Sysprog
Summary

November 29, 2020

Contents

Chapter 1

Introduction

1.1 Computer arithmetic

Arithmetic operation have important mathematical properties but we cannot assume all usual mathematical properties, due to finiteness of representations e.g $x^2 \geq 0$ is valid for floats but not for ints and associativity is valid for ints but not for floats.

C and C++ dont provide any memory protections:

- Out of bounds array references
- Invalid pointer values
- Abuses malloc/free

Besides asymptotic complexity constant factors matter to hence even the exact operation count does not predict performance, Optimization happens at multiple levels e.g algorithms,data representations, procedures and loops.

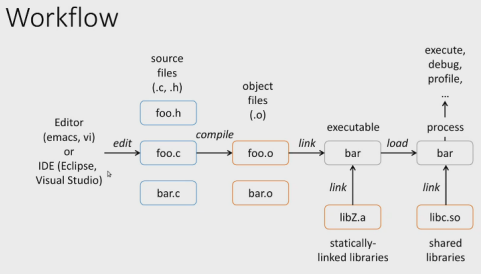
Chapter 2

Introduction to C

#include Takes a file and includes it into the compilation of the current file

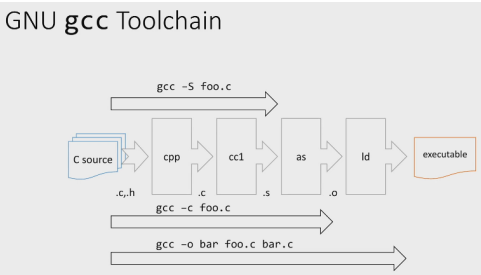
return 0 In C there are no exceptions, hence to indicate that everything went well we return 0 otherwise we return a non-zero value

2.1 Workflow



The steps:

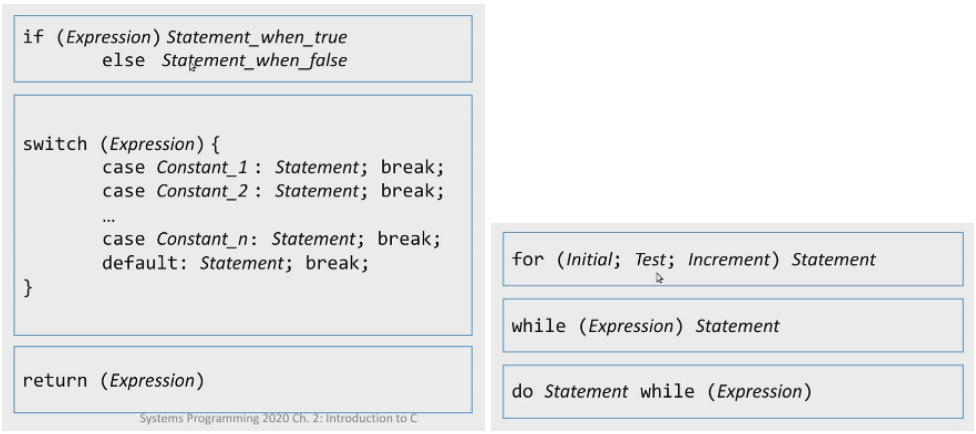
- Edit a file
- Get different kinds of c files
- c files get compiled into object code, which is machine code which doesnt connect to any part of the program i.e each of the files get compiled in isolation
- the linker converts the object files into executables (i.e code which can run)
- loading takes the executable and turns it into a running process in memory and in doing so pulls in the last bits of code needed to run.



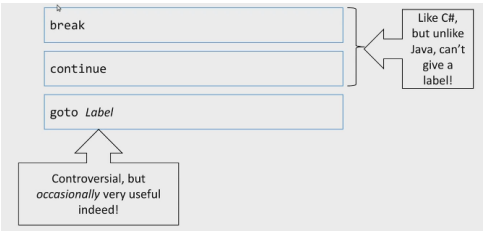
2.1.1 Questions

- What is the difference between an object file and an executable The object contains machine code, typically it only contains part of the program, hence it contains a lot of placeholders (it knows what it needs to do but not how)

2.2 control flow in C



For the switch statement break is needed to leave, otherwise it will continue with the next case, we define a default case which is executed if it doesnt match any cases.



continue stops the current iteration of the loop and starts with the next iteration. "goto" is a unconditional branch.

2.3 Functions

Each function similar to java has:

- Name
- Return type
- Argument types
- Body

Every C program needs to have a function called main()

```c
/* program to print arguments from command line */
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    printf("argc = %d\n\n", argc);
    for (i=0; i<argc; ++i) {
        printf("argv[%d]: %s\n",i, argv[i]);
    }
    return 0;
}
```

char *argv[] can be seen as a list of strings. "int argc" indicates how many strings are in the list
printf printf takes multiple arguments, the first one is a string each % is replaced with the arguments that follow

2.4 Basic Types in C

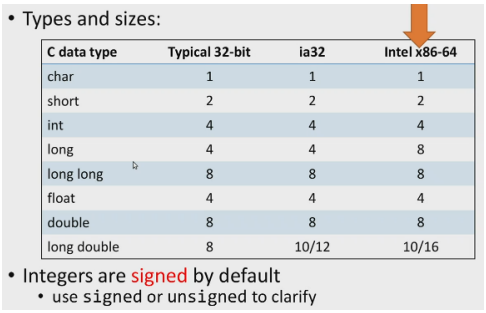
2.4.1 Declarations

Are like java e.g int my_int, double $some\_float = 0.123$ etc. when we declare a variable inside the block, the scope is just the block after the block, the variable disappears, unless we declare it as static. A static variable persists between calls i.e the next time you call the function the variable will keep its value.

Outside a block, the scope is the entire **program** i.e not just this file but all files that have been compiled in the program. (Called Global Variables). We can limit the scope of a global variable to just the file its in by using static.

2.4.2 Integers and floats

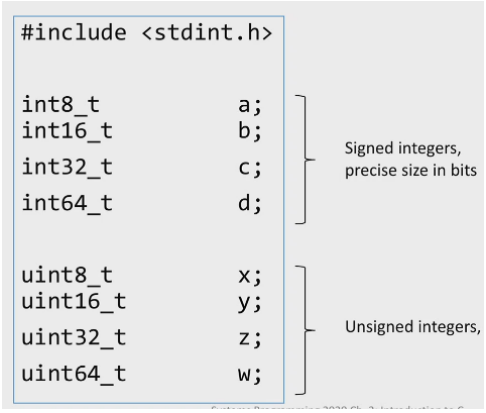
The size of an integer in c i.e the max value it can have depends on the machine you are running on.



Hence depending on the machine and using large integers the code could have different performance. Integers in C are either signed or unsigned. They are signed by default.
Rules for arithmetic on integers and floats:

- Implicit conversion between integer types, i.e adding a larger int and a smaller int results in the sum with bit size of the larger int.
- Implicit conversion between floating types
- Explicit conversion between any 2 types can be done with casting

2.4.3 C99 extended integer types

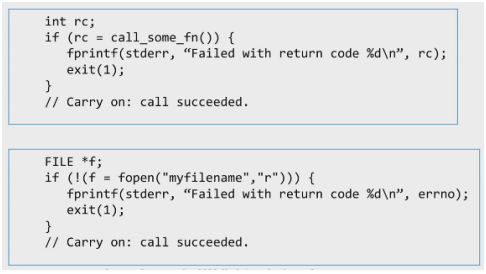


2.4.4 Booleans

Booleans in C are also integers. 0 is interpreted as false and anything thats non-zero is interpreted as true. The NOT operator "!" turns zero into a non-zero and vice-versa.
C99 supports a bool type, it is completely option and still an integer. You can use it by:

$$\#include < stdbool.h >$$

In most programming languages we distinguish between statements (things that do stuff, while,if, assignment, etc) and expressions (set of terms which evaluate to a value). In C all statements are also expressions.



```
int rc;
if (rc = call_some_fn()) {
   fprintf(stderr, "Failed with return code %d\n", rc);
   exit(1);
}
// Carry on: call succeeded.
```

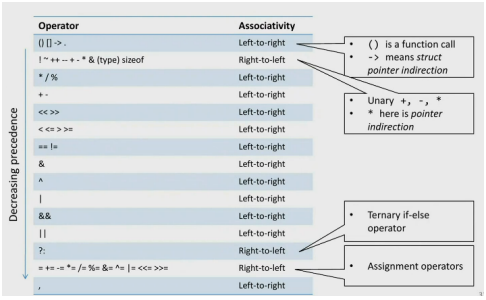
```
FILE *f;
if (!(f = fopen("myfilename","r"))) {
   fprintf(stderr, "Failed with return code %d\n", errno);
   exit(1);
}
// Carry on: call succeeded.
```

the exit method ends the programm regardless, the argument decides how the programm is stopped.

2.4.5 void

void is type that has no value, and used for functions which dont return a particular value (procedures). Also used for untyped pointers (pointers that point purely to an address i.e just a pointer we dont know whats there)

2.4.6 Operators



Left to right associativity: A+B+C → (A+B)+C
Right to left associativity: A+=B+=C → A+=(B+=C)
Post-increment i++: Take the current value of i, add one to it and return the value it used to be. Pre-increment ++i: First add one to i and then return the value of i Post/pre-increment works for pointers too.

2.4.7 Casting

An operator which takes a value of a certain type and returns a value of a completely different type. Bit-representation usually does not change.

2.5 Arrays in C

An array is a finite vector of variables, all the same type. Indexing starts from 0. In an n element array the last element is a[n-1]. In most languages bound checking is done and you get an error when you go out of bounds. In C the compiler **does not** check the array bounds!
Multi-dimensional arrays are stored in a contiguous array in row-major.

Array initializers: Arrays can be initialized when they are defined:

```
/* a[0] = 3,
   a[1] = 7,
   a[2] = 9 */
int a[3] = {3, 7, 9};

/* list[0]=0.0, …,
   list[99]=0.0 */
float list[100] = {};

int a[3][3] = {
        { 1, 2, 3},
        { 4, 5, 6},
        { 7, 8, 9},
};
```

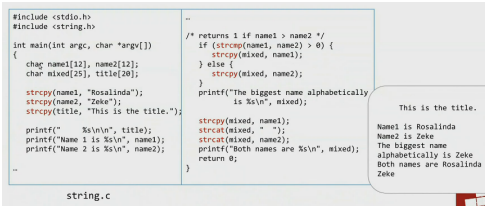
2.5.1 Strings

C has no real string type, instead a string is an array of char's terminated with null \0

```
char str[6] = {'h','e','l','l','o','\0'};
```

... is the same as:

```
char str[6] = "hello";
```



```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char name1[12], name2[12];
    char mixed[25], title[20];

    strcpy(name1, "Rosalinda");
    strcpy(name2, "Zeke");
    strcpy(title, "This is the title.");

    printf("     %s\n\n", title);
    printf("Name 1 is %s\n", name1);
    printf("Name 2 is %s\n", name2);
…
```

```
…

/* returns 1 if name1 > name2 */
    if (strcmp(name1, name2) > 0) {
        strcpy(mixed, name1);
    } else {
        strcpy(mixed, name2);
    }
    printf("The biggest name alphabetically
            is %s\n", mixed);

    strcpy(mixed, name1);
    strcat(mixed, " ");
    strcat(mixed, name2);
    printf("Both names are %s\n", mixed);
    return 0;
}
```

```
     This is the title.

Name1 is Rosalinda
Name2 is Zeke
The biggest name
alphabetically is Zeke
Both names are Rosalinda
Zeke
```

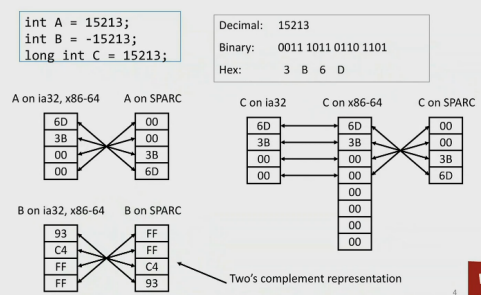
string.c

Chapter 3

Representing Integers in C

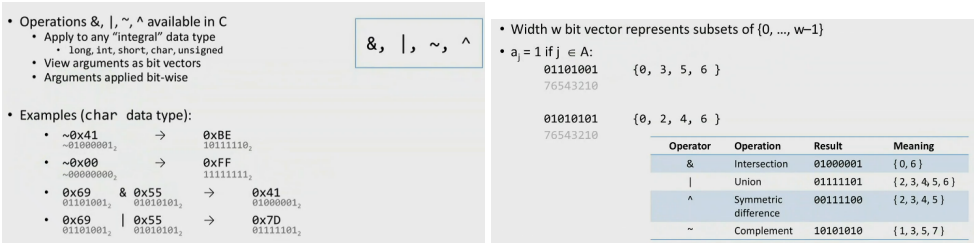
3.1 Encodings and operators

3.1.1 Representing Integers

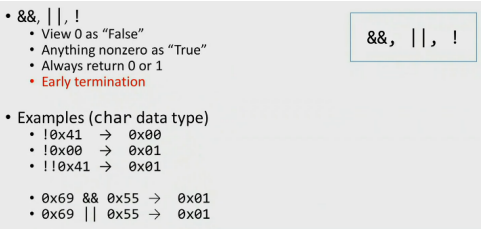


The negative in binary is the bits inverted + 1.

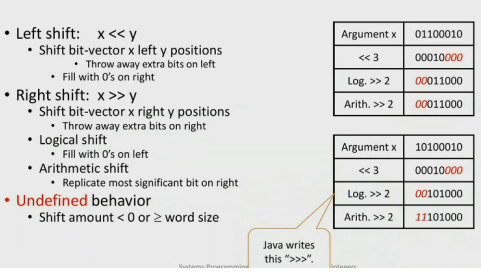
3.1.2 Bit-level operations in C



the contrast logic operations in C:



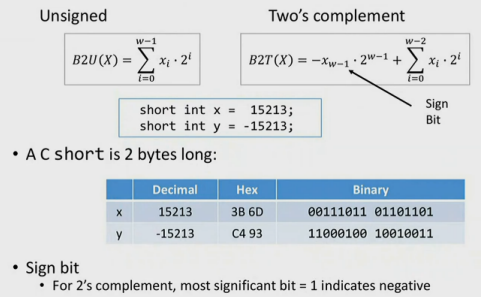
Shift Operations



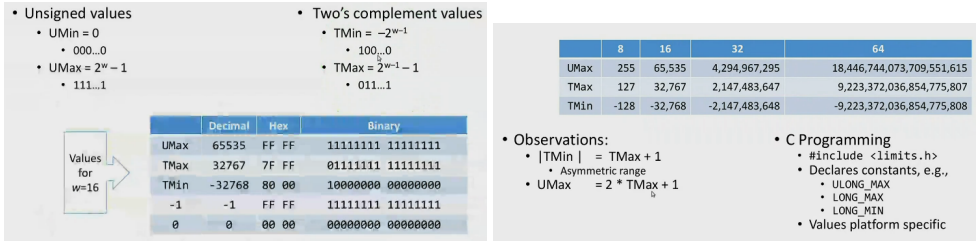
In C we do not have different operators for logical and arithmetic shift, if we have a signed int then we do an arithmetic shift, and with unsigned int logical shift

3.2 Integer Ranges

3.2.1 Encoding Integers



3.2.2 Numeric ranges



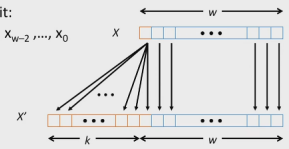
3.2.3 Signed vs Unsigned in C

By default constants are considered to be signed Integers, they are unsigned if they have "U" as a suffix. The bit composition in explicit casting remains the same hence a large unsigned value could be a negative signed value and vice versa. Implicit casting can also occur e.g during assignments or function calls.
When mixing unsigned and signed values in a single expression, signed values are implicitly cast to unsigned (including comparison operators!)

Constant 1	Constant 2	Relation	Evaluation
0	0U	==	Unsigned
-1	0	<	Signed
-1	0U	>	Unsigned
2147483647	-2147483647-1	>	Signed
2147483647U	-2147483647-1	<	Unsigned
-1	-2	>	Signed
(unsigned)-1	-2	>	Unsigned
2147483647	2147483648U	<	Unsigned
2147483647	(int) 2147483648U	>	signed

3.2.4 Sign extension

- Task:
  - Given w-bit signed integer x
  - Convert it to w+k-bit integer with same value
- Rule:
  - Make k copies of sign bit:
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$ (k copies of MSB)



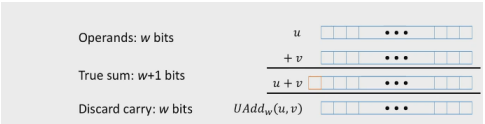
- Converting from smaller to larger integer data type
- C automatically performs sign extension for signed values

```
short int x =  15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

3.3 Integer addition and subtraction in C

3.3.1 Unsigned addition



- Standard addition function
  - Ignores carry output
- Implements modular arithmetic

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

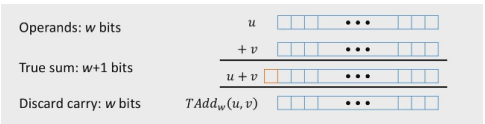
$$UAdd_w(u, v) = \begin{cases} u + v, & u + v < 2^w \\ u + v - 2^w, & u + v \geq 2^w \end{cases}$$

The sum wraps around if the true sum $\geq 2^w$. The properties are:

- Modular addition forms an Abelian group
  - **Closed** under addition
    $$0 \leq UAdd_w(u, v) \leq 2^w - 1$$
  - **Commutative**
    $$UAdd_w(u, v) = UAdd_w(v, u)$$
  - **Associative**
    $$UAdd_w(t, UAdd_w(u, v)) = UAdd_w(UAdd_w(t, u), v)$$
  - 0 is additive **identity**
    $$UAdd_w(u, 0) = u$$
  - Every element has additive **inverse**
    Let: $UComp_w(u) = 2^w - u$
    Then: $UAdd_w(u, UComp_w(u)) = 0$

this forms a different abelian group than the normal addition

3.3.2 Two's complement addition



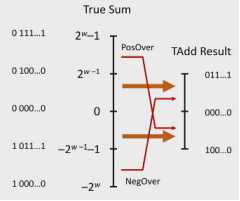
- TAdd and UAdd have identical bit-level behavior
  - Signed vs. unsigned addition in C:

```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v;
```

will give: `s == t`

Hence the only difference between Uadd and Tadd is how the result is interpreted. However the overflow is different:

- Functionality
  - True sum requires w+1 bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer



$$TAdd_w(u, v) = \begin{cases} u + v + 2^w & u + v < TMin_w \quad \text{(Neg. overflow)} \\ u + v & TMin_w \leq u + v \leq TMax_w \\ u + v - 2^w & TMax_w < u + v \quad \text{(Pos. overflow)} \end{cases}$$

The mathemimatical properties of Tadd:

- Group isomorphic to unsigneds with UAdd
  - Since both have identical bit patterns

$$TAdd_w(u, v) = U2T\left(UAdd_w(T2U(u), T2U(v))\right)$$

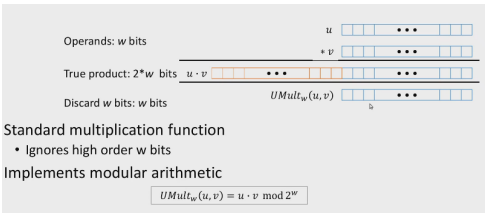
- 2's complement under TAdd forms a group
  - Closed, commutative, associative,
  - 0 is additive identity
  - Every element has additive inverse

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

3.3.3 Integer multiplication in C

Multiplication is worse than addition, because the number of extra bits which could be needed is much larger

- Computing exact product of w-bit numbers x, y
  - Either signed or unsigned
- Ranges
  - Unsigned (up to 2^w bits):
    $0 \leq x * y \leq (2^w - 1)2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (up to 2^{w-1} bits):
    $x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (up to 2^wbits, but only for $(TMin_w)^2$):
    $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
- Maintaining exact results
  - Would need to keep expanding word size with each product computed
  - Done in software by "arbitrary precision" arithmetic packages



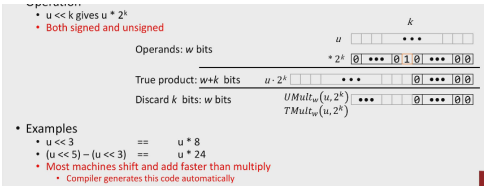
Unsigned multiplication with addition forms a commutative ring. Signed multiplication is isomorphic algebra to unsigned multiplication and addition. Difference to normal mathematics is that there is no guarantee that the addition of two positive numbers signed numbers is also positive.

3.4 Integer multiplication and division using shifts

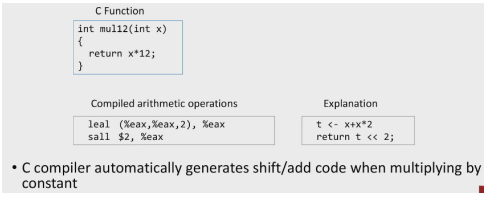
3.4.1 Power of 2 multiply with shift

For both signed and unsigned numbers we have:

$$u << k \text{ gives } u \cdot 2^k$$



Shifting and addition/subtraction are quite simple operations. On the other hand multiplication of random numbers are more expensive. But usually not necessary to write the multiplication explicitly because the compiler writes the multiplications as shifts and additions/subtractions:



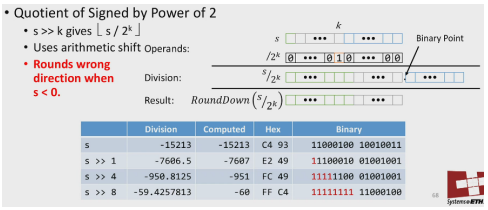
3.4.2 Unsigned power of 2 divide with shift

Same as multiplication but right logical shift where the bits after the binary are truncated. Hence

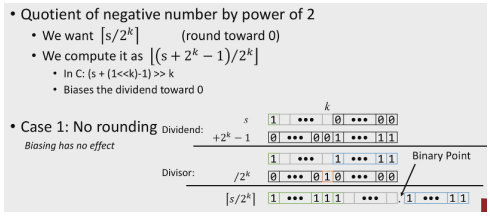
$$u >> k \text{ gives } \lfloor \frac{u}{2^k} \rfloor$$

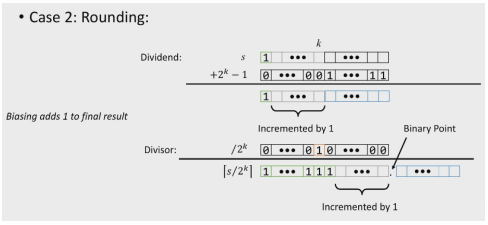
The compiler will rewrite division into shifts and additions aswell.

3.4.3 Signed power of 2 divide with shift



The rounding we want is toward 0. The correct power of 2 division is given by:





When doing a signed division the compiler will do a check if the number is less than zero, if so then it will add the bias and proceed.

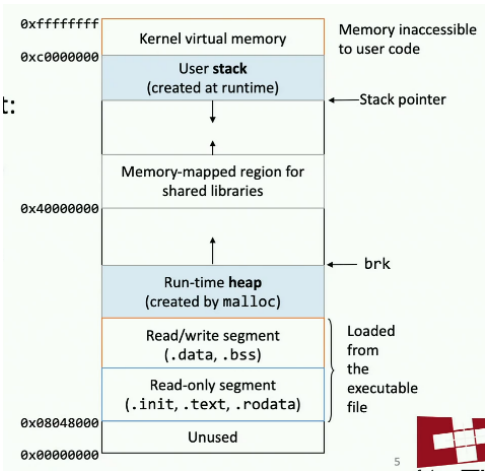
Chapter 4

Pointers

Pointers are variables which store a memory address. The Operating system gives each process an address space. The address space contains the virtual memory which is only visible my the process. Memory is byte addressable.

4.1 Loading

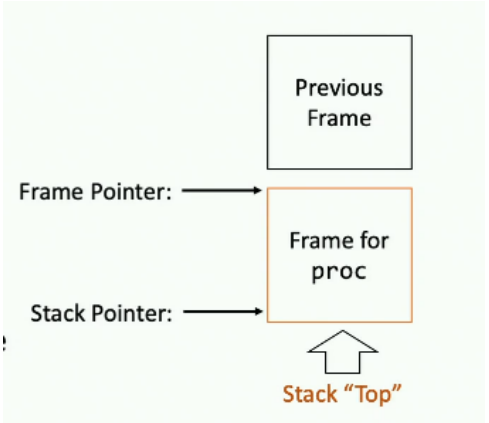
The file is C program is compiled, and we get an executable file, which is what we are actually trying to run. The OS needs to load this file, hence it inspects whats inside, and will load different parts of the programm into different parts of the virtual address space.



4.2 Stack

The Stack is created at runtime, everytime a procedure is called the stack grows, when a procedure completes the stack shrinks, the stack grows downwards. We keep track where the stack is in memory with the stack pointer.

4.2.1 Stack Frames

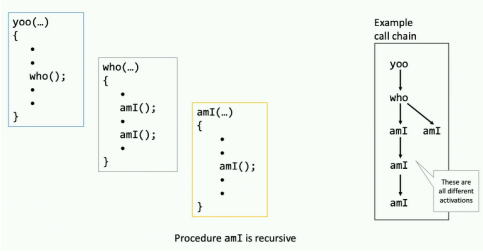


The unit of allocation in a stack is a stack Frame. For each instance of a procedure we get a stack frame. The following information is in a stack frame:

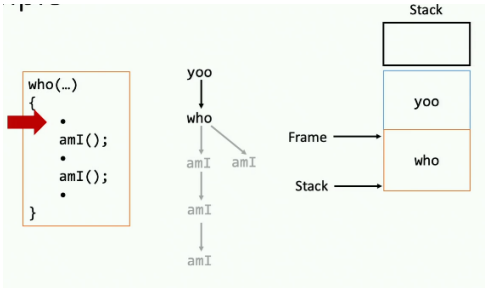
- Local variables
- Return information
- Temporary space

4.2.2 Call Chain example:

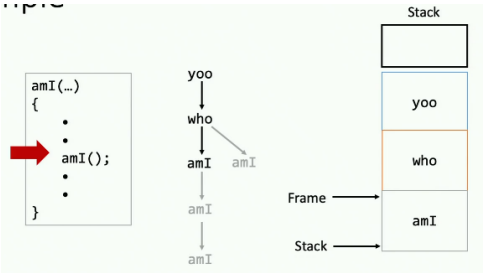
Given the following call chain:



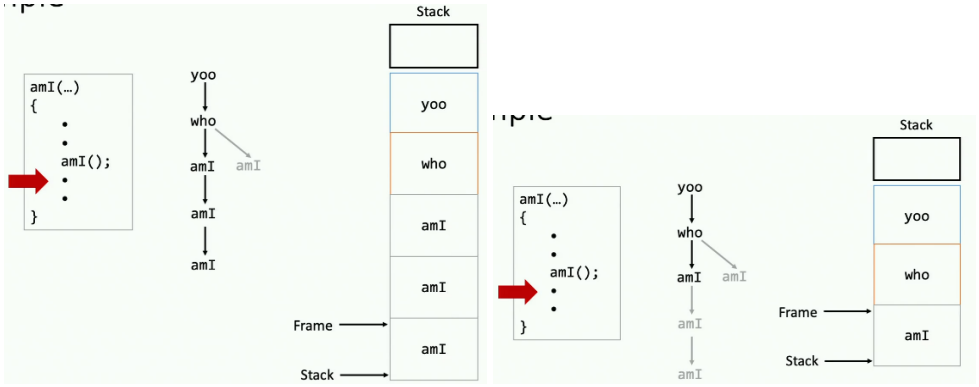
We start with an empty stack, yoo is called first, hence we have a stack frame for yoo. yoo then calls who so we add a stack frame for who on top of the stack. In general whichever procedure youre currently in, that stack frame will be on top of the stack.



The Frame pointer keeps track of the beginning of the Frame, the stack pointer the end of the frame. When we enter the procedure amI, we create a new frame and hence adjust the two pointers.



For each recursive call of amI we add a stack frame, and when they return we pop the stack.



4.3 Pointers

4.3.1 Addresses and &

& ist the reference operator. Given a variable x, $\&x$ produces the virtual address where the value of x is stored. We can use $\%p$ in the printf() method to print the address of x out.

4.3.2 Pointers

Pointers are variables which store memory addresses.

