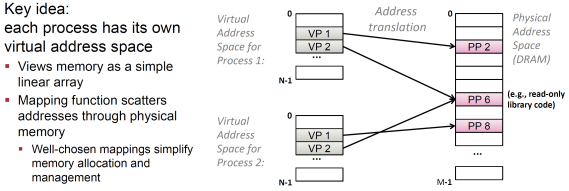
- Each object can have multiple addresses.
- Every byte in main memory can be considered as one physical address or one (or more) virtual addresses.

Some simple systems like elevators still use physical addressing, but in general all modern devices (laptops, desktops...) use systems with virtual addressing. A basic principle is the following: in memory there is a page table which acts as a translation table between virtual pages and virtual pages.



The following are some advantages of virtual memory:

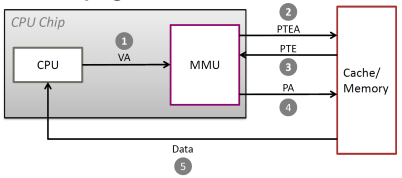
- Efficient use of limited main memory (RAM). RAM is a cache for the parts of a virtual address space: some parts are cached in RAM, some are located on disk. Both the RAM and the disk are DRAM memories, but RAM is a lot faster.
- Simplifies memory management for programmers. Each process gets the same full, private linear address space.



- Isolates address spaces. It is possible that two virtual addresses of two different processes map to the same memory location. In order to protect the processes from problems (e.g. race conditions), there are permission bits to regulate their interaction.

Let's take a look at the address translation process in case of a page hit:

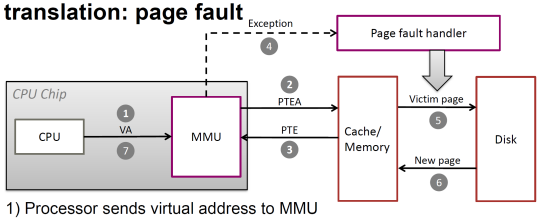
Address translation: page hit



1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) MMU sends physical address to cache/memory
5) Cache/memory sends data word to processor

And in case of a page fault:

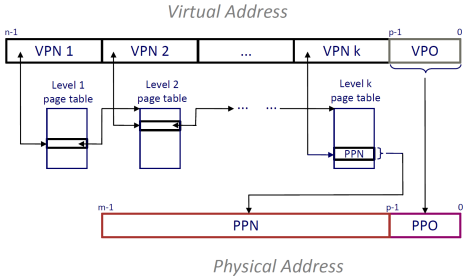
Address translation: page fault



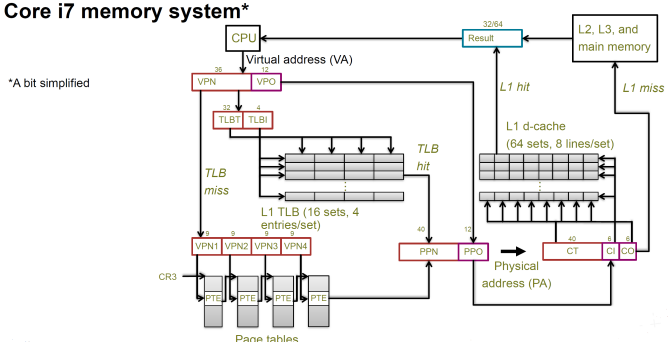
1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim (and, if dirty, pages it out to disk)
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction

In order to speed up the translation we can use a TLB. We send the virtual address to the MMU which, instead of looking directly in the memory in order to translate, looks in the TLB: a cache for the virtual/ physical address translation. If the physical address corresponding to the requested virtual address is located in the TLB we save a memory access. If we have a miss we do the translation as before with access to the memory and we work with TLB in the same way we would do with a normal cache.

Translating with a k-level page table



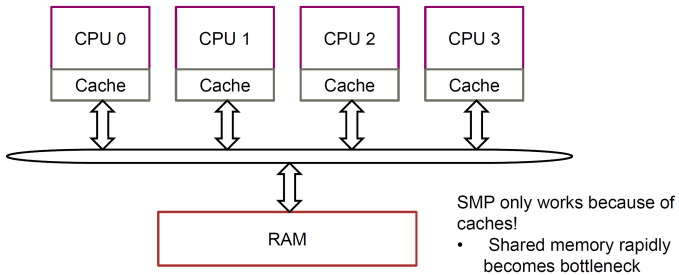
Core i7 memory system*



*A bit simplified

Multicore

One processor isn't fast enough and some jobs can parallelize. A solution is attaching multiple processors to the system bus.



Having more than one core is now crucial since some walls have been hit:

- The memory wall
- The ILP wall
- The power wall
- Moore's law is not respected anymore
- Processor cores can't get any faster
- Clock frequencies are going down

For this reasons nowadays we use multiple processors per chip. Mostly so far the multiprocessors share memory. Two key challenges are:

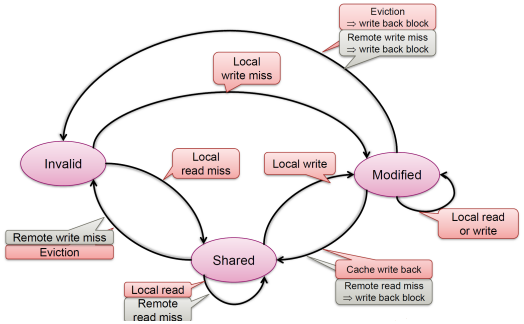
- **Coherency:** values in cache all match each other; processors all see a coherent view of memory.
- **Consistency:** the order in which changes to memory are seen by different processors.

Two important concepts are **program order** (i.e. order in which a program on a processor appears to issue read and writes) and **visibility order** (order which all reads and writes are seen by one or more processors, refers to all operations in the machine).

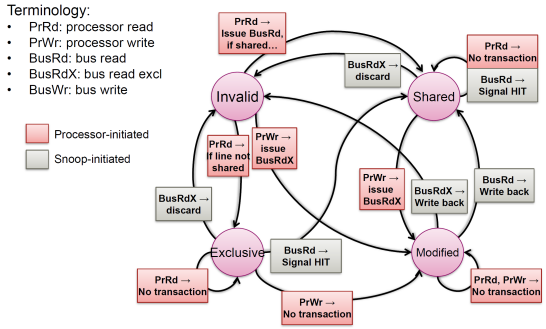
On modern machines most CPU cores are cache coherent which means they behave as if they were all accessing a single memory array. This makes programming easier but is hard to implement and memory is slower as a result.

An important property is sequential consistency, i.e. operations from each processor appear in program order and every processor's visibility order us the same interleaving of all the program orders. In order to achieve sequential consistency we can use a snoopy cache (the cache snoops on read/ writes of other processors and, if a line is valid in another cache, the local line of the snooping processor gets invalidated. This works for write-through caches, but for write-back caches we need a more advanced protocol.

MSI state machine: all transitions



MESI state machine



Some interesting things

- **Pass by value:** the caller and callee have two independent variables with the same value. If the callee modifies the parameter variable, the effect is not visible to the caller.
- **Pass by reference:** the caller and the callee use the same variable for the parameter. If the callee modifies the parameter variable, the effect is visible to the caller's variable.

A **worm** is a program that can run by itself and can propagate a fully working version of itself to other computers. A **virus** adds itself to other programs and cannot run independently. A very famous example is the **buffer overflow** attack. When a function is called, its return address is pushed in the stack. Then the function is called and, very often it allocates space on the stack. If the function uses some not carefully implemented input function (which takes data from example from the user keyboard), the following scenario can happen: the input of the keyboard is saved in the stack, but if the input is too big and the system is not carefully designed, it can happen than the input overwrites some memory in the stack which it should not overwrite. A malicious user could for example overwrite the return address (which is also saved in the stack) such that the return address is the one of a malicious function which does whatever the user wants!