```
type *name;             // declare a pointer
type *name = address;  // declare + initialize a pointer
```

A variable that contains a memory address

points to somewhere in the process' virtual address space

```c
int main(int argc, char *argv[])
{
    int x = 42;
    int *p;     // p is a pointer to an integer
    p = &x;     // p now stores the address of x
    printf("x is %d\n", x);
    printf("&x is %p\n", &x);
    printf("p is %p\n", p);
    return 0;
}
```

pointy.c

Pointers can be dereferenced which means that we get the value stored at the memory address of the pointer.

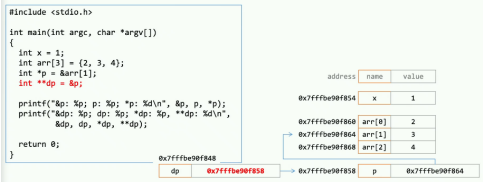
```
v = *pointer;          // dereference a pointer
*pointer = value;      // dereference / assign
```

dereference: access the memory referred to by a pointer

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int x = 42;
    int *p;     // p is a pointer to an integer
    p = &x;     // p now stores the address of x
    printf("x is %d\n", x);
    *p = 99;
    printf("x is %d\n", x);
    return 0;
}
```

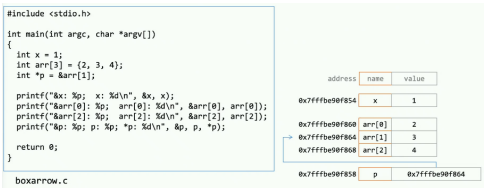
deref.c

A pointer to a pointer (**double pointer**) can be initialized as follows $**dp = \&p$ where p is a pointer.



4.3.3 Box and arrow diagrams

A way to visualize pointers in a program.



boxarrow.c

4.3.4 Address Space Layout Randomization

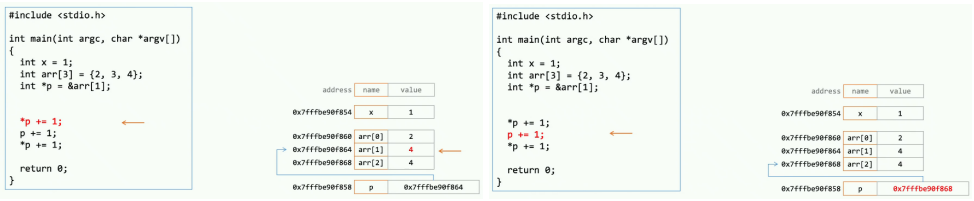
The Base of the stack will be started at a random address by the OS, hence everytime we run a programm the address of the variables will have a different address. This is done as a security measure.

4.3.5 NULL

A memory location guaranteed to be invalid. Null is of type "void *". Any attempt to dereference Null results in a segmentation fault. This is useful to dynamically allocate memory.

4.4 Pointer arithmetic

When incrementing the pointer by one the address it points to after depends on the type, in the example below the pointer is an integer pointer, hence we increment by the size of an integer

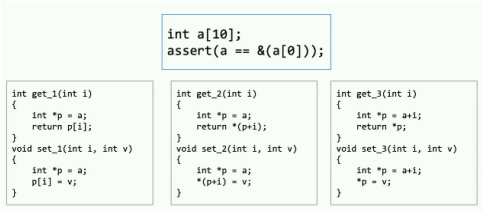


The amount of memory a value takes up depends on the machine and compiler. We can use the methods:

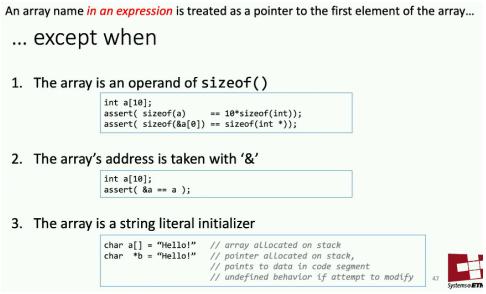
sizeof(type) or sizeof(value)

4.5 Arrays and Pointers

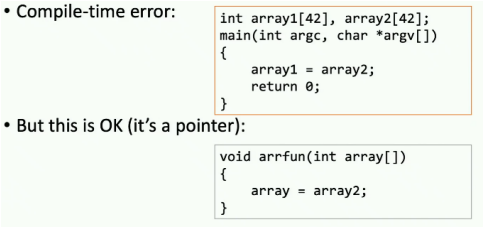
An array name in an expression is treated as a pointer to the first element of the array



Arrays are not pointers. An array is a collection of homogeneous data elements stored at contiguous memory addresses. A pointer is a variable that stores a memory address



In the compiler A[i] is always is rewritten as $*(A+i)$
An array name as a function parameter is a pointers. Arrays cannot be renamed:

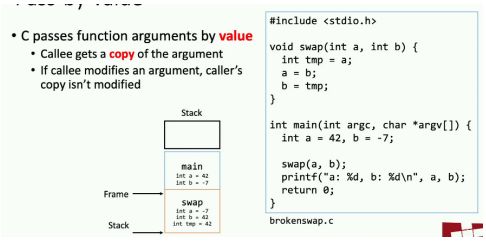


The top one doesnt work, because arrays are not pointers. The bottom one works because in the method array is a parameter hence its a pointer to the first element and the assignment will work. array2 will be treated as the address of the first element of the array.

4.6 Passing by Reference

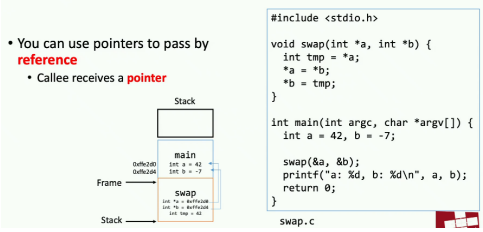
4.6.1 Pass by value

By default in C variables are passed by value. That means a copy of the arguments are made.

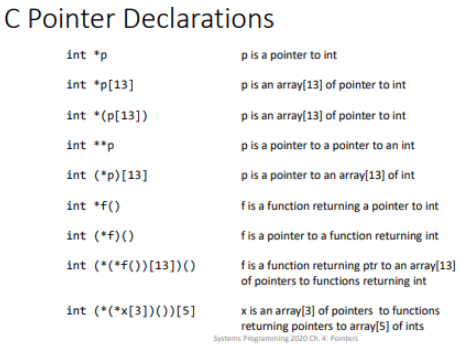


When we return swap the frame is popped from the stack, but the values in main have not been modified.

4.7 Pass by Reference



4.8 C Pointer Declarations



Chapter 5

Dynamic memory allocation

static memory allocation: Memory is allocated when the program is loaded and deallocated when the program exits. (global variables)

```
// a global variable
int counter = 0;

int main(int argc, char *argv[])
{
  counter++;
  return 0;
}
```

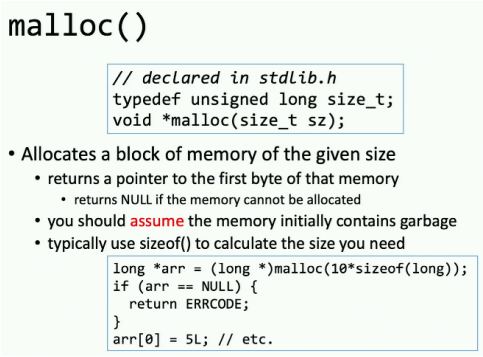
automatic memory allocation: allocated when a function is called and deallocated when function returns. We write C code, this gets compiled into an executable. When the compiler sees procedure calls in the code, in assembly it creates instructions which include push and pop instructions which modify the stack pointer which create and deallocate stack frames.

```
int foo(int a)
{
  int x = a + 1;
  return x
}

int main(int argc, char *argv[])
{
  int y = foo(10);
  return 0;
}
```

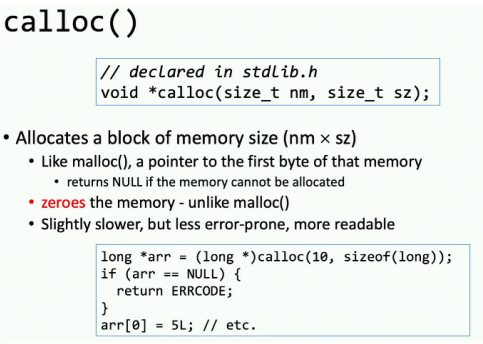
We want memory that is more dynamic e.g persists across multiple function calls, memory too big to fit on the stack,memory allocation for functions with an unknown return size etc.
The stack size is usually 8MB.

Dynamically allocated memory: The program explicitly requests a new block of memory. C requires manual memory management in contrast to other languages which have automatic memory management (garbage collector). In manual memory management the cod explicitly deallocates memory. This gives us much more control over memory but requires much more care.

5.1 The C memory API

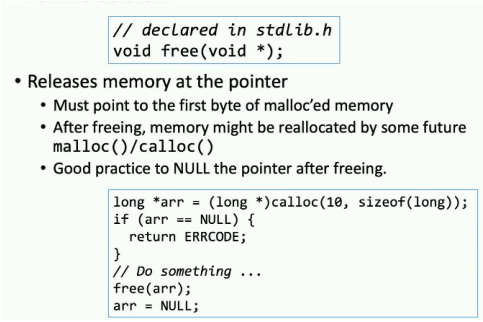


The return of malloc is a pointer to the beginning of the memory allocation. If you request more memory than is available malloc will return null. malloc returns a void pointer so that we can cast it to the appropriate type later. malloc has no guarantee to how the memory is initialized.
The method calloc() zeros out the requested memory



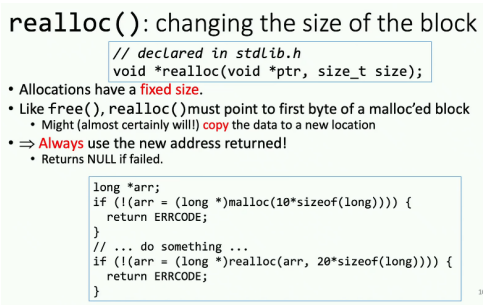
5.1.1 Deallocation

Deallocation is done with the free method. For each malloc/calloc there must be an appropriate free call



Its good practice to null the pointer after freeing it, because after you free it, its not pointing to anything meaningful anymore. Dereferencing it will then have a clear error.

5.1.2 Resize



realloc does not guarantee how the memory region is initialized.

5.1.3 size_t

- `size_t`:
  - An unsigned integer of some size
  - Return type of `sizeof()`
  - Large enough to hold the size of the largest possible array in memory
    ⇒ can be used to store a pointer
- `ptrdiff_t`:
  - An unsigned integer of some size
  - Result of subtracting two pointers
  - Used for array loops, size calculations, etc.
  - Ends up identical to `size_t`

5.1.4 Summary

```c
/* This program simply reads integers
   into a dynamic array until eof.
   The array is expanded as needed */

#include <stdio.h>
#include <stdlib.h>

// Initial array size
#define INIT_SIZE 8

int main(int argc, char *argv[])
{
  int num;     // Num. of integers
  int *arr;    // Array of ints.
  size_t sz;   // Array size
  int m, in;   // Index & input num

  // Allocate the initial space.
  sz = INIT_SIZE;
  arr = (int *)calloc(sz, sizeof(int));
  if (arr == NULL) {
    fprintf(stderr, "calloc failed.\n");
    exit(1);
  }
```

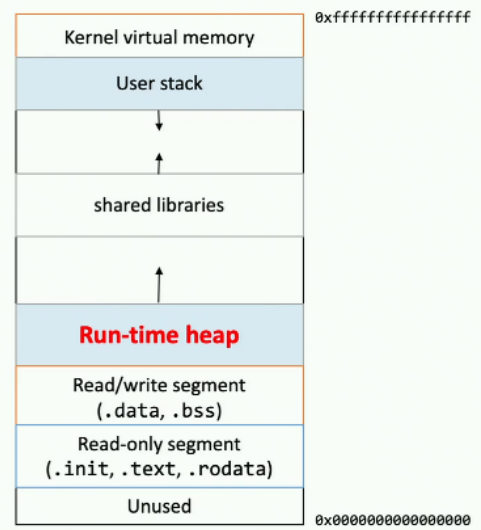
```c
  // Read in the numbers.
  num = 0;
  while(scanf("%d", &in) == 1) {
    // See if there's room.
    if(num >= sz) {
                // There's not.  Get more.
      sz *= 2;
      arr = (int *)realloc(arr, sz*sizeof(int));
      if(arr == NULL) {
        fprintf(stderr, "realloc failed.\n");
        exit(1);
      }
    }
    // Store the number.
    arr[num++] = in;
  }
  // Print out the numbers
  for(m = 0; m < num; ++m) {
    printf("%d\n", arr[m]);
  }
  free(arr); // Always good to free
  return 0;
}
```

calloc.c

the method scanf is used to get user input. It blocks the program until the input is given.

5.2 Managing the heap

The heap ("free store") is a large pool of unused memory, used for dynamically allocated data structures. malloc() allocates chunks of memory in the heap, and maintains bookkeeping data in the heap to track allocated blocks. Free returns the allocated chunks.



5.2.1 Memory Corruption

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
  int a[2];
  int *b = malloc(2*sizeof(int)), *c;

  a[2] = 5;   // assign past the end of an array
  a[0] += 2;  // assume malloc zeroes out memory
  c = b+3;    // mess up your pointer arithmetic
  free(&(a[0])); // pass pointer to free() that wasn't malloc'ed
  free(b);
  free(b);    // double-free the same block
  b[0] = 5;   // use a free()'d pointer

  // any many more!
  return 0;
}
```

5.2.2 Memory Leaks

A memory leak happens when code fails to deallocate memory that will no longer be used.

```
// assume we have functions FileLen,
// ReadFileIntoBuffer, and NumWordsInString

int NumWordsInFile(char *filename)
{
    char *filebuf = (char *)malloc(FileLen(filename)+1);
    if (filebuf == NULL) {
        return -1;
    }

    ReadFileIntoBuffer(filename, filebuf);

    // Leak! we never free(filebuf);
    return NumWordsInString(filebuf);
}
```

Memory footprint: The amount of memory your program uses implications of a leak: For short-lived programs this might be ok, but for long-lived programs this is usually bad and might:

- slow down over time due to VM thrashing
- use up all available memory and crash
- starve other programs out of memory

5.3 Structures and unions

5.3.1 Structured data

Allows us to have a collection of data of the same type in memory. A bit like a class but there are no methods or constructors.

```
struct typename {
    type name;
    type name;
    …
    type name;
};
```

```
// New structured data type called "struct Point"

struct Point {
    int x;
    int y;
};

struct Point origin = { 0, 0 };
```

- Use "." to refer to fields in a struct
- Use "->" to refer to fields through a pointer to a struct

```
struct Point {
  int x, y;
};

int main(int argc, char **argv) {
  struct Point p1 = {0, 0};  // p1 on the stack
  struct Point *p1_ptr = &p1;

  p1.x = 1;
  p1_ptr->y = 2;
  return 0;
}
```

p->x
is nicer than
(*p).x

you can assign the value of a struct from a struct of the same type. This copies the entire contents (unlike for arrays)

```
#include <stdio.h>

struct Point {
  int x, y;
};

int main(int argc, char **argv) {
  struct Point p1 = {0, 2};
  struct Point p2 = {4, 6};

  printf("p1: {%d,%d}  p2: {%d,%d}\n", p1.x, p1.y, p2.x, p2.y);
  p2 = p1;
  printf("p1: {%d,%d}  p2: {%d,%d}\n", p1.x, p1.y, p2.x, p2.y);

  return 0;
}
```

Even if the struct has an array in it the contents will be copied over by the assignment operator. This is because the copy operations copies the region of memory for that struct and doesnt care about the types.
Structs can be passed as arguments, by default they are passed by value, to pass by reference we pass a pointer to the struct. We can also return structs (right picture)

```c
#include <stdio.h>

struct point {
  int x, y;
};

void DoubleXBroken(struct point p) {
  p.x *= 2;
}

void DoubleXWorks(struct point *p) {
  p->x *= 2;
}

int main(int argc, char *argv) {
  struct point a = {1,1};

  DoubleXBroken(a);
  printf("(%d,%d)\n", a.x, a.y);

  DoubleXWorks(&a);
  printf("(%d,%d)\n", a.x, a.y);

  return 0;
}
```

```c
// a complex number is a + bi
struct complex {
  double real;  // real component
  double imag;  // imaginary component
};

struct complex AddComplex(struct complex x, y)
{
  struct complex retval;

  retval.real = x.real + y.real;
  retval.imag = x.imag + y.imag;
  return retval;
}

struct complex MultiplyComplex(struct complex x, y)
{
  struct complex retval;

  retval.real = (x.real * y.real) - (x.imag * y.imag);
  retval.imag = (x.imag * y.real) - (x.real * y.imag);
  return retval;
}
```

5.3.2 Unions

Like a struct but only allocates enough memory to hold the largest member.

```c
union u {
    int ival;
    float fval;
    char *sval;
};
union u my_uval;
```

Only use a union if youre interested in only one of the members.

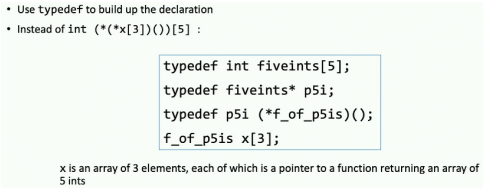
5.4 Type definitions

5.4.1 typedef

Introduces a new type definition i.e a new name for a type, hence it can be used wherever the original type could be.

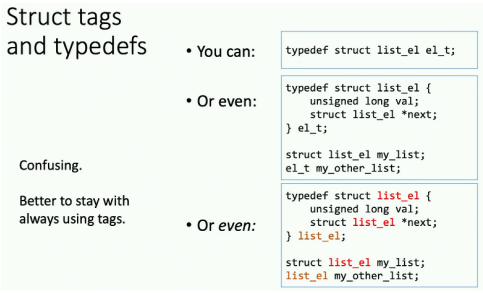
```c
typedef unsigned uint32_t;
uint32_t ui;
…
typedef int **myptr;
int *p;
myptr mp = &p;
…
typedef struct skbuf skbuf_t;
skbuf_t *sptr;
```

This helps us avoiding complex declarations e.g:



5.4.2 Struct tags and typedefs

Struct tags are the names that we give the structs e.g point in struct point{...}.



5.4.3 C namespaces

There are 4 namespaces in C:

- Label names (goto...)
- Tags (one namespace for all struct, union,enum's)
- Member names (one namespace for each struct union,enum)
- Everything else (including typedef)

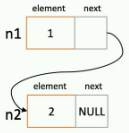
```c
// structure tag
struct id {
  int id;  // structure member
};

// different structure
struct id2 {
  char id; // structure member
};

// ordinary identifier
void id() {
  goto id;

id: // label

}
```

5.5 Dynamic Data Structures

5.5.1 Singly-Linked List

A list consisting of nodes. Each node in the list contains some element as its payload and a pointer to the next node in the linked list. The last node in the list contains NULL instead of the next pointer. We can implement this using structs. We represent each node as a struct.

```c
#include <stdio.h>

struct node {
  int element;
  struct node *next;
};

int main(int argc, char **argv) {
  struct node n1, n2;

  n1.element = 1;
  n1.next = &n2;
  n2.element = 2;
  n2.next = NULL;
  return 0;
}
```

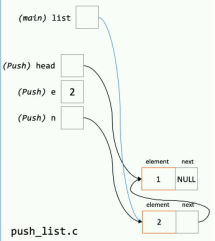


manual_list.c

We create a method push() which dynamically adds nodes to the list.

Push()

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct node_st {
  int element;
  struct node_st *next;
};

struct node *Push(struct node *head, int e) {
  struct node *n =
    (struct node *)malloc(sizeof(struct node));

  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;

  return n;
}

int main(int argc, char **argv) {
  struct node *list = NULL;

  list = Push(list, 1);
  list = Push(list, 2);

  return 0;
}
```



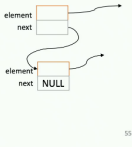
push_list.c

We notice that the above implementation of push() has a leak as we have not deallocated the malloc(). We can check if the program has a leak in the terminal with (assuming our file is called *push_list.c*)

$ gcc -o push_list -g -Wall *push_list.c*
$ valgrind -leak-check=full *./push_list.c*

We can create a generic linked list by using "void *" as a type.

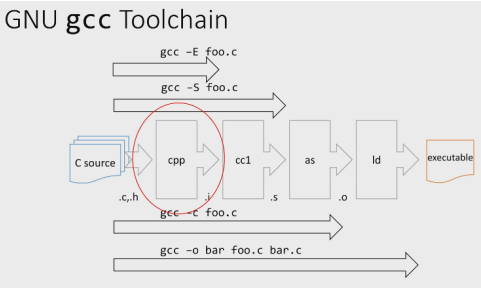
```c
struct node {
  void *element;
  struct node *next;
};

struct node *Push(struct node *head, void *e) {
  struct node *n = (struct node *)malloc(sizeof(struct node));

  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;

  return n;
}
```



Chapter 6

Wrapping up C

6.1 The C Preprocessor



The cpp is not part of the C language but rather an extra language that sits on top.

```
#include <file1.h>

#include "file2.h"
```

- Include the "header" file inline in the source code
- Basic mechanism for defining APIs
- Use of <> or "" determines where to look for the file
  - Use <> for system headers
  - Use "" for your own headers
- Included files can include other files
  - Beware of including files twice!

The cpp will go look for the files in the include statement. Hence using $<, >$ *or*", " will tell the cpp where to start looking. Since included files can include other files we could get a cyclic dependance.

6.1.1 Macro Definitions

Macro definitions

```
#define FOO BAZ
#define BAR(x) (x+3)

…

#undef FOO
```

- Token-based macro substitution
- Any subsequent occurrence of `FOO` is replaced with `BAZ`
  - Until a `#undef FOO`
- `BAR(4)` is replaced with `(4+3)`
  - Not 7!
- `BAR(hello)` is replaced with `(hello+3)`
- `QUX` is replaced with `1`

C macros purely work on text. The default definition of any macro is 1. **Preprocessor conditionals:**

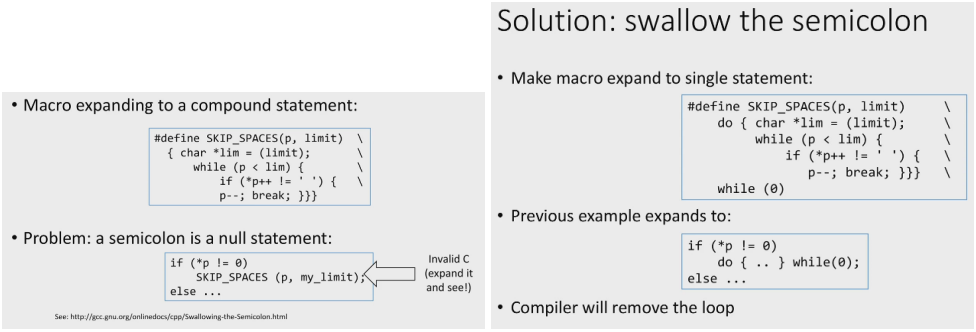
```
#if expression
… (text 1)
#else
… (text 2)
#endif
```

- Text 1 is used if the expression evaluates to "true" (non-zero), otherwise Text 2
  - `#else` is optional
  - `#elif` can be used as well
  - Expression must be literals and macros only!

```
#ifdef FOO
… (text 1)
#else
… (text 2)
#endif

#ifndef BAR
… (text 1)
#else
… (text 2)
#endif
```

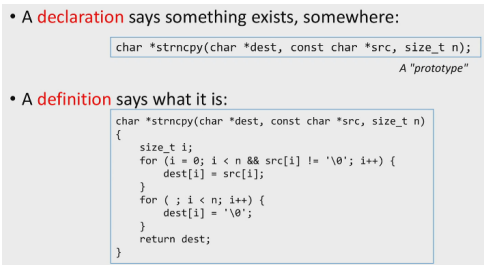
- Text 1 is used if a macro `FOO` is defined, otherwise Text 2
  - Opposite for BAR
  - `#else` is optional
- Shorthand for
  - `#if defined(FOO)`
  - `#if !defined(BAR)`

In C we can use a backslash to continue the code on the next line. This is used for readability



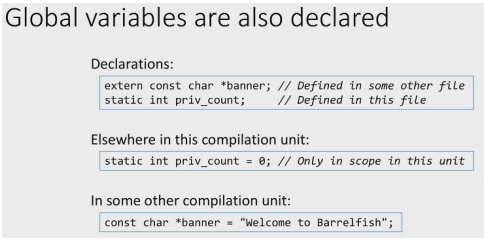
6.2 Modularity

In C there is a difference between Declarations and Definitions:



Compilation Units: What gets fed into the C compiler (cc1) i.e compilation unit is what comes out of the cpp meaning the c file after all the substitutions have been made.
Declarations can be annotated with:

- **static:** definition (also static) is in this compilation unit, and cant be seen outside it
- **extern:** definition is somewhere else, either in this compilation unit or another.



A Module is a self contained piece of a larger program. It consists of:

- Externally visible:
  - functions to be invoked
  - typedefs and perhaps global variables
  - cpp macros
- Internal functions types, global variables that clients shouldnt look at

Modularity in C is done using header files:

C header files

- Specify interfaces with header files
- Module `foo` has interface in `foo.h`
  - Clients of foo `#include "foo.h"`
  - foo.h contains no definitions
    - only external declarations
- Implementation typically in `foo.c`
  - Also includes `foo.h`
  - Contains no external declarations
    - only definitions and internal declarations

There is cpp idom for header files to avoid a cyclic dependancy:

"file.h":

```
#ifndef __FILE_H_
#define __FILE_H_

… (declarations, macros)

#endif // __FILE_H_
```

- cpp boilerplate ensures file contents only appear once
- Never `#include` a .c file!

the $\#ifndef$ ensures that the body will only be used once in the file.
Example: Linked List:

ll.c

```
#include "ll.h"

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct node *Push(struct node *head, void *element) {
  … implementation here …
}
```

ll.h

```
struct node {
  void *element;
 struct node *next;
};

extern struct node *Push(struct node *head, void *element);
```

example_ll_customer.c

```
#include "ll.h"

int main(int argc, char **argv) {
  struct node *list = NULL;
  char *hi = "hello";
  char *bye = "goodbye";

  list = Push(list, (void *) hi);
  list = Push(list, (void *) bye);
  return 0;
}
```


6.3 Function Pointers

A pointer with the address of some code which corresponds to that function. Calling a function pointer, calls whatever function that pointer is pointing to.

```
int (*func)(int *, char);
```

- "`func` is a pointer to a function which takes two arguments, a pointer to `int` and a `char`, and returns an `int`"
- Rare in OO languages,
  - but c.f. C# "delegates"; Eiffel "agents"
- As with all types, can be used with `typedef`
- Basis for lots of techniques in Systems code

From the Linux kernel…

```
struct file_operations {
    ...
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ...
};
```

- Record (struct) type of function pointers
  - Sometimes called a *vtable*
  - Each member is a different function type
  - Implements a "method" (or "feature") of some "object"
    - In this case, a `struct file`
  - Provides polymorphism (e.g. files and sockets)
- Not really objects: no protection or hiding
  - But does provide the reuse benefits

6.4 Assertions

6.4.1 Assertions

assert(*<scalar expression>*);

- At run time, evaluate the expression.
- If true, do nothing
- If false:
  - Print "*file.c:line*: *func*: Assertion `*expr*' failed."
  - Abort (dump core)

Where true is a non-zero value.
Example:

Assertion example

```
#include <assert.h>

// It's a bug to call this with null a or b
void array_copy(int a[], int b[], size_t count)
{
    int i;
    assert(a != NULL);
    assert(b != NULL);
    for(i=0; i<count; i++) {
        a[i] = b[i];
    }
}

int main(int argc, char *argv[])
{
    // This is not going to go well...
    array_copy(NULL, NULL, 0);
    return 0;
}
```

```
$ ./a.out
a.out: assert_text.c:8: array_copy: Assertion `a != ((void *)0)' failed.
Aborted
$
```

Assert is a controlled crash, hence it is of no use to the user. Assertions are for programmers to find bugs, not for programs to detect erros.

6.5 goto

goto jumps to a specific label in a program.

6.5.1 When to use goto:

DONT,however...it can be useful in:

- Early termination of nested loops

Early termination of nested loops

```
int a[MAXROW][MAXCOL]

  int i,j;
  for(i=0; i< MAXROW; i++) {
      for(j=0; j < MAXCOL; j++) {
          if (a[i][j] == 0) {
              goto found;
          }
      }
  }
  <do some processing when not found>
found:
  <do some processing on i and j>
```

Compare:

```
int a[MAXROW][MAXCOL]

int i,j;
bool found=false;
for(i=0; i< MAXROW; i++) {
    for(j=0; j < MAXCOL; j++) {
        if (a[i][j] == 0) {
            found=true;
            break;
        }
    }
    if (found)
        break;
}
if (found) {
    <do processing on i and j>
} else {
    <do processing when not found>
}
```

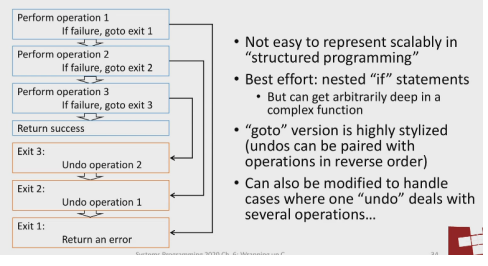
- But consider simply using `return()`
- Other languages have `break(label)`

- Cleanup conditions

Cleanup conditions

Goto is used for *recovery code*

- General idea:
  - Code performs a sequence of operations
  - Any one can fail
  - If one fails, all previous operations must be ***undone***
- Canonical example:
  - Malloc'ing a sequence of buffers for data
  - If one fails, must free all previous generated buffers
- Code is often (not always) auto-generated



- Not easy to represent scalably in "structured programming"
- Best effort: nested "if" statements
  - But can get arbitrarily deep in a complex function
- "goto" version is highly stylized (undos can be paired with operations in reverse order)
- Can also be modified to handle cases where one "undo" deals with several operations...

6.6 setjmp() and longjmp()

setjmp()

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

- `setjmp(env)`:
  - Saves the current stack state / environment in **env**
  - Returns 0.

longjmp()

```
#include <setjmp.h>

void longjmp(jmp_buf env, int val);
```

- `longjmp(env,val)`:
  - Causes *another* return to the point saved by env
  - This new return returns `val` (or 1 if `val` is 0)
  - This can only be done *once* for each `setjmp()`
  - It is invalid if the function containing the `setjmp` returns

Example:

Toy example

```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void second(void) {
    printf("second\n");
    longjmp(buf,1);
}

void first(void) {
    second();
    printf("first\n");          // does not print
}

int main() {
    if ( ! setjmp(buf) ) {
        first();                // when executed, setjmp returns 0
    } else {                    // when longjmp jumps back, setjmp returns 1
        printf("main\n");       // prints
    }

    return 0;
}
```

Output:
second
main

This can be used for Coroutines

6.7 Coroutines

A way for functions to call eachother without doing subroutine calls. This allows us to have 2 functions running at the same time.

Creation/deletion

```
struct Coro *Coro_new(void)
{
    struct Coro *self = (struct Coro *)calloc(1, sizeof(struct Coro));
    self->stack = (void *)calloc(1, 65536 + 16);
    return self;
}
```

```
void Coro_free(struct Coro *self)
{
    free(self->stack);
    free(self);
}
```

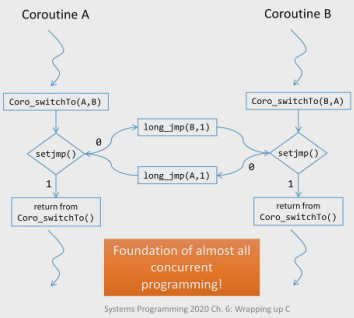
Switching

```
void Coro_switchTo(struct Coro *self,
                   struct Coro *next)
{
    if (setjmp(self->env) == 0) {
      longjmp(next->env, 1);
    }
}
```

- First return from `setjmp()`:
  ⇒ `longjmp` to the next coroutine
- Second return:
  ⇒ continue where we left off

This allows us to implement a socalled Context switch:

The context switch



Systems Programming 2020 Ch. 6: Wrapping up C

But how do we start?

Initialization (the hard bit)

```
struct Coro_Cl {
    void *context;
    Coro_func_t *func;
};
```

This is a ***closure***: a function bundled with a set of arguments.

```
void Coro_Start_(struct Coro_Cl *block)
{
    (block->func)(block->context);
    printf("Scheduler error: returned from coro start!\n");
    exit(-1);
}
```

Initialization (the hard bit)

```
void Coro_start(struct Coro *self, struct Coro *other,
                void *context, Coro_func_t *callback)
{
        struct Coro_Cl sblock;
        struct Coro_Cl *block = &sblock;
        block->context = context;
        block->func    = callback;

        setjmp(self->env);
        self->env[0].__jmpbuf[6] = ((unsigned long)(self->stack));
        self->env[0].__jmpbuf[7] = ((long)Coro_Start_);
        Coro_switchTo(self, other);
}
```

Machine-dependent hack:
6 ⇒ %rsp
7 ⇒ %rip

Now: replace `emit()` and `getchar()` in original code with `Coro_switchTo()` and you're done.

Chapter 7

Implementing dynamic memory allocation

In order to run a java program we need alot of stuff which actually isnt written in java. The Java runtime contains memory allocator,memory manager, garbage collector and a low level I/O routine much of which is written in C/C++. The java compiler generates byte code. The Java trace compiler will translate that into efficient machine code this is the JAVA VM. In C this doesnt exist, there is a C library, but it is not needed to run a C program. Hence this is why C is good for writing a Operating Systems.

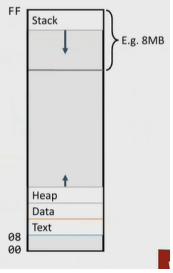
7.1 Dynamic Memory allocation

The Memory allocator gives out memory to applications in blocks. It sits inbetween the application itself which calls malloc and allocates memory out of some area of the virtual address space which has been allocated to it inturn by the OS, when the process starts it gets allocated to a certain amount of memory (the heap). malloc allocates memory out of the heap in arbitrary sized units and hands it to the application and frees it after use. Recall the malloc package:

```
#include <stdlib.h>
void *malloc(size_t size)
    Successful:
        Returns a pointer to a memory block of at least size bytes
        (typically) aligned to 8- or 16-byte boundary
        If size == 0, returns NULL
    Unsuccessful: returns NULL (0) and sets errno
void free(void *p)
    Returns the block pointed at by p to pool of available memory
    p must come from a previous call to malloc() or realloc()
void *realloc(void *p, size_t size)
    Changes size of block p and returns pointer to new block
    Contents of new block unchanged up to min of old and new size
    Old block has been free()'d (logically, if new != old)
```

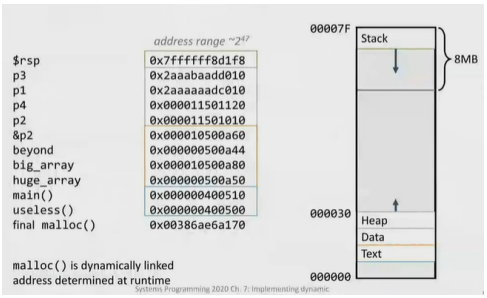
Memory allocation example:

```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
 /* Some print statements ... */
}
```

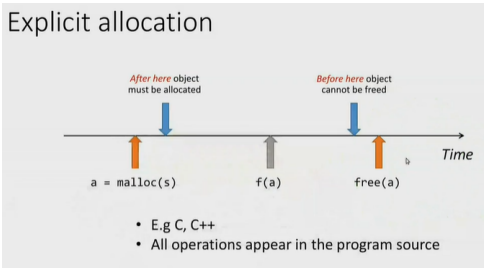


Where does everything go?

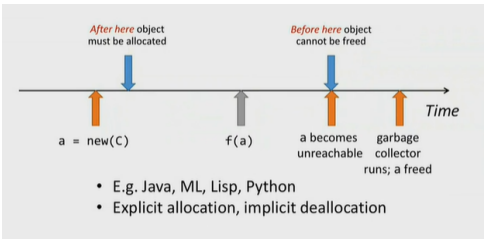
The bottom is empty because we want to have the Null value. Heap is the part that malloc uses, the heap grows upwards when more and more memory is used. The Stack starts at the top of the memory and grows down. The Data segment is data the compiler knows exists. Text contains the machine code



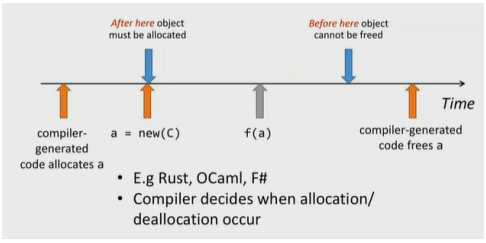
7.1.1 Explicit allocation



The blue arrow on the right is the earliest point in which the memory can be freed i.e it will not be used any more in the program, the orange arrow on the right is when the program calls free. The blue arrow on the left is the latest possible point in which the memory must be malloced.
In contrast garbage collection has the following timeline



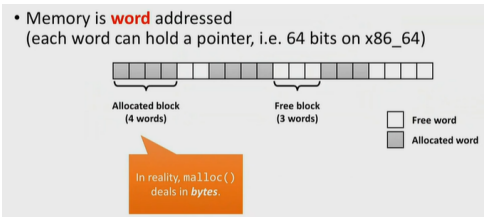
Garbage collection makes it easier to code but decreases performance.
There is a third approach is Compiler support which is something inbetween the above approaches. The compiler looks at the program and figures out that an object is needed, because of the function call. The compiler can prove when the object is no longer needed.



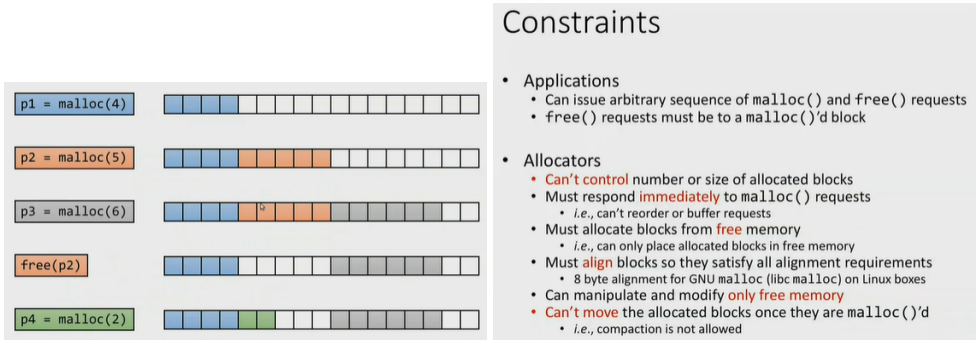
Compiler support cant be used for arbitrary programs and hence have a garbage collection which kicks in

7.2 The Explicit memory allocation Problem

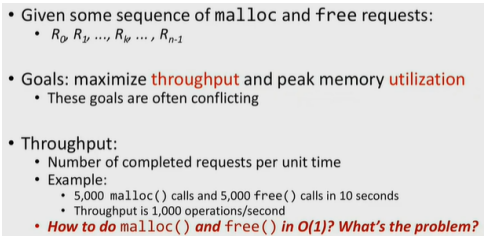
Assumptions



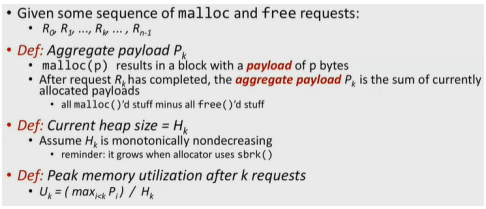
Allocation Example:



7.2.1 Performance goal: Throuput



7.2.2 Performance goal: Peak memory utilization

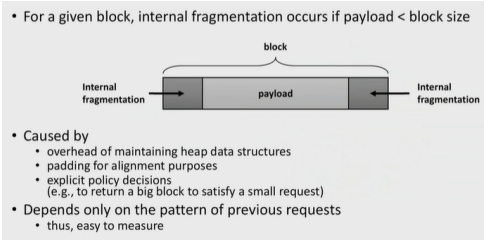


7.2.3 Fragmentation

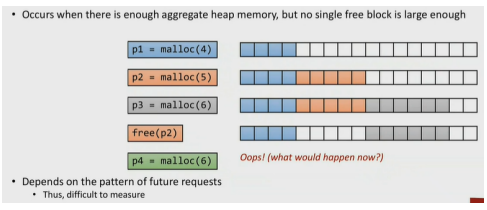
The heap gets into a state where it becomes hard to efficiently use it. We distinguish between:

- internal fragmentation
- external fragmentation

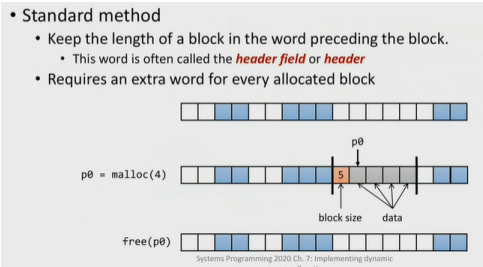
Internal fragmentation:



External fragmentation:

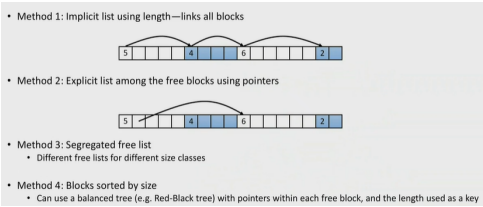


7.2.4 How to free blocks

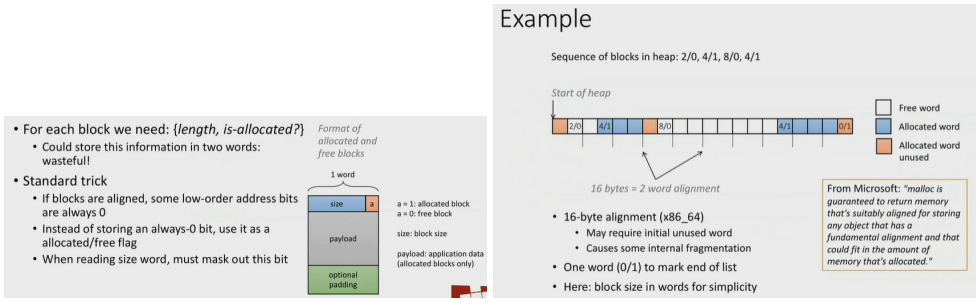


this method introduces internal fragmentation

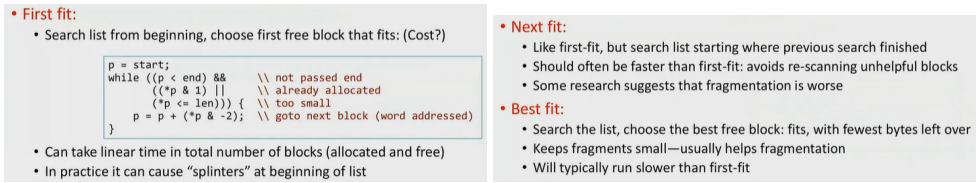
7.2.5 Keeping track of free blocks



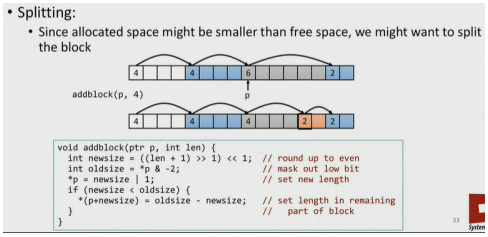
- **Implicit List**



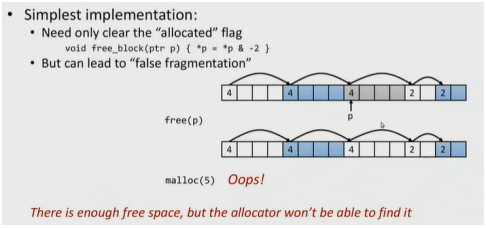
It is the payload which is aligned. 16 byte alignment makes it more efficent for the hardware to access values which are 16 bytes in size. A 16 byte aligned value is guaranteed not to span 2 memory pages and not to span 2 cache lines The start of the heap is a word that never gets returned to the user because when we allocate the very first block of the heap, the first word is used to hold the size and the flag, which means what were returning is a pointer to the second word. A size of 0 indicates the end of the heap. finding a free block



Allocating a free block

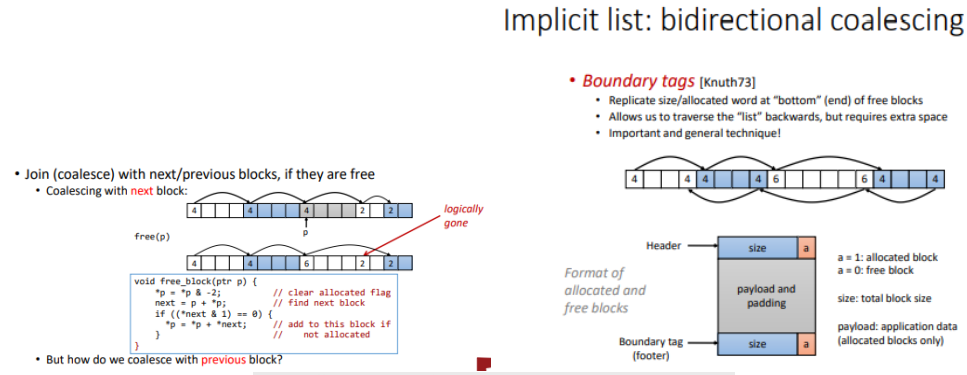


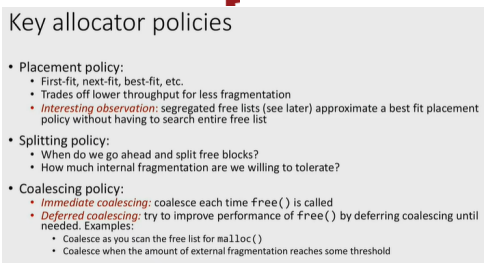
Freeing a block



7.3 Coalescing

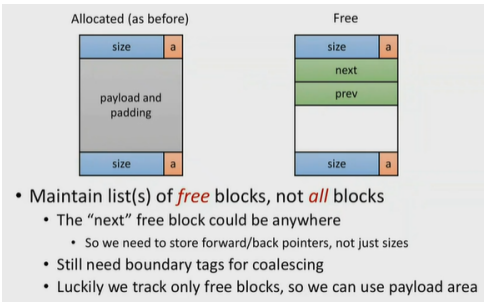
Transforming smaller blocks into fewer larger blocks to reduce external fragmentation



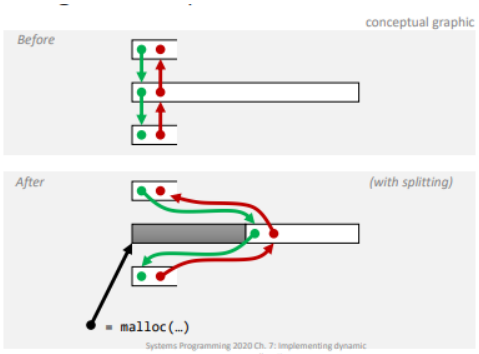


7.4 Explicit free lists

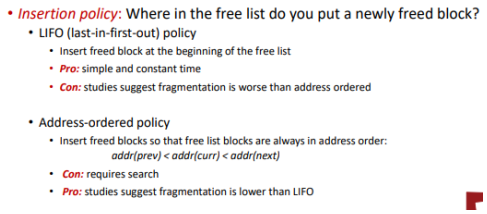
7.4.1 concept



7.4.2 Allocation



7.4.3 freeing

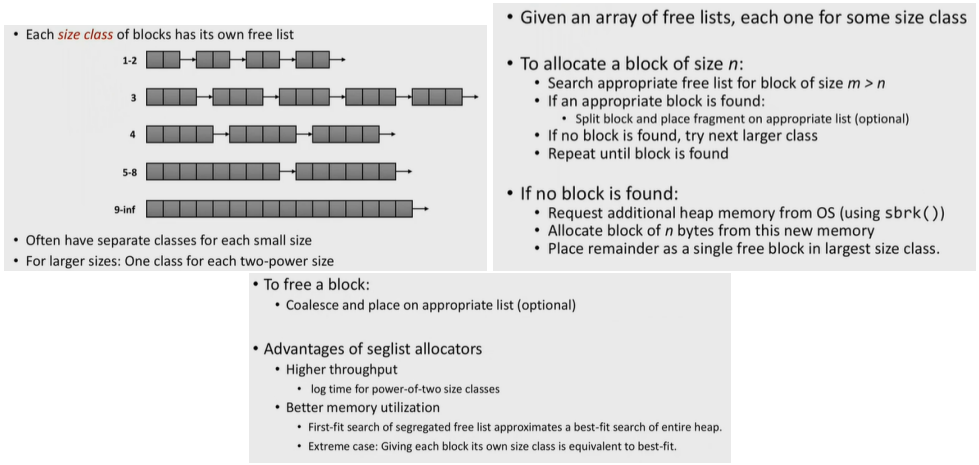


7.4.4 summary

Allocation is linear time in number of free blocks which is much faster than implicit lists when most of the memory is full. Its slightly more complicated to allocate and free since we need to splice blocks in and out of the list. Some extra space needed for the pointers.

7.5 Segregated free lists

Builds on the explicit free list. The idea is that we have a list for each size class of blocks:

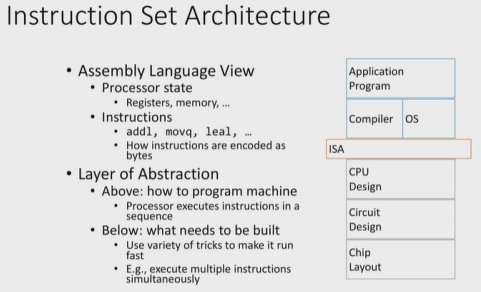


Chapter 8

Basic x86 architecture

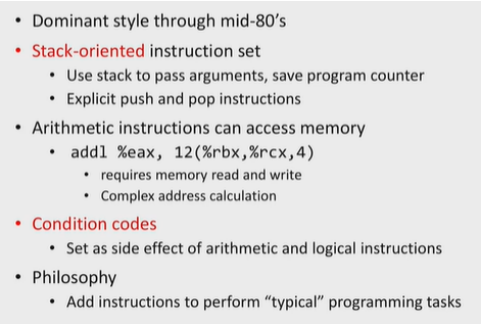
8.1 What is an instruction set architecture

Architecture:(also instruction set architecture:ISA) The parts of a processor design that one needs to understand to write assembly code (e.g instruction set specification, registers)



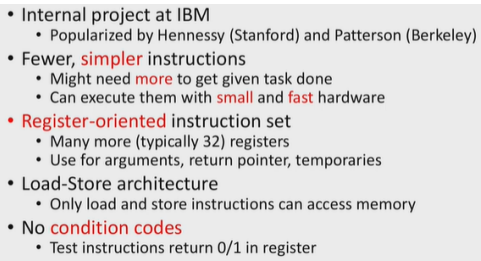
Microarchitecture: Implentation of the architecture (cache sizes and core frequency)

8.1.1 CISC: Complex Instruction Set



8.1.2 RISC: Reduced Instruction Set

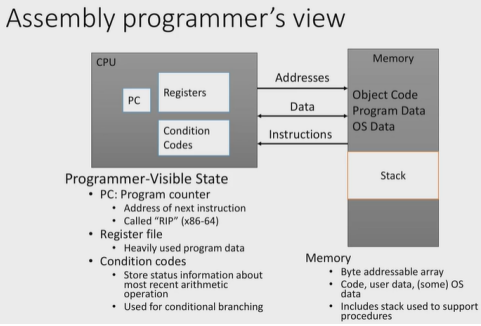
Idea was to have fewer easier instructions and give more work to the compiler. This lead to more space available for cache aswell as better compiler optimization.



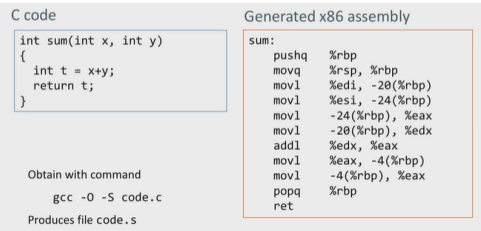
8.2 Basics of machine code

There are two common ways to write $x86$ Assembler:

- *AT&T* syntax (What we will use)
- Intel syntax: Generally used for Windows machines



8.2.1 Compiling into assembly



8.2.2 Assembly data types

Integer data of 1,2,4 or 8 bytes which hold data values or addresses (untyped pointer)
Floating point data of 4,8 or 10bytes
No aggregate types(arrays,structures,...) only coniguously allocated bytes in memory

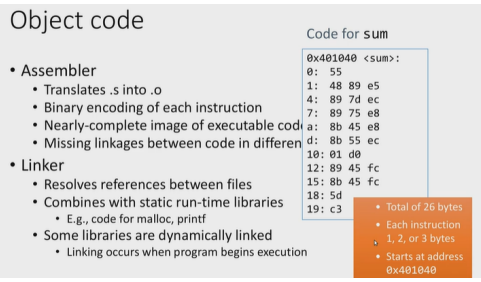
8.2.3 Assembly code operations

We can perform arithmetic function on register or memory data.
Transfer data between memory and register i.e

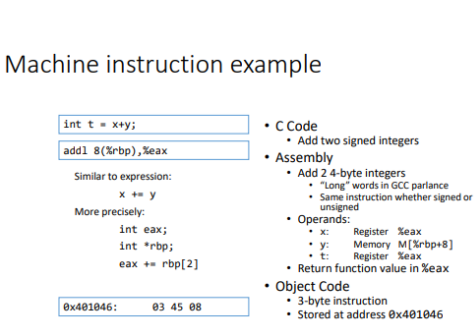
- Load data from memory into register
- Store register data into memory

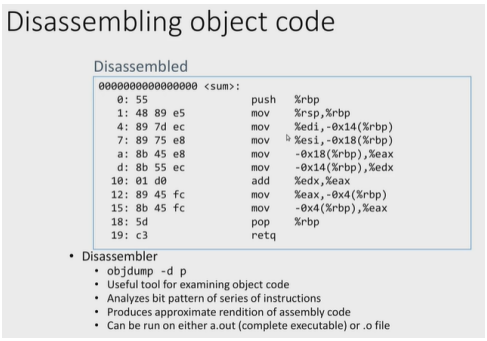
Transfer control i.e Unconditional jumps to/from procedures and conditional branches

8.2.4 Object Code



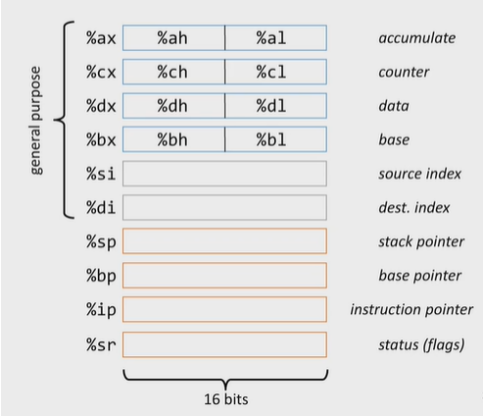
8.2.5 Machine instruction example





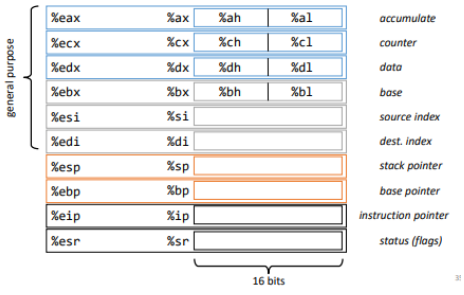
8.3 x86 architecture

8.3.1 8086 Registers



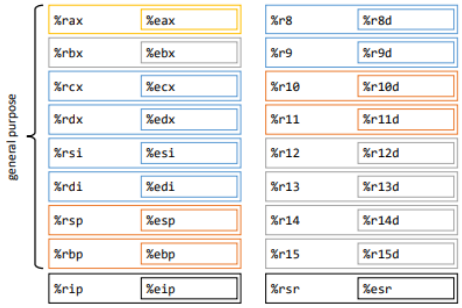
The source/destination index registers were limited by the instruction set to only hold adresses.
The stack pointer was a register only used for the stack.
The only general purpose were the top 4 registers which are divided into lo and hi (remnants of the previous 8-bit design)

8.3.2 80386 Registers



The 8086 was extended by 16 bits, they kept the old names to refer to the halfs and added e to the names to indicate the extended register.

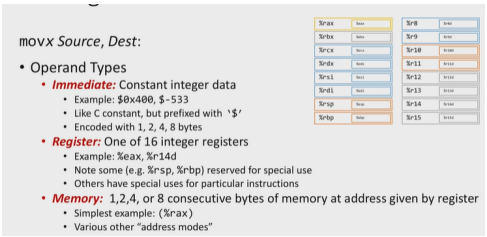
8.3.3 x86-64 integer registers



Extension of the 386 by 32 bits with prefix now r. They added 8 more registers. all registers are now general purpose.

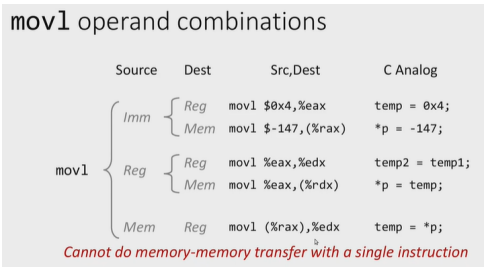
8.3.4 Moving data

- movx *Source, Dest*
  - x in {b, w, l,q}
  - movq *Source, Dest*:
    Move 8-byte "quad word"
  - movl *Source, Dest*:
    Move 4-byte "long word"
  - movw *Source, Dest*:
    Move 2-byte "word"
  - movb *Source, Dest*:
    Move 1-byte "byte"

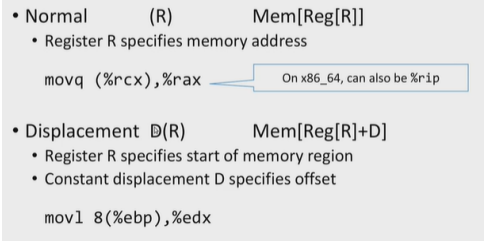

movx *Source, Dest*:

- Operand Types
  - ***Immediate:*** Constant integer data
    - Example: `$0x400, $-533`
    - Like C constant, but prefixed with '$'
    - Encoded with 1, 2, 4, 8 bytes
  - ***Register:*** One of 16 integer registers
    - Example: %eax, %r14d
    - Note some (e.g. %rsp, %rbp) reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 1,2,4, or 8 consecutive bytes of memory at address given by register
    - Simplest example: (%rax)
    - Various other "address modes"

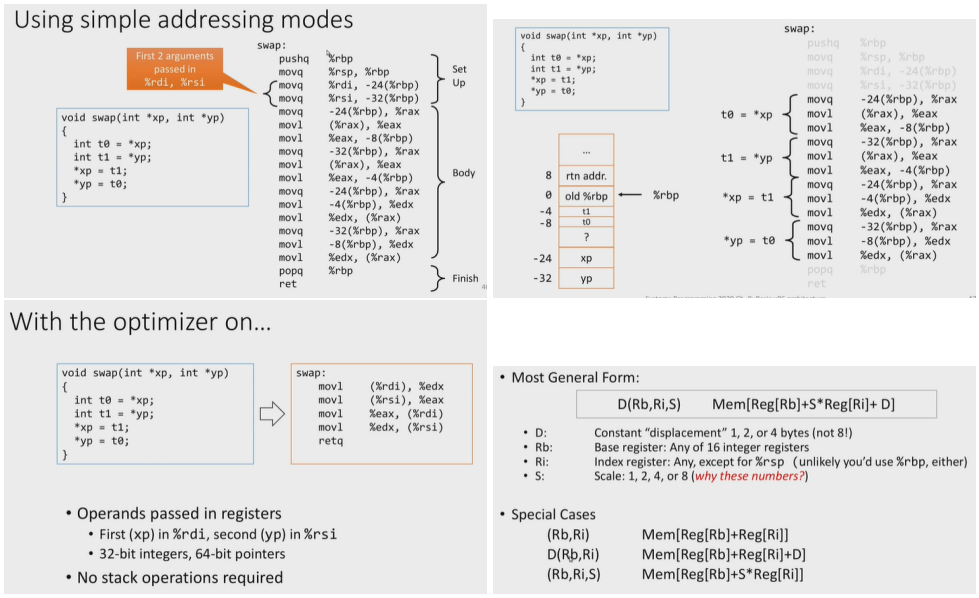
Possible combinations



8.3.5 Simple memory addressing modes



Since we can use the instruction pointer as a source, code can be executed anywhere in memory. Displacement is a register offset. **Example:**



The most general form allows us to perform tricks e.g array indexing, accessing fields in a struct, accessing fields in an array of structs. S is one of those sizes depending on the data type being used.

8.3.6 Address Computation

Address computation is done with the lea function:

$$leaSrc, Dest$$

where Src is the address mode expression and Dest gets set to the address denoted by the expression. With this function we can compute addresses without a memory reference or compute arithmetic expressions of the form $x + k \cdot y, k \in \{1, 2, 4, 8\}$

8.4 x86 integer arithmetic

8.4.1 ordinary arithmetic operations

• Two-operand instructions (longword variants):

Mnemonic	Format	Computation	
addl	Src,Dest	Dest ← Dest + Src	
subl	Src,Dest	Dest ← Dest - Src	
imull	Src,Dest	Dest ← Dest * Src	
sall	Src,Dest	Dest ← Dest << Src	Also called shll
sarl	Src,Dest	Dest ← Dest >> Src	Arithmetic
shrl	Src,Dest	Dest ← Dest >> Src	Logical
xorl	Src,Dest	Dest ← Dest ^ Src	
andl	Src,Dest	Dest ← Dest & Src	
orl	Src,Dest	Dest ← Dest | Src	

• No distinction between signed and unsigned int (why?)

There is no distinction between signed and unsigned values. The processor doesnt have much of a concept of the difference, its just a register, just a 64-bit value. Hence none of these operations imply signed or unsigned computation because their results are the same.

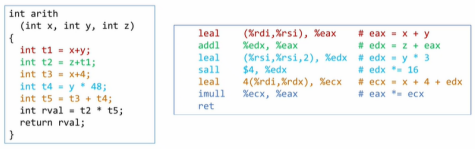
8.4.2 Single operand instructions

• One operand instructions

Mnemonic	Format	Computation
incl	Dest	Dest ← Dest + 1
decl	Dest	Dest ← Dest - 1
negl	Dest	Dest ← -Dest
notl	Dest	Dest ← ~Dest

• See book for more instructions

8.4.3 Using leal for arithmetic expressions



```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

```
leal    (%rdi,%rsi), %eax     # eax = x + y
addl    %edx, %eax            # edx = z + eax
leal    (%rsi,%rsi,2), %edx   # edx = y * 3
sall    $4, %edx              # edx *= 16
leal    4(%rdi,%rdx), %ecx    # ecx = x + 4 + edx
imull   %ecx, %eax            # eax *= ecx
ret
```

The following happens in the lea code lines:

1. rdi and rsi are the first arguments which are x and y respectively. lea takes rdi as the base and rsi as the index i.e it adds the two together. The result is put into eax.
2. takes y, adds it to y multiplied by 2 and stores it in edx, hence we have a multiplication by 3. There is no instruction on a typical RISC machine which multiplies a number by 3. We multiply by 3 because later we need a multiplication by 48 which is 3*16. A multiplication by 16 can easily be computed a left shift.
3. we add rdi to rdx and add 4. Hence we are doing 2 instructions in one with lea

8.5 condition codes

Extra bits in the processor state. Can be thought of as 1-bit registers. They are contained in the status register.

- Single bit registers
  - CF Carry Flag (for unsigned)
  - ZF Zero Flag
  - SF Sign Flag (for signed)
  - OF Overflow Flag (for signed)
- Implicitly set (think of it as *side effect*) by arithmetic operations
  Example: addl/addq Src,Dest ↔ t = a+b
  - **CF set** if carry out from most significant bit (unsigned overflow)
  - **ZF set** if t == 0
  - **SF set** if t < 0 (as signed)
  - **OF set** if two's complement (signed) overflow
    (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
- ***Not*** set by lea instruction
- Full documentation link on course website

8.5.1 Compare Instruction

- Explicit Setting by Compare Instruction

`cmpl/cmpq` *Src2,Src1*

`cmpl b,a` like computing `a - b` without setting destination

CF set if carry out from most significant bit
(used for unsigned comparisons)
ZF set if `a == b`
SF set if `(a-b) < 0` (as signed)
OF set if two's complement (signed) overflow:
`(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

8.5.2 Test Instruction

- Explicit Setting by Test instruction

`testl/testq` *Src2,Src1*

`testl b,a` like computing `a & b` without setting destination

- Sets condition codes based on value of *Src1* & *Src2*
- Useful to have one of the operands be a mask

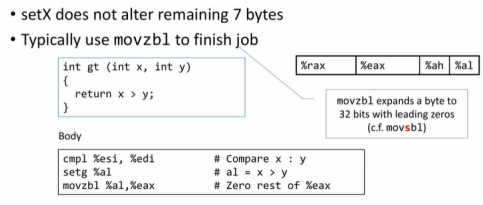
ZF set when `a & b == 0`
SF set when `a & b < 0`

8.5.3 SetX Instructions

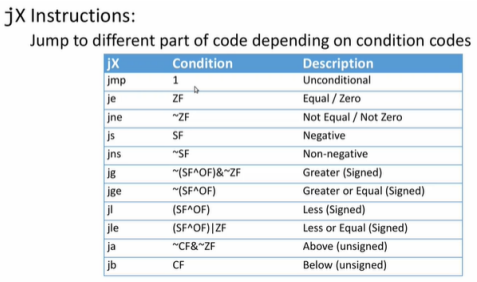
- SetX Instructions
  - Set single byte based on combinations of condition codes

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

The result of the computations get set into the low byte of some other register. If setting the value on a lower order of bits of a register than the top bits are set to 0 e.g if we set eax to a value, then rax gets the same value and we zero out the top bits., Example:



8.5.4 Jump Instructions

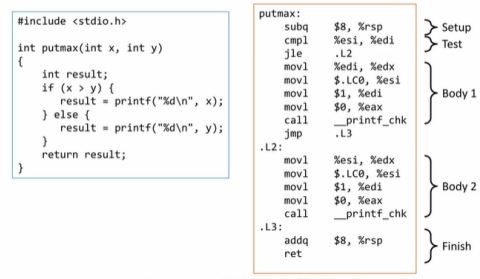


jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Non-negative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

Chapter 9

Compiling C control flow

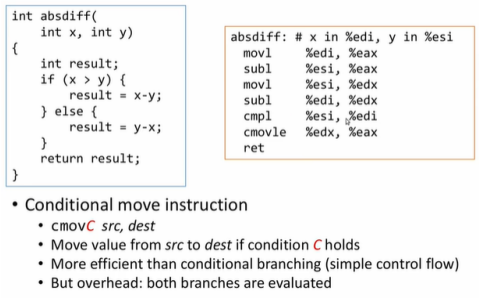
9.1 if-then-else statements



Body 1 is the case where x¿y, Body 2 is x¡= y.

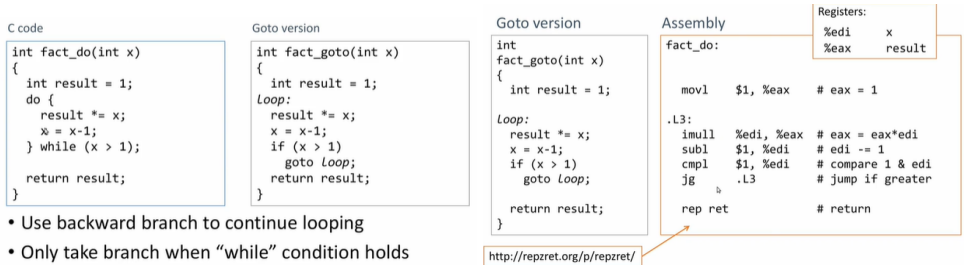
9.1.1 Conditional move

In some cases the compiler can avoid using branch statements and use the so called conditional move instruction:



Conditional move is more efficient than branching.

9.2 do-while loops



9.3 while loops

C code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

Goto version

```
int fact_while_goto2(int x)
{
  int result = 1;
  goto middle;
loop:
  result *= x;
  x = x-1;
middle:
  if (x > 1)
    goto loop;
  return result;
}
```

- More recent technique
- First iteration jumps over body computation within loop

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x--;
  };
  return result;
}
```

```
# x in %edx, result in %eax
  jmp   .L34       #   goto Middle
.L35:              # Loop:
  imull %edx, %eax #   result *= x
  decl  %edx       #   x--
.L34:              # Middle:
  cmpl  $1, %edx   #   x:1
  jg    .L35       #   if >, goto Loop
```

9.4 for loops

A for loop is a while loop with an initialisation instruction.

"For" loop example

```
int result;
for (result = 1; p != 0; p = p>>1)
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

General Form

```
for (Init; Test; Update)
    Body
```

Test	Init	Update	Body
p != 0	result = 1	p = p >> 1	{ if (p & 0x1) { result *= x; } x = x*x; }

For version

```
for (Init; Test; Update )
    Body
```

Goto version

```
  Init;
  goto middle;
loop:
  Body
  Update ;
middle:
  if (Test)
    goto loop;
done:
```

```
for (result = 1; p != 0; p = p>>1)
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

```
  result = 1;
goto middle;
loop:
  if (p & 0x1)
    result *= x;
  x = x*x;
  p = p >> 1;
middle:
  if (p != 0)
    goto loop;
done:
```

9.4.1 Summary of loop variants

General form with conditional move

C code

```
val = Test ? Then-Expr : Else-Expr;
```

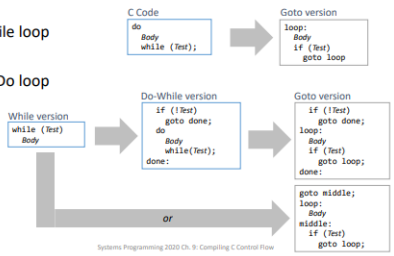
Conditional move version

```
val1 = Then-Expr;
val2 = Else-Expr;
val1 = val2 if !Test;
```

- Both values get computed
- Overwrite then-value with else-value if condition doesn't hold
- Don't use when:
  - Then or else expressions have side effects
  - Then and else expressions are too expensive

Summary

- Do-While loop
- While-Do loop



General "do-while" translation

- Body:

{
  $Statement_1$;
  $Statement_2$;
  ...
  $Statement_n$;
}

- Test returns integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true

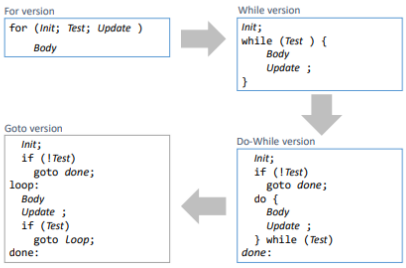
C code

```
do
  Body
  while (Test);
```

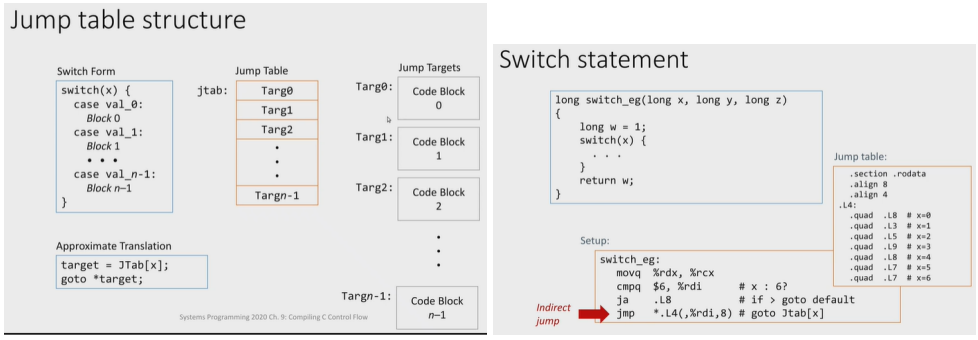
Goto version

```
Loop:
  Body
  if (Test)
    goto Loop
```

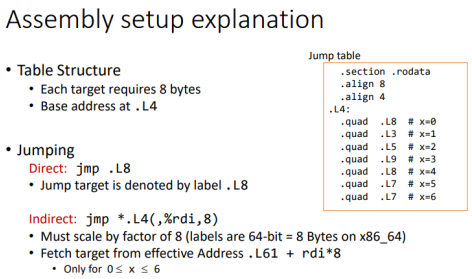
"For" → "While" → "Do-While"



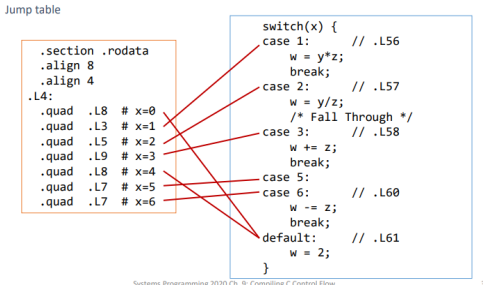
9.5 compact switch statements



When we execute the case statement, we look at the value to be tested. We get the address to jump to from the jump table. The assembly code is compact because of the indirect jump, which jumps to an address whos value depends on x (rdi).



The order of the jump table might not be in the same order as the cases because there might be values for x which are not checked and hence must be directed to the default case.

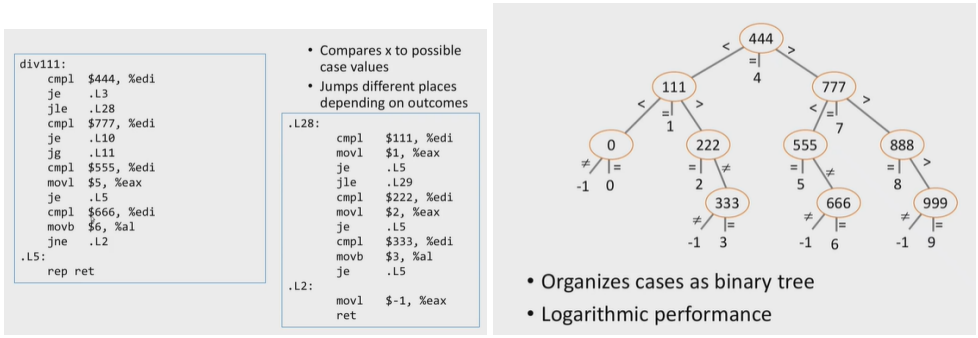


9.6 sparse switch statements

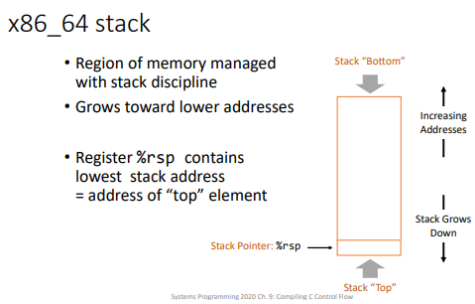
The case values have to be constants. If the values are far apart from eachother, its not practical to use a jump table e.g

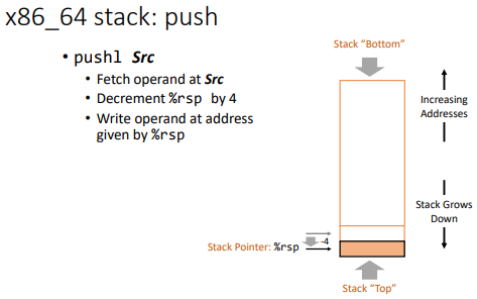
```
/* Return x/111 if x is multiple
   && <= 999.  -1 otherwise */
int div111(int x)
{
  switch(x) {
  case   0: return 0;
  case 111: return 1;
  case 222: return 2;
  case 333: return 3;
  case 444: return 4;
  case 555: return 5;
  case 666: return 6;
  case 777: return 7;
  case 888: return 8;
  case 999: return 9;
  default: return -1;
  }
}
```

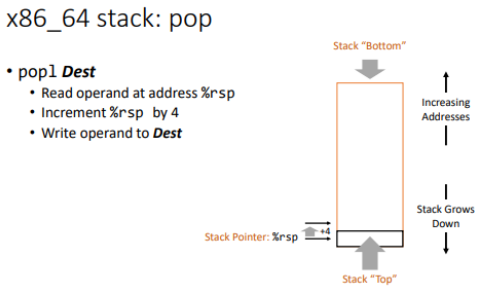
An obvious solution would be to implement this with if else statements, but this grows linear with the cases. When compiled we get the following code:

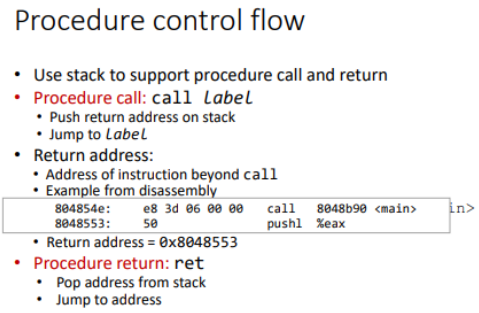


9.7 Procedure call and return



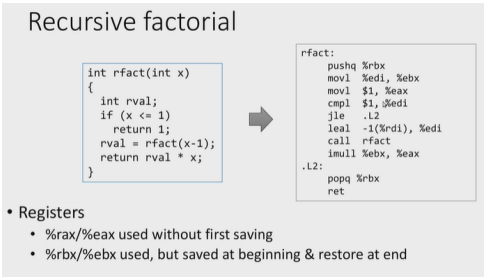






9.8 Calling conventions

How do you construct a call/return using machine code.



9.8.1 Register saving conventions

Register saving conventions

- When procedure yoo calls who:
  - yoo is the ***caller***
  - who is the ***callee***
- Can a register be used for temporary storage?

```
yoo:
    • • •
    movl $15213, %edx
    call who
    addl %edx, %eax
    • • •
    ret
```

```
who:
    • • •
    movl 8(%rsp), %edx
    addl $91125, %edx
    • • •
    ret
```

- Contents of register %edx overwritten by who

- When procedure yoo calls who:
  - yoo is the ***caller***
  - who is the ***callee***
- Can register be used for temporary storage?
- Conventions
  - ***"Caller Save"***
    - Caller saves temporary in its frame before calling
  - ***"Callee Save"***
    - Callee saves temporary in its frame before using

x86-64 integer registers

%rax	Return value, # varargs	%r8	Argument #5
%rbx	Callee saved; base ptr	%r9	Argument #6
%rcx	Argument #4	%r10	Static chain ptr
%rdx	Argument #3 (& 2nd return)	%r11	Used for linking
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved; frame ptr	%r15	Callee saved

Argument Registers are caller save registers.

- Arguments passed to functions via registers
  - If more than 6 integral parameters, then pass rest on stack
  - These registers can be used as caller-saved as well
- All references to stack frame via stack pointer
  - Eliminates 32-bit need to update %ebp/%rbp
- Other registers
  - 6+1 callee saved
  - 2 or 3 have special uses

Example:

x86-64 stack frame example

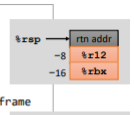
```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su(long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += a[i];
}
```

```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movslq  %esi,%rbx
    movq    %r12, -8(%rsp)
    movq    %rdi, %r12
    leaq    (%rdi,%rbx,8), %rdi
    subq    $16, %rsp
    leaq    8(%rdi), %rsi
    call    swap
    movq    (%r12,%rbx,8), %rax
    addq    %rax, sum(%rip)
    movq    (%rsp), %rbx
    movq    8(%rsp), %r12
    addq    $16, %rsp
    ret
```

- Keeps values of a and i in callee save registers
- Must set up stack frame to save these registers

```
swap_ele_su:
    movq    %rbx, -16(%rsp)     # Save %rbx
    movslq  %esi,%rbx           # Extend & save i
    movq    %r12, -8(%rsp)      # Save %r12
    movq    %rdi, %r12          # Save a
    leaq    (%rdi,%rbx,8), %rdi # &a[i]
    subq    $16, %rsp           # Allocate stack frame
    leaq    8(%rdi), %rsi       #     &a[i+1]
    call    swap                # swap()
    movq    (%r12,%rbx,8), %rax # a[i]
    addq    %rax, sum(%rip)     # sum += a[i]
    movq    (%rsp), %rbx        # Restore %rbx
    movq    8(%rsp), %r12       # Restore %r12
    addq    $16, %rsp           # Deallocate stack frame
    ret
```

```
swap_ele_su:
    movq    %rbx, -16(%rsp)     # Save %rbx
    movq    %r12, -8(%rsp)      # Save %r12
    subq    $16, %rsp           # Allocate stack frame
    movq    (%rsp), %rbx        # Restore %rbx
    movq    8(%rsp), %r12       # Restore %r12
    addq    $16, %rsp           # Deallocate stack frame
```



Interesting features of the stack frame

- Allocate entire frame at once
  - All stack accesses can be relative to %rsp
  - Do by decrementing stack pointer
  - Can delay allocation, since safe to temporarily use red zone - see next slide
- Simple deallocation
  - Increment stack pointer
  - No base/frame pointer needed

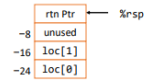
We can keep the values of a and i in registers because the swap function has to preserve callee save registers, hence we dont need to write them to memory.

9.8.2 Procedure Calls: Locals in the red zone

```
/* Swap, using local array */
void swap_a(long *xp, long *yp) {
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
  movq  (%rdi), %rax
  movq  %rax, -24(%rsp)
  movq  (%rsi), %rax
  movq  %rax, -16(%rsp)
  movq  -16(%rsp), %rax
  movq  %rax, (%rdi)
  movq  -24(%rsp), %rax
  movq  %rax, (%rsi)
  ret
```

- Avoiding stack pointer change
  - Can hold all information within small window beyond stack pointer



9.8.3 Procedure Calls: Long swap

```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    ret
```

- Operands passed in registers
  - First (xp) in `%rdi`, second (yp) in `%rsi`
  - 64-bit pointers
- No stack operations required (except `ret`)
- Avoiding stack
  - Can hold all local information in registers

9.8.4 Procedure Calls: non-leaf w/o stack frame

- No values held while swap being invoked
- No callee save registers needed

```
long scount = 0;

/* Swap a[i] & a[i+1] */
void swap_ele_se(long a[], int i)
{
    swap(&a[i], &a[i+1]);
    scount++;
}
```

```
swap_ele_se:
  movslq %esi,%rsi          # Sign extend i
  leaq   (%rdi,%rsi,8), %rdi # &a[i]
  leaq   8(%rdi), %rsi      # &a[i+1]
  call   swap               # swap()
  incq   scount(%rip)       # scount++;
  ret
```

9.8.5 Procedure Calls: call using a jump

```
long scount = 0;

/* Swap a[i] & a[i+1] */
void swap_ele(long a[], int i)
{
    swap(&a[i], &a[i+1]);
}
```

```
swap_ele:
  movslq %esi,%rsi          # Sign extend i
  leaq   (%rdi,%rsi,8), %rdi # &a[i]
  leaq   8(%rdi), %rsi      # &a[i+1]
  jmp    swap               # swap()
```

"Tail call" optimization

Chapter 10

Compiling C data Structures

10.1 One-dimensional arrays

Basic data types

- Integral
  - Stored & operated on in general (integer) registers
  - Signed vs. unsigned depends on instructions used

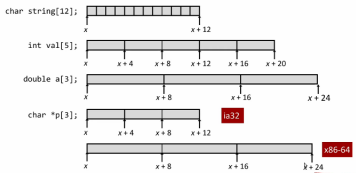
Intel	GAS	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int
quad word	q	8	[unsigned] long int (x86-64)

- Floating Point
  - Stored & operated on in floating point registers

Intel	GAS	Bytes	C
Single	s	4	float
Double	l	8	double
Extended	t	10/12/16	long double

10.1.1 Array allocation

- T A[L] : Array of
  - data type T
  - length L
- Contiguously allocated region of L * sizeof(T) bytes



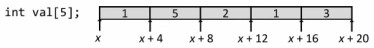
When allocating multiple arrays its not guaranteed that they will be contiguous in memory.

10.1.2 Array access

- Array of data type *T* and length *L*

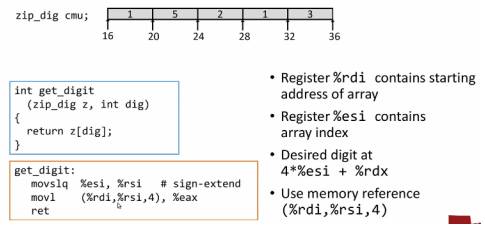
```
T A[L];
```

- Identifier A can be used as a pointer to array element 0: Type *T**



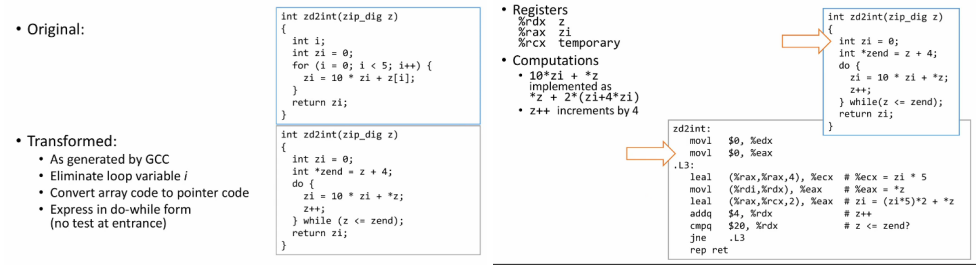
Reference	Type	Value
val[4]	int	3
val	int *	x
val + 1	int *	x + 4
&val[2]	int *	x + 8
val[5]	int	??
*(val+1)	int	5
val + i	int *	x + 4 *I*

Example:

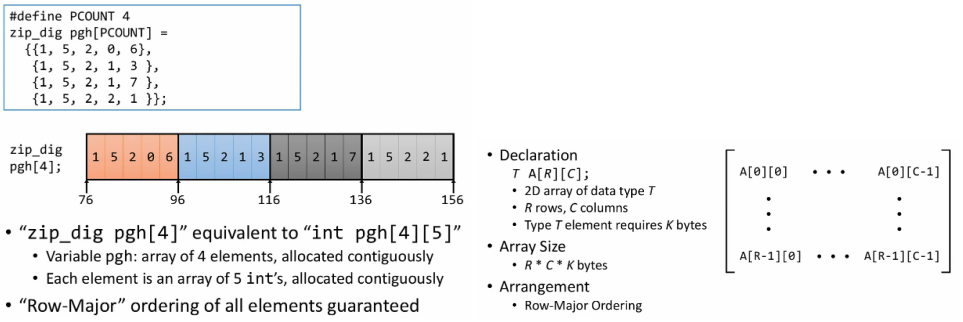


In c there is no bound checking, Out of range behaviour is implementation dependant and there is no guaranteed relative allocation of different arrays.

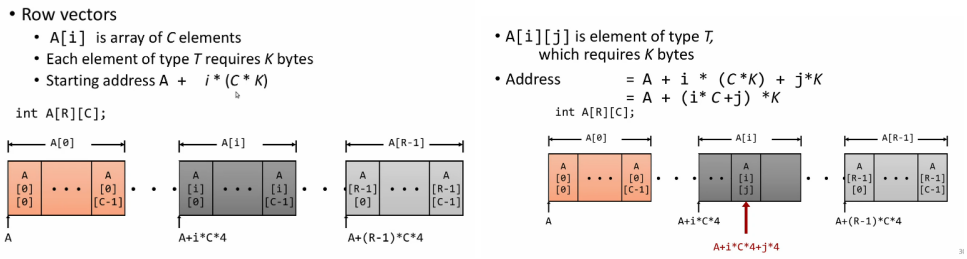
10.1.3 Array Loop example



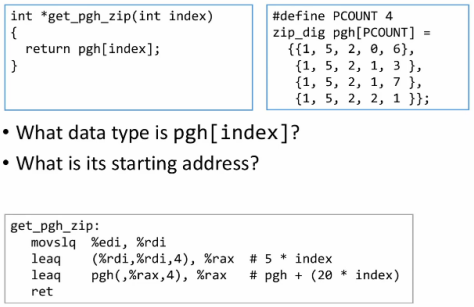
10.2 Nested Array



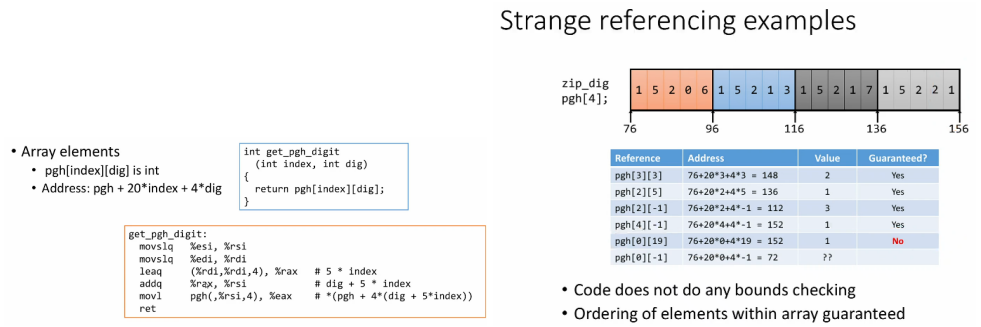
10.2.1 Nested array row access



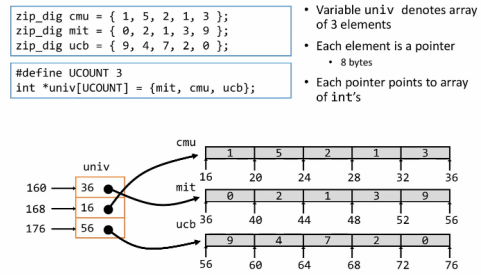
10.2.2 Nested array row access code



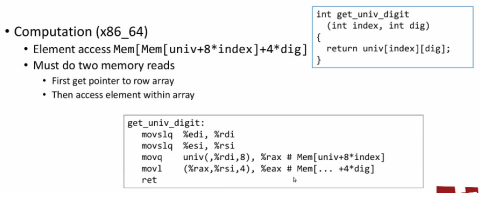
10.2.3 Nested array element access code



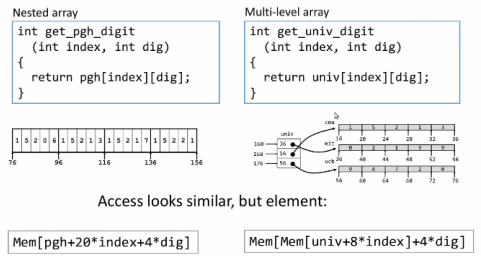
10.3 Multi-level arrays



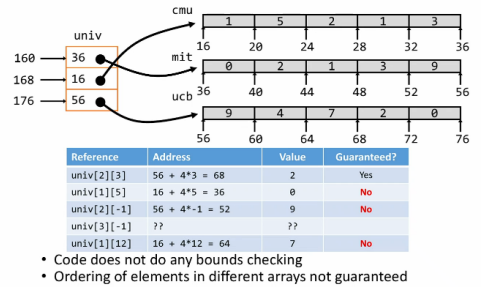
10.3.1 Element access in multi-level array



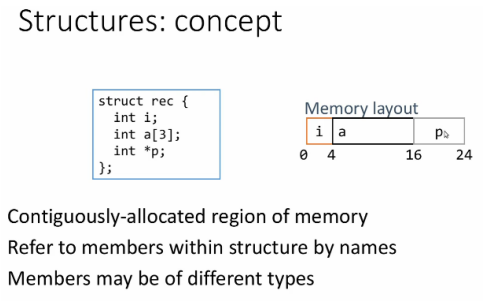
In comparison to nested arrays we need two memory accesses to access an element in a multi-level array. The first one is to access the first pointer table, the second is to access the element of the array the pointer is pointing to. For each level, we need a memory access more. For nested array we always need one memory access.



10.3.2 Referencing examples

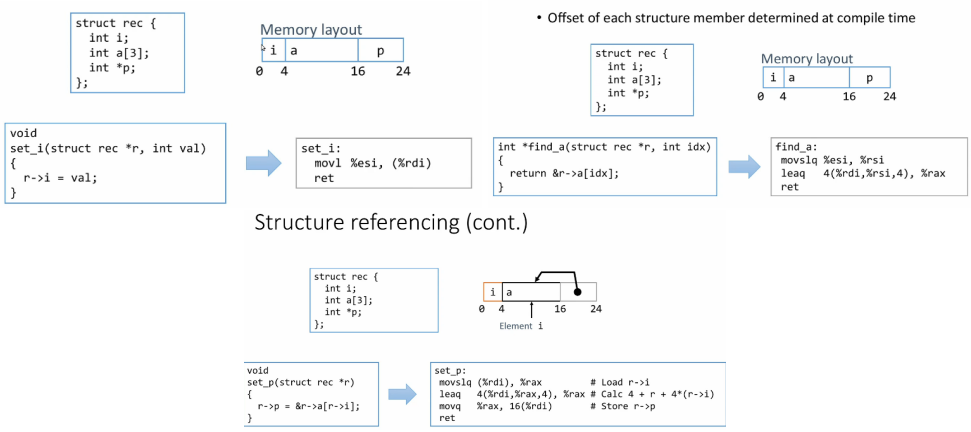


10.4 Structures



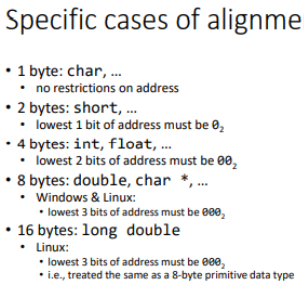
The members appear in the same order in memory as declared, but there could be padding inbetween

10.4.1 Accessing structure members

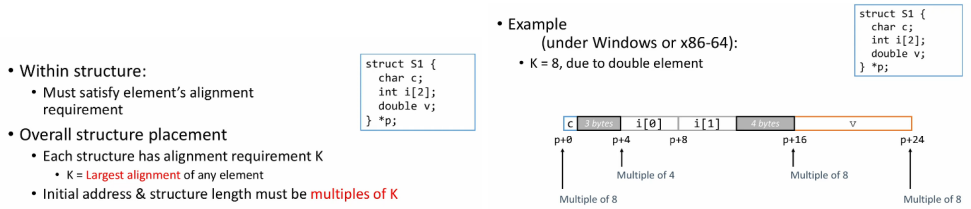


10.5 Alignment

If we have a primitive data type which requires K bytes, the address where that data is stored must be a multiple of K. The compiler inserts gaps in structures to ensure correct alignment of fields if necessary.

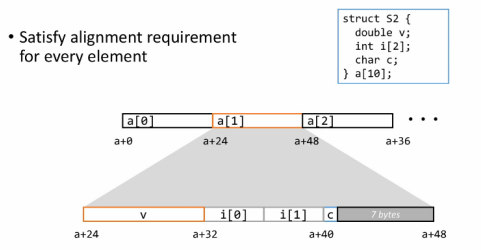


10.5.1 Alignment with structures

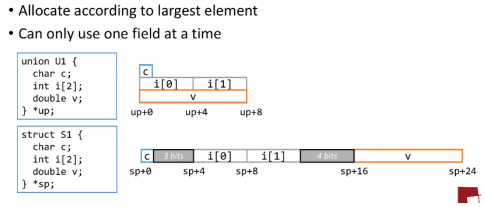


In some cases reordering the fields from largest to smallest size can save memory.

10.6 Arrays of structures

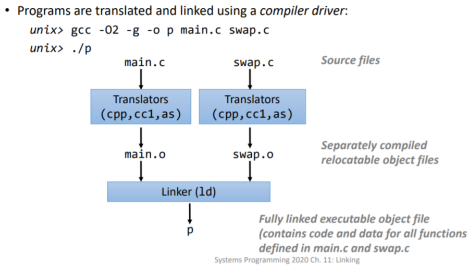


10.7 Unions

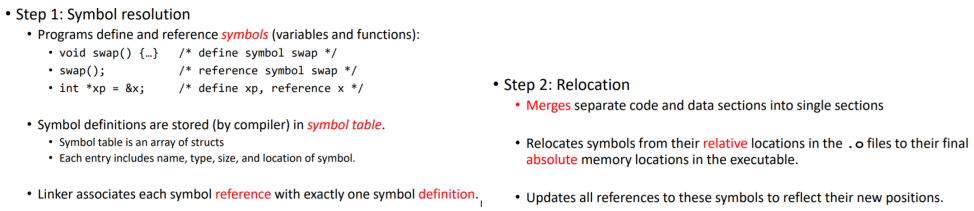


Chapter 11

Linking



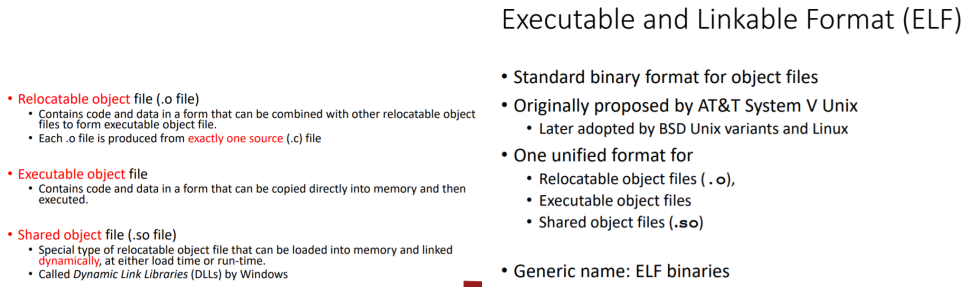
Linkers allow modularity. A program can be written as a collection of smaller source files, rather than one big one. Linkers offer more efficiency aswell, as we can compile the filles seperately. If we change a source file we dont need to recompile all the source files, but only the changed one and then it is relinked. Besides being faster it also saves memory as common functions can be aggregated into a single file. Linkers work as follows:



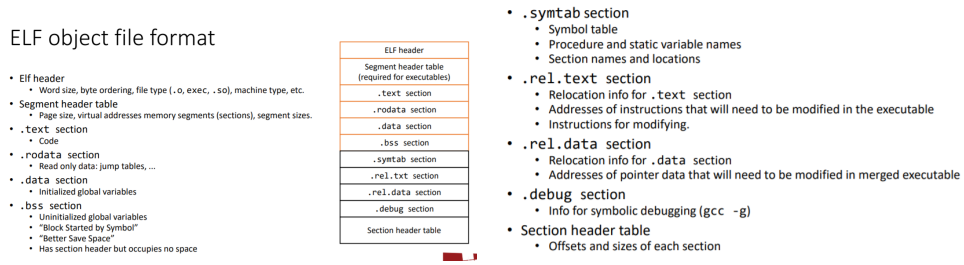
the symbol table is located in the object file.

11.1 Object files

There are 3 types object files:



Sections: refers to bits in a file **Segments:** refers to bits in memory

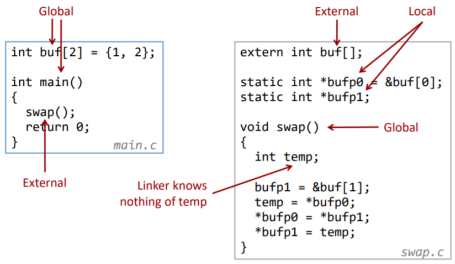


11.2 Linker symbols

Linker symbols can fall into one of the three categories:

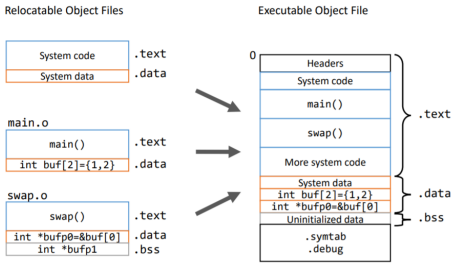
- **Global symbols:** Symbols definded by a module m that can be referenced by other modules e.g non-static C functions and non-static global variables
- **External symbols:** Global symbols that are referenced by module m but defined by some other module. (if they are not defined it results in a linker error)
- **Local symbols:** Symbols that are defined and referenced exclusively by module m e.g C functions and variables defined with the static attribute. **Local linker symbols are not local program variables**

11.2.1 Resolving symbols



declaring extern is good practice but not necessary.
"int temp" is not a local symbol, it is a local variable which sits on the stack i.e its purely a feature of the code. There will be some information in the debug section of the object file regarding the value of temp. The linker does not need to deal with temp, because the compiler has already generated all the code to deal with it.

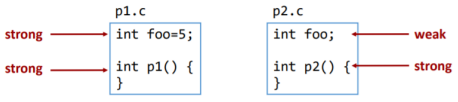
11.2.2 Relocating code and data



Each corresponding section of the individual files are merged into a single file. All of the references between the files are matched up i.e there are no more unresolved symbols. All symbols are given an address

11.2.3 Strong and weak symbols

Strong: Procedures and initialized globals **Weak:** uninitialized globals



If we compile the left one by itself we get foo in the .data file. If we compile the right one by itself we get foo in the .bss file. However if we compile both together the linker thinks both foo's are the same thing

The linker's symbol rules

1. Multiple strong symbols are not allowed
   - Each item can be defined only once
   - Otherwise: Linker error
2. Given a strong symbol and multiple weak symbol, choose the strong symbol
   - References to the weak symbol resolve to the strong symbol
3. If there are multiple weak symbols, pick an arbitrary one
   - Can override this with `gcc -fno-common`

Hence if we declare foo in the right as extern it means its a reference to a strong symbol somewhere else and the above rules dont reply. Hence if we use extern we will never use weak symbols.

Linker puzzles

`int x;` `int p1() {}`	`int p1() {}`	Link time error: two strong symbols (p1)
`int x;` `int p1() {}`	`int x;` `int p2() {}`	References to x will refer to the same uninitialized int. Is this what you really want?
`int x;` `int y;` `int p1() {}`	`double x;` `int p2() {}`	Writes to x in p2 might overwrite y! Evil!
`int x=7;` `int y=5;` `int p1() {}`	`double x;` `int p2() {}`	Writes to x in p2 will overwrite y! Nasty!
`int x=7;` `int p1() {}`	`int x;` `int p2() {}`	References to x will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Hence avoid Global variables if possible. Otherwise use static, initialize defined global variables or use "extern" when using external global variables.

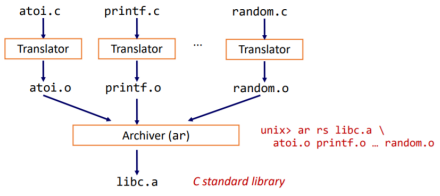
11.3 Static libraries

Packaging commonly-used functions

- How to package functions commonly used by programmers?
  - Math, I/O, memory management, string manipulation, etc.
- Awkward, given the linker framework so far:
  - **Option 1:** Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - **Option 2:** Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

Archive: A collection of documents **Library:** A collection of object files The solution to this is static libraries.

- Static libraries (`.a` archive files)
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link into executable.



Archiver allows incremental updates
Recompile function that changes and replace .o file in archive.

Commonly used libraries are:

- libc.a (the C standard library)
- libm.a (the C math library)

The standard C library is always included in the path when we search for something.
We can include a library to the path with the gcc as follows:

$$-l < name_of_library >$$

e.g to include the math library we would write -lm. In this case the math library is already in a directory the linker already knows about. To add another directory to the search path we write:

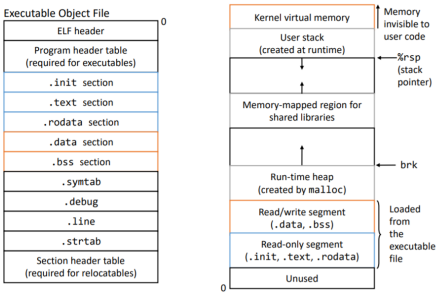
$$-L < file_path >$$

a "." for the file path means the current directory

- 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 the current unresolved references.
  - As each new `.o` or `.a` file, obj, is encountered, try to resolve each unresolved reference in the list against the symbols defined in obj.
  - If any entries in the unresolved list at end of scan, then error.
- Problem:
  - Command line order matters!
  - Moral: put libraries at the end of the command line.

```
unix> gcc -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

11.3.1 Loading executable object files



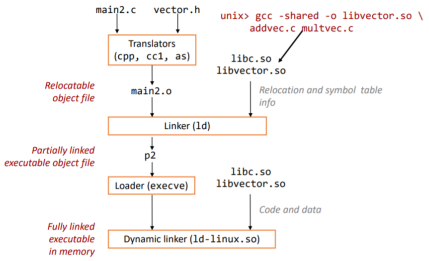
The standard C library knows where the end of the bss file is and hence knows where to start allocating memory for the heap. The loader takes the Executable Object File and distributes it in memory as seen on the right.

11.4 Shared libraries

- Static libraries have the following disadvantages:
  - Duplication in the stored executables (every function needs the standard libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink
- Solution: shared libraries
  - Object files that contain code and data that are loaded and linked into an application *dynamically,* at either *load-time* or *run-time*
  - Also called: dynamic link libraries, DLLs, .so files

- Dynamic linking can occur when executable is first loaded and run (load-time linking).
  - Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`).
  - Standard C library (`libc.so`) usually dynamically linked.
- Dynamic linking can also occur after program has begun (run-time linking).
  - In Unix, this is done by calls to the `dlopen()` interface.
    - High-performance web servers.
    - Runtime library interpositioning
- Shared library routines can be shared by multiple processes.
  - More on this when we learn about virtual memory

11.4.1 Dynamic linking at load time



11.4.2 Dynamic linking at runtime

```
#include <stdio.h>
#include <dlfcn.h>

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main(int argc, char *argv[])
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* dynamically load the shared lib that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    ...
```

```
    ...

    /* get a pointer to the addvec() function we just loaded */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    /* Now we can call addvec() it just like any other function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

Chapter 12

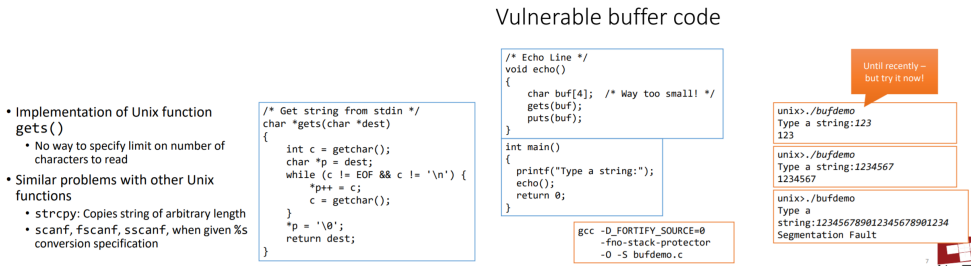
Code Vulnerabilities

12.1 Worms and Viruses

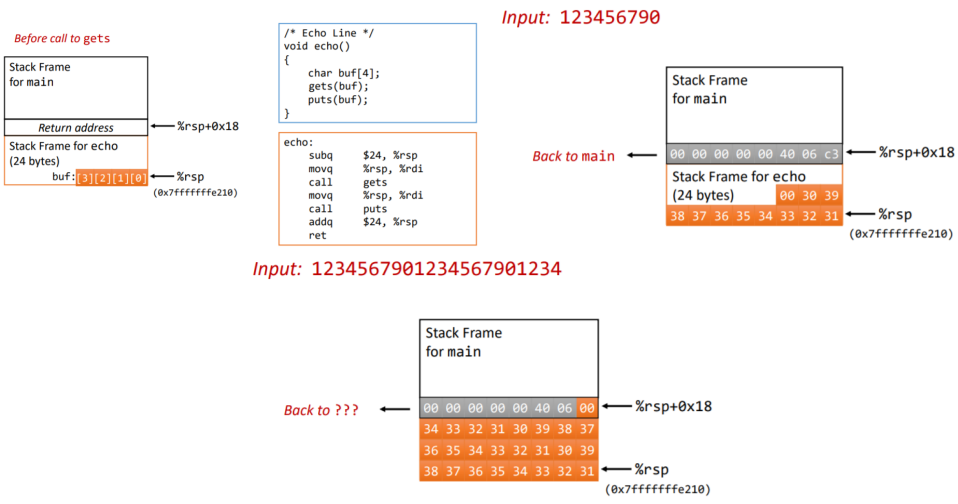
Worm: A program that can run by itself, it can propagate a full working version of itself to other computers

Virus: Code that can add itself to other programs, it cannot run independently

12.2 Stack Overflow bug

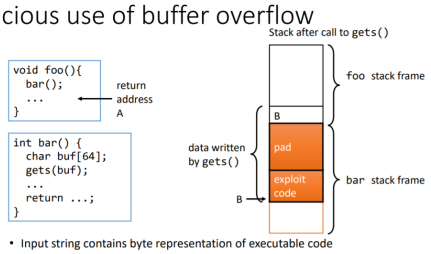


As soon as the string gets large enough, we get a segmentation fault. The program doesn't have anything obvious that crashes itself, it doesnt corrupt/manipulate its own memory. What happens is that the input can be used to subvert a poorly written program e.g gets in order to corrupt memory and cause a segmentation fault. These days modern compilers have protection against this particular attack.



As soon as we fill up the stack frame and add more input, the return address of the buffer becomes random. So rather than causing the program to crash we can make it jump and execute some other code. If we know the address that the stack pointer points to, we can cause the return address to jump into the code we have just added to the stack frame. It was possible to find out the stack pointer before address randomization, because you could test it out on your own machine first, and hence all machines of the same type would have the same stack pointer.

Malicious use of buffer overflow



- Input string contains byte representation of executable code
- Overwrite return address with address of buffer
- When `bar()` executes `ret`, will jump to exploit code

Exploits based on buffer overflows

- *Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines*
- Internet worm (one vector)
  - Early versions of the finger server (`fingerd`) used `gets()` to read the argument sent by the client:
    - `finger droh@cs.cmu.edu`
  - Worm attacked `fingerd` server by sending bad argument:
    - `finger "exploit-code padding new-return-address"`
    - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

12.3 Stopping overrun bugs

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- Use library routines that limit string lengths
  - `fgets` instead of `gets`
  - `strncpy` instead of `strcpy`
  - Don't use `scanf` with `%s` conversion specification
    - Use `fgets` to read the string
    - Or use `%ns` - where n is a suitable integer

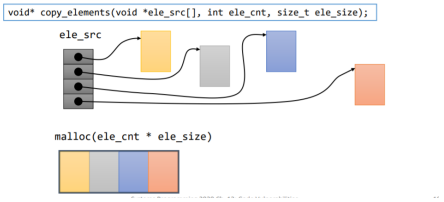
System-level protections

- Compiler-inserted checks on functions
  - Compiler now understands library calls...
- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Makes it difficult to predict beginning of inserted code
- Nonexecutable code segments
  - In older x86, can mark region of memory as either "read-only" or "writeable"
    - Can execute anything readable
  - Add explicit "execute" permission to hardware

12.4 XDR vulnerability

SUN XDR library

- Widely used for transferring data between machines (e.g. for NFS)



XDR code

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /*
     * Allocate buffer for ele_cnt objects, each of ele_size bytes
     * and copy from locations designated by ele_src
     */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL)
        /* malloc failed */
        return NULL;
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
```

XDR vulnerability

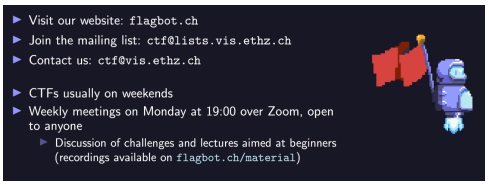
`malloc(ele_cnt * ele_size)`

- What if (on a 32-bit machine):
  - `ele_cnt` = $2^{20} + 1$
  - `ele_size` = $4096 = 2^{12}$
  - Allocation = ??

- How can I make this function secure?

Allocating a large amount of memory would most likely fail with malloc, but since this is a large unsigned value (signed * unsigned = unsigned) it wraps around in this case $2^20 + 1 = 1$ which on a 32-bit machine would allocate 32-bits. So malloc would return true and we would end up overwriting an enormous amount of data.

12.5 CTF



Chapter 13

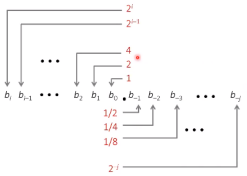
Floating Point

13.1 Representing Floating Point

13.1.1 Fractional binary numbers

- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number:

$$\sum_{k=-j}^{i} b_k \cdot 2^k$$



- Observations
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Value	Representation
$5^3/_4$	101.11_2
$2^7/_8$	10.111_2
$^{63}/_{64}$	0.111111_2

This notation has its limitation, we can only exactly represent numbers of the form $\frac{x}{2^k}$. Rational numbers have repeating bit patterns.

Value	Representation
1/3	$0.0101010101[01]\ldots_2$
1/5	$0.001100110011[0011]\ldots_2$
1/10	$0.0001100110011[0011]\ldots_2$

You can't represent 0.1_{10}

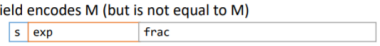
13.1.2 Floating point representation

Floating point representation
(recap from Digital Circuits)

- Numerical form:

$$(-1)^s . M . 2^E$$

  - Sign bit s determines whether number is negative or positive
  - Significand M normally a fractional value in range [1.0,2.0).
  - Exponent E weights value by power of two
- Encoding
  - MSB (Most Significant Bit) s is sign bit s
  - exp field encodes E (but is not equal to E)
  - frac field encodes M (but is not equal to M)



13.2 Types of IEEE floating point numbers

13.2.1 Precisions

IEEE 754 standards (and others...)

Precision	Significand bits	Exponent bits	Total
Half	11	5	16
Single	24	8	32
Double	53	11	64
Quadruple	113	15	128
Octuple	237	19	256
Google bfloat16	*7*	*8*	*16*
Nvidia TensorFloat	*10*	*8*	*19*
AMD fp24	*17*	*7*	*24*

13.2.2 Floating point in C

- C99 guarantees two levels:
  - `float` - single precision
  - `double` - double precision
- `long double` can be double, extended, or quadruple precision

- Conversions/casting
  - Casting between `int`, `float`, and `double` changes bit representation
  - `double`/`float` → `int`
    - Truncates fractional part (like rounding toward zero)
    - Not defined when out of range or NaN: Generally sets to TMin
  - `int` → `double`
    - Exact conversion, as long as `int` has ≤ 53 bit word size
  - `int` → `float`
    - Will round according to rounding mode

13.2.3 Normalized Values

- Condition: exp ≠ 000…0 and exp ≠ 111…1

- Exponent coded as biased value: $E = Exp - Bias$
  - *Exp*: unsigned value exp
  - *Bias* = $2^{e-1} - 1$, where e is number of exponent bits
    - Single precision: 127 (*Exp*: 1…254, *E*: -126…127)
    - Double precision: 1023 (*Exp*: 1…2046, *E*: -1022…1023)

- Significand coded with implied leading 1: $M = 1.xxx\ldots x_2$
  - xxx…x: bits of `frac`
  - Minimum when 000…0 ($M = 1.0$)
  - Maximum when 111…1 ($M = 2.0 - \varepsilon$)
  - Get extra leading bit for "free"

- Value: `float F = 15213.0;`
  $15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13}$

- Significand
  - $M =$ 1.1101101101101_2
  - `frac` = $1101101101101\underline{0000000000}_2$

- Exponent
  - $E =$ 13
  - *Bias* = 127
  - *Exp* = $140 = 10001100_2$

- Result:

0	10001100	11011011011010000000000
s	exp	frac

13.2.4 Denormalized values

- Condition: exp = 000…0

- Exponent value: $E = -Bias + 1$ (instead of E = 0 – Bias)
- Significand coded with implied leading 0: $M = 0.xxx\ldots x_2$
  - xxx…x: bits of `frac`
- Cases
  - exp = 000…0, frac = 000…0
    - Represents value 0
    - Note distinct values: +0 and –0 (why?)
  - exp = 000…0, frac ≠ 000…0
    - Numbers very close to 0.0
    - Lose precision as get smaller
    - Equispaced

There are 2 distinct values for 0 because of the sign bit

{single,double}

Description	exp	frac	Numerical value
Zero	00…00	00…00	0.0
Smallest pos. denorm. • Single ≈ 1.4×10^{-45} • Double ≈ 4.9×10^{-324}	00…00	00…01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
Largest denormalized • Single ≈ 1.18×10^{-38} • Double ≈ 2.2×10^{-308}	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
Smallest pos. normalized • Just larger than largest denormalized	00…01	00…00	$2^{-\{126,1022\}}$
One	01…11	00…00	1.0
Largest normalized • Single ≈ 3.4×10^{38} • Double ≈ 1.8×10^{308}	11…10	11…11	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$

13.2.5 Special values

Special values

- Condition: exp = 111…1
- Case: exp = 111…1, frac = 000…0
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g. 1.0/0.0 = -1.0/-0.0 = +∞, 1.0/-0.0 = -∞
- Case: exp = 111…1, frac ≠ 000…0
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ - ∞, ∞ * 0

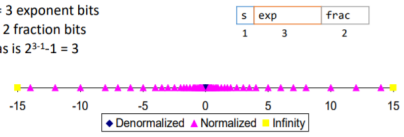
13.2.6 Properties of encoding

- FP zero same as integer zero
  - All bits = 0 (for +0)
- Can (almost) use unsigned integer comparison
  - Must first compare sign bits
  - Must consider -0 = 0
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

13.3 Floating point ranges

Distribution of values

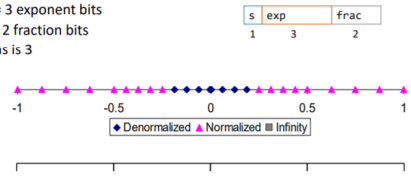
- *Really* tiny 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is $2^{3-1}-1 = 3$



- Notice how the distribution gets denser toward zero.

Distribution of values (close-up view)

- *Really* tiny 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3



Denormalized numbers have equal spacing around zero because they have the same E value.

13.4 Floating-point rounding and arithmetic

13.4.1 Rounding

- Rounding modes (illustrate with CHF rounding)

	CHF 1.40	CHF 1.60	CHF 1.50	CHF 2.50	CHF -1.50
Towards zero	1	1	1	2	-1
Round down (-∞)	1	1	1	2	-2
Round up (+∞)	2	2	2	3	-1
Nearest Even (default)	1	2	2	2	-2

Rounding to nearest even avoids biased rounding. All other rounding types are statistically biased i.e a sum of set of positive numbers will consistently be over- or under- estimated.

- Applying to other decimal places / bit positions
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth

Value	Result	Description
1.2349999	1.23	(less than half way)
1.2350001	1.24	(greater than half way)
1.2350000	1.24	(half-way – round up)
1.2450000	1.24	(half way – round down)

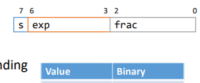
13.4.2 Rounding binary numbers

- Binary fractional numbers
  - "Even" when least significant bit is 0
  - "Half way" when bits to right of rounding position = $100\ldots_2$
- Examples
  Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Result
$2\,3/32$	10.00011_2	10.00_2	< ½ : down	2
$2\,3/16$	10.00110_2	10.01_2	> ½ : up	$2\,1/4$
$2\,7/8$	10.11100_2	11.00_2	= ½ : up	3
$2\,5/8$	10.10100_2	10.10_2	= ½ : down	$2\,½$

13.4.3 Creating a floating point number

- Steps
  - Normalize to have leading 1
  - Round to fit within fraction
  - Postnormalize to deal with effects of rounding



Value	Binary
128	10000000
15	00001101
17	00010001
19	00010011
138	10001010
63	00111111

- Case study
  - Convert 8-bit unsigned numbers to tiny floating point format

Normalize

Requirement
- Set binary point so that numbers of form 1.xxxxx
- Adjust all to have leading one
  - Decrement exponent as shift left

Value	Binary	Fraction	Exponent
128	10000000	1.0000000	7
15	00001101	1.1010000	3
17	00010001	1.0001000	4
19	00010011	1.0011000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5

Rounding

1.BBGRXXX

Guard bit: LSB of result
Round bit: 1st bit removed
Sticky bit: OR of remaining bits

- Round up conditions
  - Round = 1, Sticky = 1 ⇒ > 0.5
  - Guard = 1, Round = 1, Sticky = 0 ⇒ Round to even

Value	Fraction	GRS	Incr?	Rounded
128	1.0000000	000	N	1.000
15	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	111	Y	10.000

Postnormalize

- Issue
  - Rounding may have caused overflow
  - Handle by shifting right once & incrementing exponent

Value	Rounded	Exp	Adjusted	Result
128	1.000	7		128
15	1.101	3		15
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	1.000/6	64

13.5 Floating point addition and multiplication

13.5.1 Floating point multiplication

- Exact Result: $(-1)^s\ M\ 2^E$
  - Sign s: $s_1 \wedge s_2$
  - Significand M: $M_1 * M_2$
  - Exponent E: $E_1 + E_2$

$(-1)^{s1}\ M_1\ 2^{E1}$ x $(-1)^{s2}\ M_2\ 2^{E2}$

- Fixing
  - If $M \geq 2$, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit `frac` precision
- Implementation
  - Biggest chore is multiplying significands

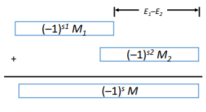
Mathematical properties of floating point multiplication

- Compare to commutative ring
  - Closed under multiplication? *Yes*
    - But may generate infinity or NaN
  - Multiplication is commutative? *Yes*
  - Multiplication is associative? *No*
    - Possibility of overflow, inexactness of rounding
  - 1 is multiplicative identity? *Yes*
  - Multiplication distributes over addition? *No*
    - Possibility of overflow, inexactness of rounding
- Monotonicity
  - $a \geq b$ & $c \geq 0 \Rightarrow a * c \geq b * c$? *Almost*
    - Except for infinities & NaNs

13.5.2 Floating point addition

$(-1)^{s1}\ M_1\ 2^{E1}$ + $(-1)^{s2}\ M_2\ 2^{E2}$
Assume $E_1 > E_2$

- Exact result: $(-1)^s\ M\ 2^E$
  - Sign s, significand M:
    - Result of signed align & add
  - Exponent E: E_1



- Fixing
  - If $M \geq 2$, shift M right, increment E
  - if $M < 1$, shift M left k positions, decrement E by k
  - Overflow if E out of range
  - Round M to fit **`frac`** precision

Mathematical properties of floating point addition

- Compare to those of Abelian Group
  - Closed under addition? *Yes*
    - But may generate infinity or NaN
  - Commutative? *Yes*
  - Associative? *No*
    - Overflow and inexactness of rounding
  - 0 is additive identity? *Yes*
  - Every element has additive inverse? *Almost*
    - Except for infinities & NaNs
- Monotonicity
  - $a \geq b \Rightarrow a+c \geq b+c$? *Almost*
    - Except for infinities & NaNs

13.6 SSE floating point

Vector instructions: SSE family

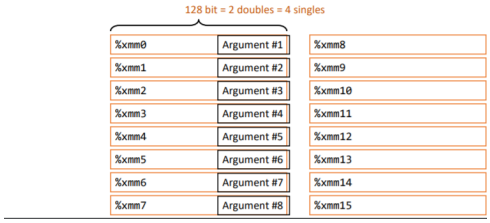
- SIMD (single-instruction, multiple data) vector instructions
  - New data types, registers, operations
  - Parallel operation on small (length 2-8) vectors of integers or floats
  - Example:

- Floating point vector instructions
  - Available with Intel's SSE (streaming SIMD extensions) family
  - SSE starting with Pentium III: 4-way single precision
  - SSE2 starting with Pentium 4: 2-way double precision
  - **All x86-64 have SSE3 (superset of SSE2, SSE)**

Here we focus on SSE3:

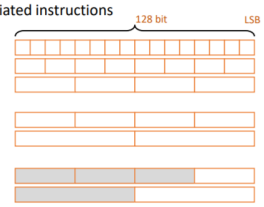
SSE3 registers

- All caller saved
- %xmm0 for floating point return value



SSE3 registers

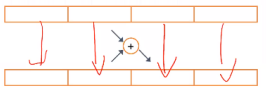
- Different data types and associated instructions
- Integer vectors:
  - 16-way byte
  - 8-way 2 bytes
  - 4-way 4 bytes
- Floating point vectors:
  - 4-way single
  - 2-way double
- Floating point scalars:
  - single
  - double



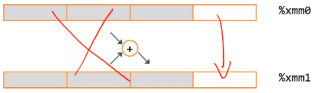
Example:

SSE3 instructions: examples

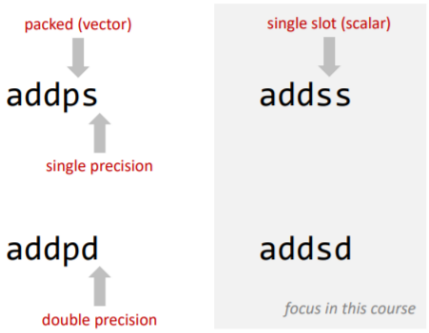
- Single precision 4-way vector add: `addps %xmm0 %xmm1`



- Single precision scalar add: `addss %xmm0 %xmm1`



13.6.1 SSE3 instruction names



SSE3 basic instructions

- Moves:

Single	Double	Effect
movss	movsd	D ← S

  - Usual operand form: reg → reg, reg → mem, mem → reg

- Arithmetic:

Single	Double	Effect
addss	addsd	D ← D + S
subss	subsd	D ← D – S
mulss	mulsd	D ← D x S
divss	divsd	D ← D / S
maxss	maxsd	D ← max(D,S)
minss	minsd	D ← min(D,S)
sqrtss	sqrtsd	D ← sqrt(S)

SSE3 conversion instructions

- Conversions
  - Same operand forms as moves

Instruction	Description
cvtss2sd	single → double
cvtsd2ss	double → single
cvtsi2ss	int → single
cvtsi2sd	int → double
cvtsi2ssq	quad int → single
cvtsi2sdq	quad int → double
cvttss2si	single → int (truncation)
cvttsd2si	double → int (truncation)
cvttss2siq	single → quad int (truncation)
cvttsd2siq	double → quad int (truncation)

13.6.2 SSE3 Examples

x86-64 FP code example

```
double funct(double a, float x, double b, int i)
{
  return a*x - b/i;
}
```

```
a %xmm0 double
x %xmm1 float
b %xmm2 double
i %edi int

funct:
  cvtss2sd %xmm1, %xmm1      # %xmm1 = (double) x
  mulsd %xmm0, %xmm1         # %xmm1 = a*x
  cvtsi2sd %edi, %xmm0       # %xmm0 = (double) i
  divsd %xmm0, %xmm2         # %xmm2 = b/i
  movsd %xmm1, %xmm0         # %xmm0 = a*x
  subsd %xmm2, %xmm0         # return a*x - b/i
ret
```

x86-64 FP code example

- Compute inner product of two vectors
  - Single precision arithmetic
  - Uses SSE3 instructions

```
float ipf (float x[], float y[], int n)
{
    int i;
    float result = 0.0;

    for (i = 0; i < n; i++) {
        result += x[i]*y[i];
    }
    return result;
}
```

```
ipf:
    xorps   %xmm1, %xmm1        # result = 0.0
    xorl    %ecx, %ecx          # i = 0
    jmp     .L8                 # goto middle
.L10:                           # loop:
    movslq  %ecx,%rax           # icpy = i
    incl    %ecx                # i++
    movss (%rsi,%rax,4), %xmm0  # t =  y[icpy]
    mulss (%rdi,%rax,4), %xmm0  # t *= x[icpy]
    addss   %xmm0, %xmm1        # result += t
.L8:                            # middle:
    cmpl    %edx, %ecx          # i:n
    jl      .L10                # if < goto loop
    movaps  %xmm1, %xmm0        # return result
    ret
```

13.6.3 Constants

```
double cel2fahr(double temp)
{
  return 1.8 * temp + 32.0;
}
```

- Here: Constants in decimal format
  - compiler decision
  - hex more readable

```
# Constant declarations
.LC2:
  .long 3435973837     # Low order four bytes of 1.8
  .long 1073532108     # High order four bytes of 1.8
.LC4:
  .long 0              # Low order four bytes of 32.0
  .long 1077936128     # High order four bytes of 32.0
```

```
# Code
cel2fahr:
  mulsd .LC2(%rip), %xmm0  # Multiply by 1.8
  addsd .LC4(%rip), %xmm0  # Add 32.0
  ret
```

- Previous slide: Claim

```
.LC4:
  .long 0              # Low order four bytes of 32.0
  .long 1077936128     # High order four bytes of 32.0
```

- Convert to hex format:

```
.LC4:
  .long 0x0            # Low order four bytes of 32.0
  .long 0x40400000     # High order four bytes of 32.0
```

- Convert to double:
  - Remember: e = 11 exponent bits, bias = 2^{e-1}-1 = 1023

The top bit is the sign bit, the next 11 bits are the exp field, hence the first 12 bits are the exponent which correspond to 3 hex digits. i.e 0x404 =1028 is the exponent, since the bias is 1023 we have 1028-1023 = 5, so the actual exponent value is 5. The rest is the significand which is all zeroes. There is an implied leading one. Hence the value is $1 \cdot 2^5$

13.6.4 Vector Instructions

- Recently, `gcc` can autovectorize to some extent
  - e.g. `-ftree-vectorize`
  - No speed-up guaranteed
  - Very limited
  - `icc` is usually much better
- For highest performance vectorize yourself using ***intrinsics***
  - Intrinsics = C interface to vector instructions
- Most recently:
  - Intel AVX512: 8-way double, 16-way single

Chapter 14

Optimizing Compilers

14.1 Optimizing Compilers

- Use optimization flags, default can be no optimization (-O0)!
- Good choices for gcc: -O2, -O3, -march=xxx, -m64
- Try different flags and maybe different compilers
  - icc is often faster than gcc

Default is O0 because it offers faster compilation. O1,O2,O3 are more optimization respectively but higher number does not guarantee that there is better performance. The compiler only does optimization which preserves the functionality of the code Example:

Example

- Compiled with no optimization: ~820 cycles
- -O1: 200 cycles
- -O2: 132 cycles
- -O3: 32 cycles

```
double a[4][4];
double b[4][4];
double c[4][4]; // set to zero

/* Multiply 4 x 4 matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            for (k = 0; k < 4; k++)
                c[i*4+j] += a[i*4 + k]*b[k*4 + j];
}
```

- -O3 –m64 -march=haswell –fno-tree-vectorize: **8 cycles** (yes, 8!)

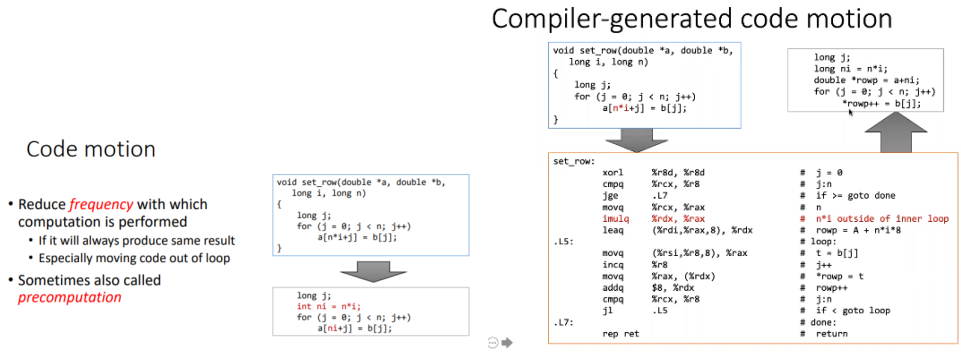
Strengths and Weaknesses

- Compilers are good at: mapping program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- Compilers are not good at: improving asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter
- Compilers are not good at: overcoming "optimization blockers"
  - potential memory aliasing
  - potential procedure side-effects

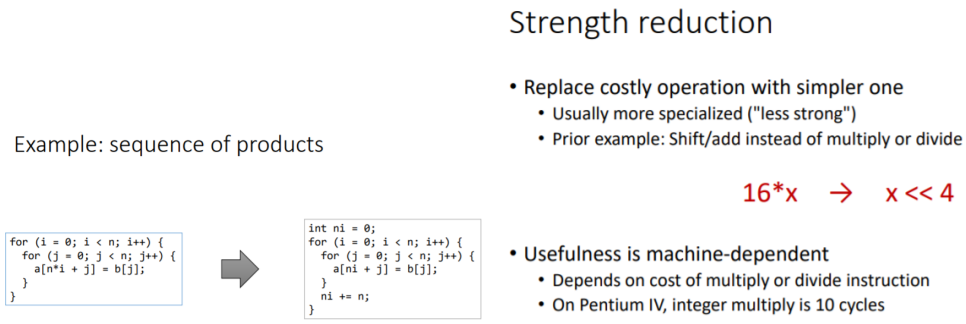
Limitations of optimizing compilers

- *If in doubt, the compiler is conservative*
- Operate under fundamental constraints
  - Must not change program behavior under any possible condition
  - Often prevents it from making optimizations when would only affect behavior under pathological conditions.
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - e.g., data ranges may be more limited than variable types suggest
- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
- Most analysis is based only on static information
  - Compiler has difficulty anticipating run-time inputs

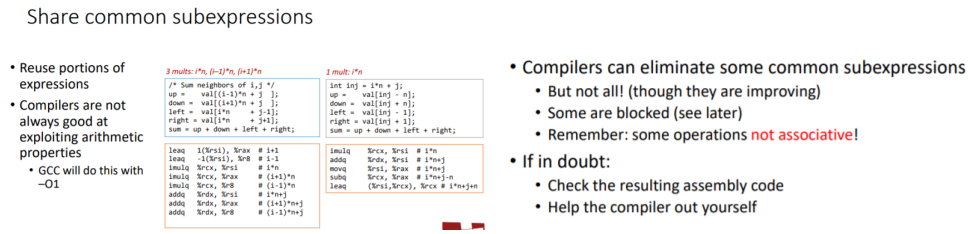
14.2 Code motion and precomputation



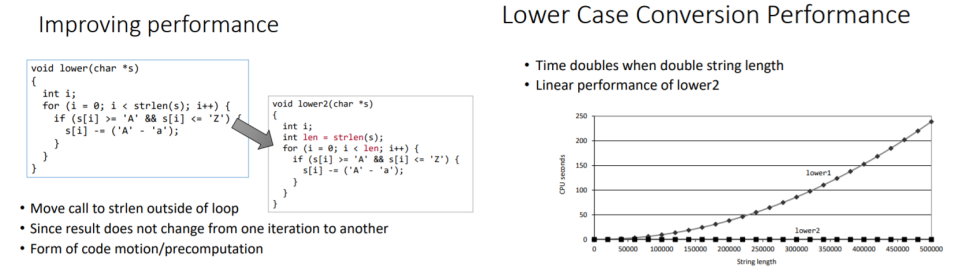
14.3 Strength reduction



14.4 Common subexpressions



14.5 Optimization blocker: Procedure calls



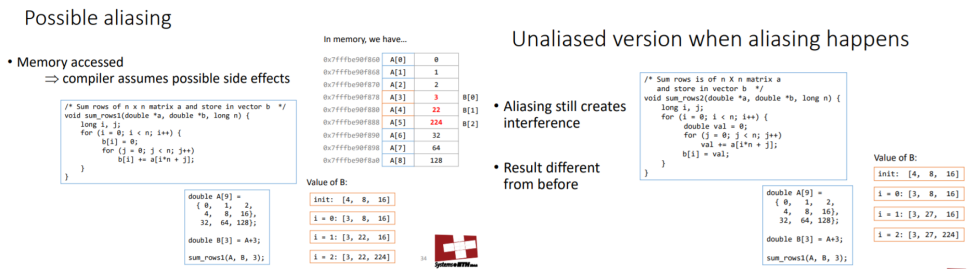
the compiler treats procedures as black boxes and does not replace it with its result, even if its called multiple times in a loop because it might not return the same result given the same inputs, as it could be dependant on global states. Hence in the above example we have to optimize the code because the compiler wont do it.

14.6 Optimization blocker: memory aliasing

When two different memory references specify a single location

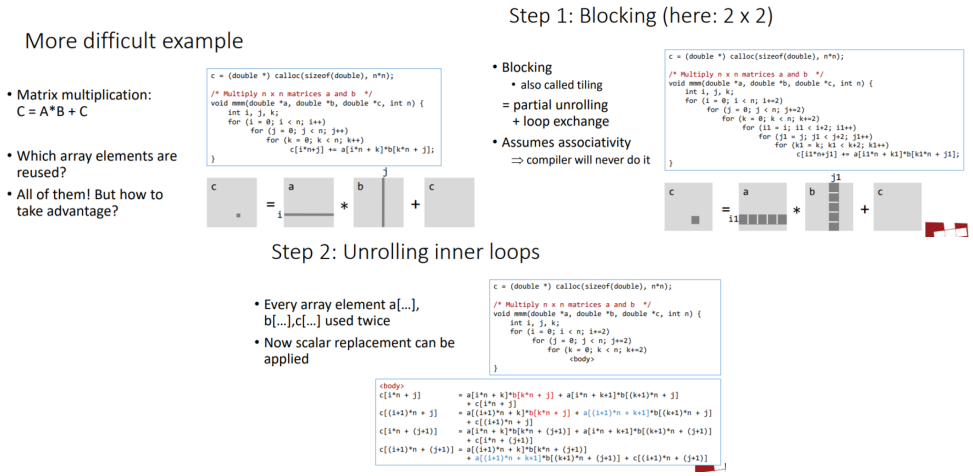
- Easy to have happen in C
  - Since allowed to do address arithmetic
  - Direct access to storage structures
- Get in habit of introducing local variables
  - Accumulating within loops
  - Your way of telling compiler not to check for aliasing

Example:



What happens in the possible aliasing scenerio is that B[0] is overwritten by the first three elements of A becoming 3, then as we sum up the second row of A to calculate B[1], we set B[1] = A[4] to 0 and add A[3] =3 + A[4] (which is now 3) + A[5]=16 hence we get 22. The compiler doesnt optimize the code because it assumes this is the behaviour we want. We can remove aliasing by using temporary variables in this case copying the array elements into "val". The unaliased version still creates some interference because we are still overwriting some memory.

14.7 Blocking and unrolling



Blocking is more cache line friendly because we are using spatial locality more efficiently. The size of the block depends on the architecture i.e how much memory is brought to the cache. The second improvement is unrolling of the inner loops, which allows us to reuse some of the computed values. In general this is the responsability of the programmer, the compiler will not do this optimization

Chapter 15

Architecture and Optimization

In order to measure performance we use benchmarks. The benchmark should be as general as possible testing different datatypes e.g:

Benchmark example: data type for vectors

- Data types: different declarations for `data_t`:
  - `int`
  - `long`
  - `float`
  - `double`


```
/* data structure for vectors */
struct vec {
    size_t len;
    data_t *data;
};
```

```
/* retrieve vector element
   and store at val */
int get_vec_element
  (struct vec* v, size_t idx, data_t *val)
{
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

Benchmark computation

- Data types (`data_t`):
  - `int`
  - `long`
  - `float`
  - `double`
- Operations: definitions of OP and IDENT
  - `+` / `0`
  - `*` / `1`

```
void combine1(struct vec *v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

A way to express performance of a program that operates on vectors or lists is Cycles per Element (CPE). The **Execution time** is given by:

$$CPE \cdot n + \text{ overhead}$$

We can increase performance by using the optimization flags, or other basic code optimizations:

Benchmark Performance

```
void combine1(struct vec *v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14

Basic Optimizations

- Move `vec_length` out of loop
- Avoid bounds check on each cycle, instead access array directly (no procedure call in loop)
- Accumulate in temporary

```
void combine4(struct vec *v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

Effect of Basic Optimizations

- Eliminates sources of overhead in loop

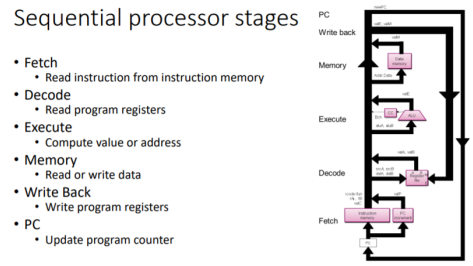
```
void combine4(struct vec *v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

But can we do better?

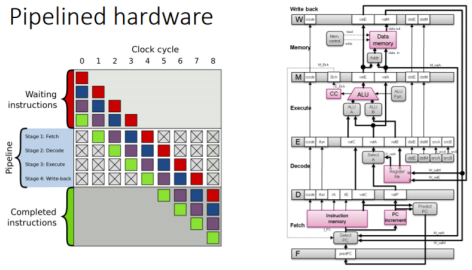
15.1 modern processor design

15.1.1 Sequential processor stages



In practice this is too slow. Not all instructions need each step. And there will be steps which are idle. The solution is using pipelined hardware.

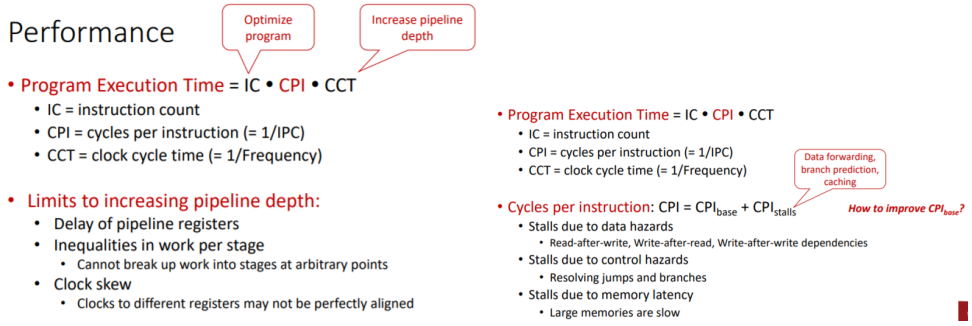
15.1.2 Pipelined hardware



The following limitations for pipelining exist:

- **Data Hazards:** Hazards caused by data dependencies i.e we need a result from another calculation. These can be solved by implementing data forwarding or stalling the pipeline
- **Control Hazards:** Hazards caused by branches i.e we dont know for sure wether or not to take a branch. This can be solved by rollback or stalling the fetch stage until the branch is known.

15.1.3 Performance

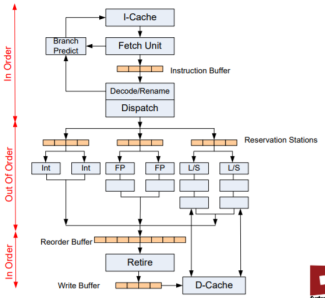


We can optimize IC by optimizing our code. We can decrease CCT, by pipelining. When there are more stages, the signal doesnt need to traverse as far and hence we can increase the clock frequency. What we can do to improve the CPI is widen the Processor i.e use a superscalar processor

15.1.4 Superscalar processor

Superscalar CPU design

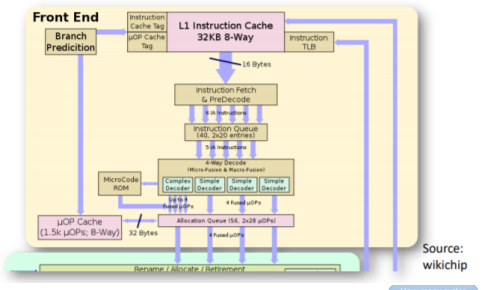
- Definition: A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.
- Benefit: without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have
- Most CPUs since about 1998 are superscalar.
- Intel: since Pentium Pro



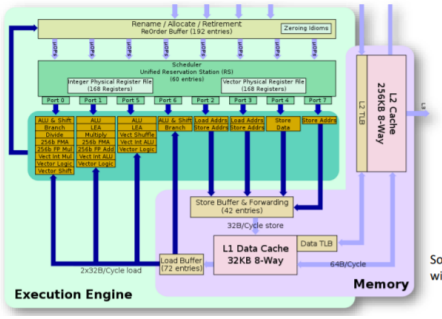
We need to fetch instructions in the order they are being executed in. But when we are actually executing them then we can execute any instruction which has no dependencies. We do need to preserve the semantics of the program hence we need to commit the instructions in the same order as they have been fetched, this is done with the reorder buffer.

15.2 Superscalar processor performance

15.2.1 Intel Haswell CPU schematics



Source: wikichip



Source: wikichip

Example: Intel Haswell CPU

How many cycles need to wait until can issue next op in this functional unit

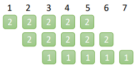
- 8 total functional units
- Multiple instructions can execute in parallel
  - 2 load, with address computation
  - 1 store, with address computation
  - 4 integer
  - 2 FP multiply
  - 1 FP add
  - 1 FP divide
- Some instructions take > 1 cycle, but can be pipelined

Instruction	Latency	Cycles/issue
Load/store	4	1
Int Add	1	1
Int Multiply	3	1
Int/Long Divide	3-30	3-30
Sngl/Dbl FP Multiply	5	1
Sngl/Dbl FP Add	3	1
Sngl/Dbl FP Divide	3-15	3-15

Cycles/issue means how many cycles we need to wait before we can issue a new instruction on that FU

15.2.2 Latency vs Throughput

- Last slide: *latency* *cycles/issue*
  FP Multiply: 5 1



- How fast can 5 independent FP multiplies be executed?
  - `t1 = t2*t3; t4 = t5*t6; ...`
- And 5 sequentially dependent FP multiplies?
  - `t1 = t2*t3; t4 = t5*t1; t6 = t7*t4; ...`
- Major problem for fast execution: **Keep pipelines filled**

If we have independent multiplications we can finish the 5 multiplications in 5 cycles. If there are sequential dependencies for each of the multiplications then we can only do one multiply every 5 cycles and the whole calculation will then take 25 cycles.

15.2.3 Data Hazards

- Data dependencies for instruction *j* following instruction *i*
  - Read after Write (RAW) (*true dependence*)
    - Instruction *j* tries to read before instruction *i* tries to write it
  - Write after Write (WAW) (*output dependence*)
    - Instruction *j* tries to write an operand before *i* writes its value
  - Write after Read (WAR) (*anti dependence*)
    - Instruction *j* tries to write a destination before it is read by *i*

  Can avoid hazard with register renaming
- No such thing as a Read after Read (RAR) hazard since there is never a problem reading twice

RAW is a true dependancy because we really need to read the value that instruction i writes. WAW, WAR can be avoided because j is only writing a value and does not need to wait for anything. Register renaming allows this by using extra registers to store the write value of j and hence not overwrite the value of i.

15.2.4 Register Renaming

- Use more registers than in the ISA
  - Map *architectural* registers to a larger pool of *physical* registers
  - Give each new value produced its own physical register
  - Avoid WAW and WAR hazards
- Example: Before & after renaming
  - R1 = R2 + R3 — R1 = R2 + R3
  - R4 = R1 + R5 — R4 = R1 + R5 (RAW hazard)
  - R1 = R6 + R7 — R33 = R6 + R7
  - R6 = R1 + R3 — R34 = R33 + R3

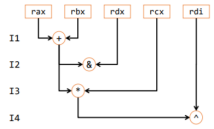
The ISA defines how many registers are available the "architectural registers". The processor itself can have a larger pool of physical registers. We dont add more registers to the ISA because we want to keep backwards compatability. There are sometimes add ons to the ISA but in general we dont want to change the contract between software and hardware.

15.2.5 Instruction Execution

Traditional view of instruction execution

- Imperative view
  - Registers are fixed storage locations
    - Individual instructions read & write them
  - Instructions must be executed in specified sequence to guarantee proper program behavior

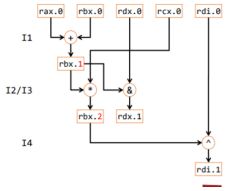
```
addq %rax, %rbx  # I1
andq %rbx, %rdx  # I2
mulq %rcx, %rbx  # I3
xorq %rbx, %rdi  # I4
```



Dataflow view of instruction execution

- Functional view
  - View each write as creating new instance of value
  - Operations can be performed as soon as operands available
  - No need to execute in original sequence

```
addq %rax, %rbx  # I1
andq %rbx, %rdx  # I2
mulq %rcx, %rbx  # I3
xorq %rbx, %rdi  # I4
```



15.2.6 Meaning for Program performance

What is the performance bound?

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency bound	**1.00**	**3.00**	**3.00**	**5.00**
Throughput bound	**0.50**	**1.00**	**1.00**	**0.50**

4 func. units for int +
2 func. units for load

2 func. units for FP *
2 func. units for load

Intel Haswell example

Instruction	Latency	Cycles/issue
Load/store	4	1
Int Add	1	1
Int Multiply	3	1
Int/Long Divide	3-30	3-30
Sngl/Dbl FP Multiply	5	1
Sngl/Dbl FP Add	3	1
Sngl/Dbl FP Divide	3-15	3-15

Instructions that can execute in parallel:
- 2 load, with address computation
- 1 store, with address computation
- 4 integer add
- 1 integer multiply
- 2 FP multiply
- 1 FP add
- 1 FP divide

But how can we make our program throughput-bound?
What can we do about the sequential dependency?

We get the latency bound by looking at the latency entries of the respective operation from the table on the right. In the original Combine4 program the add was still slower, this was because we could exploit it some more using loopunrolling. The value of the throughput bound is dependant not only on the number of functional units but also on the number of loads that can be done. We see that the throughput bound offers us better performance and hence we will want our program to be througput bound and not sequential, this can be done with reassociation

15.3 Reassociation

Loop Unrolling with Reassociation (2x1a)

- Can this change the result of the computation?
- Yes, for FP. Why?

Compare to before:

```
x = (x OP d[i]) OP d[i+1];
```

```
void unroll2aa_combine(struct vec *v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Nearly 2x speedup for Int *, FP +, FP *
- Reason: Breaks sequential dependency

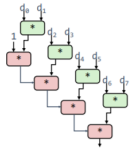
```
x = x OP (d[i] OP d[i+1]);
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Latency Bound	**1.00**	**3.00**	**3.00**	**5.00**
Throughput Bound	**0.50**	**1.00**	**1.00**	**0.50**

The reason that the change of brackets increase performance is because we remove the dependancy off the previous result, hence we allow for more efficient scheduling. For floating point values this would not be valid because FP's are not associative. If you however know that the the values wont be comprimised then you can do it. The resulting computation structure is:

- What changed:
  - Ops in the next iteration can be started early (no dependency)
- Overall Performance
  - N elements, D cycles latency/op
  - (N/2+1)*D cycles:
    CPE = D/2

```
x = x OP (d[i] OP d[i+1]);
```



15.3.1 Separate Accumulators

- Different form of reassociation

```
void unroll2a_combine(struct vec *v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- Int + makes use of two load units
- 2x speedup (over unroll2) for Int *, FP +, FP *

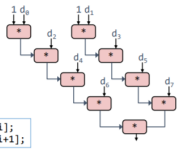
```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

Separate Accumulators

- What changed:
  - Two independent "streams" of operations
- Overall Performance
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles:
    CPE = D/2
  - CPE matches prediction!

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



This increases the performance for the add because we are overlapping operations. In the previous case we were doing two loads and then the respecive additions, whereas now we can do the loads and additions simultaneously. It does not affect the other arithmetic operations because they have a higher latency.

15.4 Combining multiple accumulators and unrolling

- Idea
  - Can unroll to any degree L
  - Can accumulate K results in parallel
  - L must be multiple of K
- Limitations
  - Diminishing returns
    - Cannot go beyond throughput limitations of execution units
  - Large overhead for short lengths
    - Finish off iterations sequentially

Unrolling & Accumulating: Int +

- Intel Haswell
- Integer addition
- Latency bound: 1.00
- Throughput bound: 0.50

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	1.27	1.01	1.01	1.01	1.01	1.01	1.01	
2		0.81		0.69		0.54		
3			0.74					
4				0.69		1.24		
6					0.56			0.56
8						0.54		
10							0.54	
12								0.56

Accumulators

Unrolling & Accumulating: Double *

- Intel Haswell
- Double FP Multiplication
- Latency bound: 5.00.
- Throughput bound: 0.50

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
2		2.51		2.51		2.51		
3			1.67					
4				1.25		1.26		
6					0.84			0.88
8						0.63		
10							0.51	
12								0.52

Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Best	0.54	1.01	1.01	0.52
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

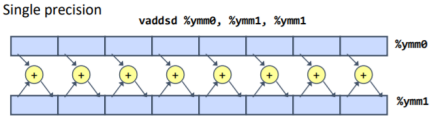
- Limited only by throughput of functional units
- Up to 42X improvement over original, unoptimized code

The compiler can do loop unrolling, but will not know which is best, hence we must make it specific in the code.

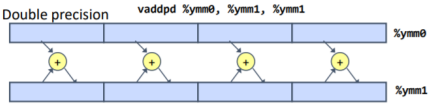
15.4.1 AVX2 SIMD operations

AVX2 SIMD operations (256-bit vectors)

- Single precision



- Double precision



- Make use of AVX Instructions (256 bits wide):

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Scalar Best	0.54	1.01	1.01	0.52
Vector Best	0.06	0.24	0.25	0.16
Latency Bound	**0.50**	**3.00**	**3.00**	**5.00**
Throughput Bound	**0.50**	**1.00**	**1.00**	**0.50**
Vector Throughput Bound	**0.06**	**0.12**	**0.25**	**0.12**

Chapter 16

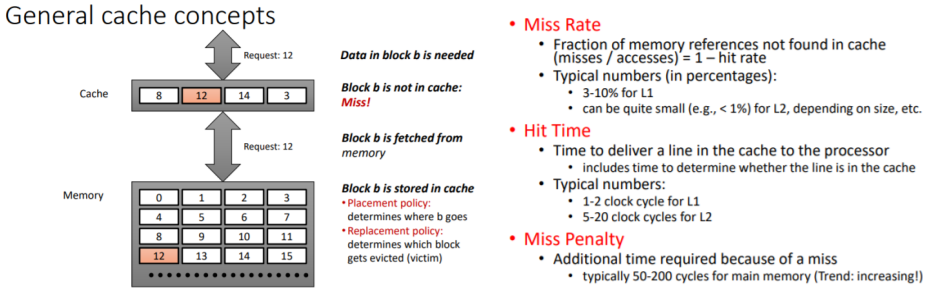
Caches

16.1 Intro

16.1.1 Processor-memory bottleneck

The CPU performance continouisly improves, but the bandwidth to access memory is evolving much slower. E.g Haswell core can process 512 Bytes/cycle but the bandwidth is 10 Bytes/cycle, hence accessing main memory results in stalls. The solution to this problem is to use a hierarchy of caches.

16.1.2 General Cache concepts



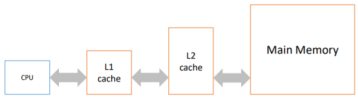
Because the huge difference between hit and miss times each percent in improved hit rate leads to enormous performance gains.

- 99% hit rate is twice as good as 97%!
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - Average access time:
    97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
    99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles
- This is why we use ***miss rate*** instead of hit rate

Another metric is "MPKI" which is misses per kilo instruction. MPKI is how many misses per thousand instructions.

16.1.3 Two-level cache performance

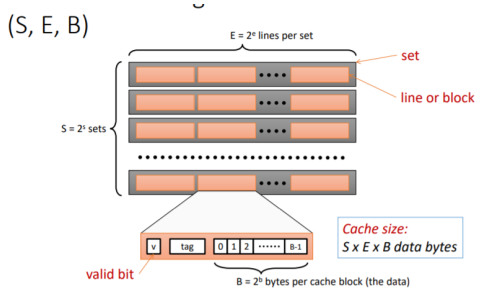
- **Average Memory Access Time** = HitTimeL1 + MissRateL1 * MissPenaltyL1
- MissPenaltyL1 = HitTimeL2 + MissRateL2 * MissPenaltyL2
- MissPenaltyL2 = DRAMaccessTime + (BlockSize/Bandwidth)



16.1.4 Types of cache misses

- Cold (compulsory) miss
  - Occurs on first access to a block
- Conflict miss
  - Most caches limit placement to small subset of available slots
    - e.g., block i must be placed in slot (i mod 4)
  - Cache may be large enough, but multiple lines map to same slot
    - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time
- Capacity miss
  - Set of active cache blocks (working set) larger than cache
- Coherency miss
  - Multiprocessor systems: see later in the course

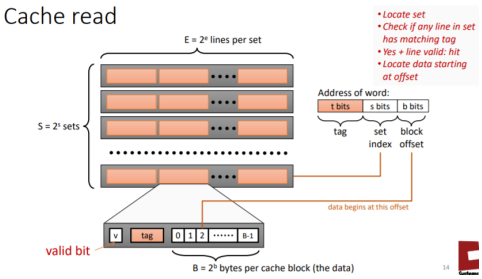
16.2 Cache organization



We need S bits to determine which set of the cache we want to access. Each set can have multiple entries (also called "ways"). Each block will have:

- **Valid bit:** Indicates wether the contents are a valid cache entry
- **Tag:** Used to determine a hit

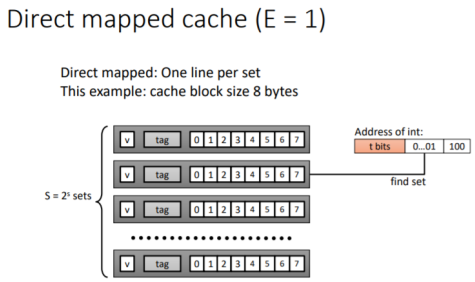
16.3 Cache reads

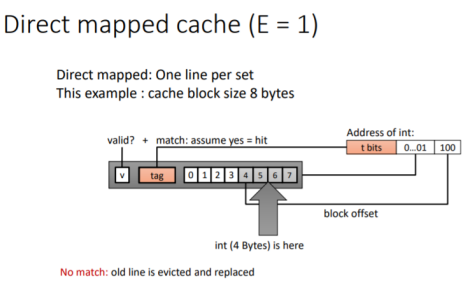


the block offset indicates which byte of the block we want to access
the set index indicates which set we want to access
Since many locations could have the same set and block access we must compare the tags to ensure we are accessing the correct element.

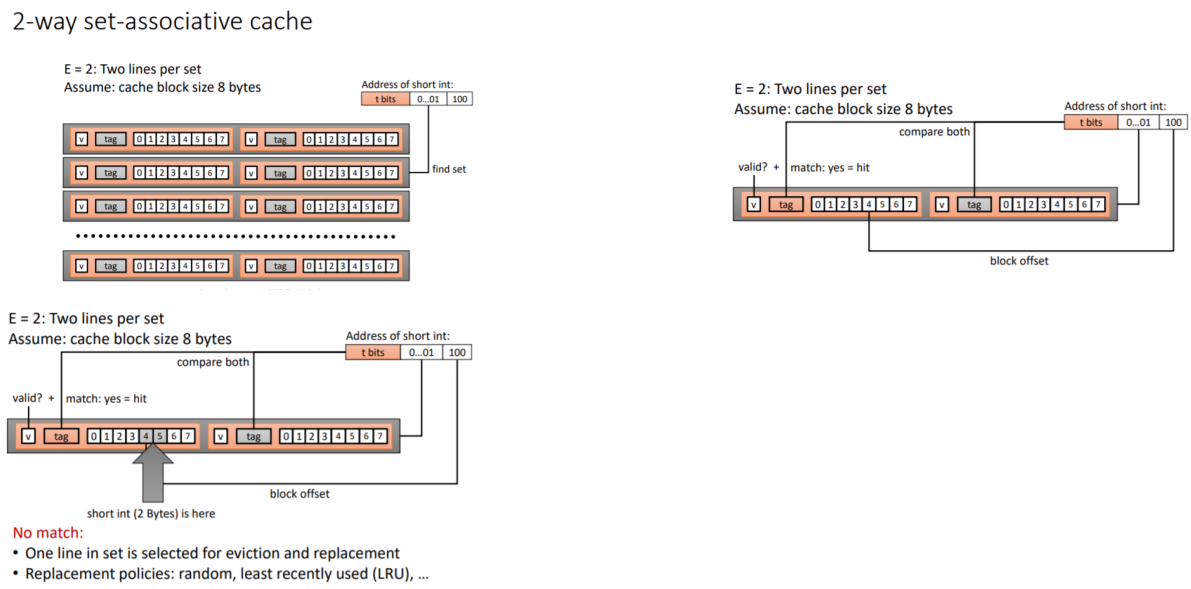
16.3.1 Direct mapped cache

We only have one way and block per set





16.3.2 2-way set-associative cache

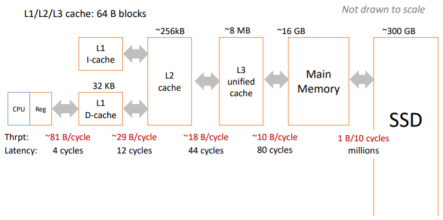


16.4 The memory hierarchy

Examples of caching in the hierarchy

Cache type	What is cached?	Where is it cached?	Latency (cycles)	Managed by
Registers	4/8-byte words	CPU core	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware
L1 cache	64-byte blocks	On-chip L1	1	Hardware
L2 cache	64-byte blocks	On-chip L2	10	Hardware
Virtual memory	4kB page	Main memory (RAM)	100	Hardware + OS
Buffer cache	4kB sectors	Main memory	100	OS
Network buffer cache	Parts of files	Local disk, SSD	1,000,000	SMB/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Intel® Core™ i7-6700K ("Skylake", 2015-)



Types of L1 caches:

- **I-cache:** Instruction Cache. There will be locality in the instructions because when we dont branch we use instructions that are right after each other.
- **D-cache:** Data Cache. there typically be locality in the data that is accessed (e.g arrays)

The Disk is persistent storage, meaning if power is lost data is preserved. The Caches are all volatile storage, hence losing power means losing data. SSD are more efficient than Disks.

16.5 Cache writes

16.5.1 Write-hit

Means we have this block in the cache and we are trying to write. There are two options:

- Write-through
  - Write immediately to memory
  - Memory is always consistent with the cache copy
  - Slow: what if the same value (or line!) is written several times
- Write-back
  - Defer write to memory until replacement of line
  - Need a dirty bit
    - ⇒ indicates line is different from memory
  - Higher performance (but more complex)

16.5.2 Write-miss

- Write-allocate (load into cache, update line in cache)
  - Good if more writes to the location follow
  - More complex to implement
  - May evict an existing value
  - Common with write-back caches
- No-write-allocate (writes immediately to memory)
  - Simpler to implement
  - Slower code (bad if value subsequently re-read)
  - Seen with write-through caches

16.5.3 Other hardware cache features

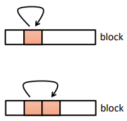
- Unified
  - Serves both instruction and data fetches
- Private
  - Only one core uses this cache
- Shared
  - Multiple cores share the cache
- Inclusive
  - Anything in this cache is also in every lower-level cache
- Exclusive
  - Anything in this cache is not in any lower-level cache

16.6 Cache optimizations

16.6.1 Locality

Why caches work: *locality*

- Programs tend to use data and instructions with addresses near or equal to those they have used recently



- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future
- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time

- Data:
  - Temporal: sum referenced in each iteration
  - Spatial: array a[] accessed in stride-1 pattern
- Instructions:
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence

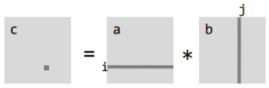
```
sum = 0;
for (i = 0; i < n; i++)
  sum += a[i];
return sum;
```

- Being able to assess the locality of code is a crucial skill for a programmer

16.6.2 Locality with matrix multiplication

Example: matrix multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n)
{
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n+j] += a[i*n + k]*b[k*n + j];
}
```



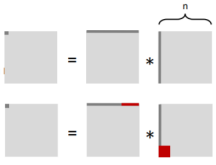
Cache miss analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
- Second iteration:
  - Again: n/8 + n = 9n/8 misses
- Total misses:
  - $9n/8 * n^2 = (9/8) * n^3$

Cache miss analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than
- First iteration:
  - n/8 + n = 9n/8 misses

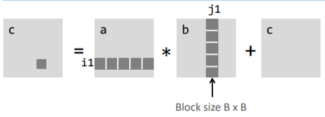
Afterwards in cache: (schematic)



16.7 Blocking

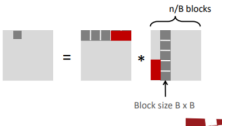
Blocked matrix multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



Cache miss analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$
  - **B is a multiple of cache block**
- Second (block) iteration:
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$
- Total misses:
  - $nB/4 * (n/B)^2 = n^3/(4B)$



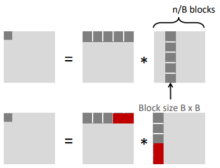
Summary

- No blocking: $(9/8) * n^3$
- Blocking: $1/(4B) * n^3$
- Suggests: max(B) s.t. $3B^2 < C$
  (can possibly be relaxed a bit, but there is a limit for B)

- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used O(n) times!
  - But program has to be written properly
- "Systems code" is usually much more subtle...

Cache miss analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$
  - **B is a multiple of cache block**
- First (block) iteration:
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$
    (omitting matrix c)



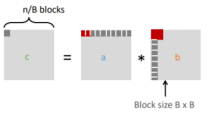
Cache miss analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$

If B ≤ cache block...

- Total misses:
  $(Bn/8 + n) * (n/B)^2$
  (Misses in matrix a per iteration; Misses in matrix b per iteration; Number of iterations)
  $= (1/(8B) + 1/B^2) * n^3$
  (omitting matrix c)

Note: if B=1, same as no blocking: $(9/8) * n^3$ misses
if B=8, we have $n^3/32$ misses = $n^3/(4B)$



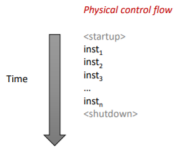
Chapter 17

Exceptions

17.1 Intro

17.1.1 Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU's *control flow* (or flow of control)



17.1.2 Altering the control flow

- Up to now: mostly two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return

  Both react to changes in ***program state***

- Insufficient for a useful system:
  Difficult to react to changes in ***system state***
  - data arrives from a disk or a network adapter
  - instruction divides by zero
  - user hits Ctrl-C at the keyboard
  - System timer expires

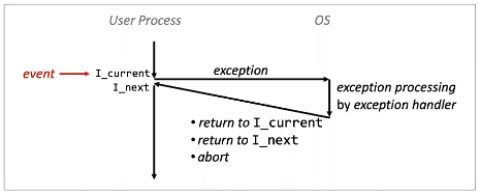
These alternative methods to alter control flow are known as Exceptional control flow

17.1.3 Exceptional control flow

- Exists at all levels of a computer system
- Low level mechanisms
  - Hardware exceptions
    - change in control flow in response to a system event (i.e., change in system state)
  - Combination of hardware and OS software
- Higher level mechanisms
  - Process context switch
  - Signals
  - Nonlocal jumps: `setjmp()/longjmp()`
  - Implemented by either:
    - OS software (context switch and signals)
    - C language runtime library (nonlocal jumps)
  - Language-level exceptions (Java, etc.)

17.1.4 Exceptions

An exception is a transfer of control to the OS in response to some event (i.e a change in processor state)



Examples: div by 0, machine check, page fault, I/O request completes, Crtl-C

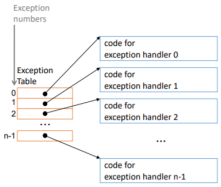
- A *synchronous* exception occurs as a results of executing an instruction.
- An *asynchronous* exception occurs as a result of events that are external to the processor.

Type of exception	Cause	Async/Sync
Interrupt	Signal from I/O device	Async
Trap	Intentional exception	Sync
Fault	Potentially recoverable error	Sync
Abort	Nonrecoverable error	Sync

17.2 Exception vectors and kernel mode

17.2.1 Exception vectors

- At boot time, OS allocates and initializes the *exception table*
  - The table's base address is stored in the Exception Table Base Register
- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs



Each exception table entry points to the beginning address of the code which must be executed for this particular exception.

17.2.2 x86 ecveption vectors

0	Divide error
1	Debug exception
2	Non-maskable interrupt (NMI)
3	Breakpoint
4	Overflow
5	Bounds check
6	Invalid opcode
7	Coprocessor not available
8	Double fault
9	Coprocessor segment overrun (386 or earlier)
A	Invalid task state segment
B	Segment not present
C	Stack fault
D	General protection fault
E	Page Fault
F	Reserved
10	Math fault
11	