x86-64 Reference Sheet (GNU assembler format)

Instructions

Data movement

`movq` Src, Dest	Dest = Src
`movsbq` Src,Dest	Dest (quad) = Src (byte), sign-extend
`movzbq` Src,Dest	Dest (quad) = Src (byte), zero-extend

Conditional move

`cmove` Src, Dest	Equal / zero
`cmovne` Src, Dest	Not equal / not zero
`cmovs` Src, Dest	Negative
`cmovns` Src, Dest	Nonnegative
`cmovg` Src, Dest	Greater (signed >)
`cmovge` Src, Dest	Greater or equal (signed ≥)
`cmovl` Src, Dest	Less (signed <)
`cmovle` Src, Dest	Less or equal (signed ≤)
`cmova` Src, Dest	Above (unsigned >)
`cmovae` Src, Dest	Above or equal (unsigned ≥)
`cmovb` Src, Dest	Below (unsigned <)
`cmovbe` Src, Dest	Below or equal (unsigned ≤)

Control transfer

`cmpq` Src2, Src1	Sets CCs Src1 – Src2
`testq` Src2, Src1	Sets CCs Src1 & Src2
`jmp label`	jump
`je label`	jump equal
`jne label`	jump not equal
`js label`	jump negative
`jns label`	jump non-negative
`jg label`	jump greater (signed >)
`jge label`	jump greater or equal (signed ≥)
`jl label`	jump less (signed <)
`jle label`	jump less or equal (signed ≤)
`ja label`	jump above (unsigned >)
`jb label`	jump below (unsigned <)
`pushq` Src	%rsp = %rsp – 8, Mem[%rsp] = Src
`popq` Dest	Dest = Mem[%rsp], %rsp = %rsp + 8
`call label`	push address of next instruction, `jmp label`
`ret`	%rip = Mem[%rsp], %rsp = %rsp + 8

Arithmetic operations

`leaq` Src, Dest	Dest = address of Src
`incq` Dest	Dest = Dest + 1
`decq` Dest	Dest = Dest − 1
`addq` Src, Dest	Dest = Dest + Src
`subq` Src, Dest	Dest = Dest − Src
`imulq` Src, Dest	Dest = Dest ∗ Src
`xorq` Src, Dest	Dest = Dest ^ Src
`orq` Src, Dest	Dest = Dest | Src
`andq` Src, Dest	Dest = Dest & Src
`negq` Dest	Dest = − Dest
`notq` Dest	Dest = ∼ Dest
`salq` k, Dest	Dest = Dest $\ll k$
`sarq` k, Dest	Dest = Dest $\gg k$ (arithmetic)
`shrq` k, Dest	Dest = Dest $\gg k$ (logical)

Addressing modes

- **Immediate**
  $val Val
  val: constant integer value
  `movq $7, %rax`
- **Normal**
  (R) Mem[Reg[R]]
  R: register R specifies memory address
  `movq (%rcx), %rax`
- **Displacement**
  D(R) Mem[Reg[R]+D]
  R: register specifies start of memory region
  D: constant displacement D specifies offset
  `movq 8(%rdi), %rdx`
- **Indexed**
  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+D]
  D: constant displacement 1, 2, or 4 bytes
  Rb: base register: any of 8 integer registers
  Ri: index register: any, except %esp
  S: scale: 1, 2, 4, or 8
  `movq 0x100(%rcx,%rax,4), %rdx`

Instruction suffixes

`b`	byte
`w`	word (2 bytes)
`l`	long (4 bytes)
`q`	quad (8 bytes)

Condition codes

CF	Carry Flag
ZF	Zero Flag
SF	Sign Flag
OF	Overflow Flag

Integer registers

%rax	Return value
%rbx	Callee saved
%rcx	4th argument
%rdx	3rd argument
%rsi	2nd argument
%rdi	1st argument
%rbp	Callee saved
%rsp	Stack pointer
%r8	5th argument
%r9	6th argument
%r10	Scratch register
%r11	Scratch register
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved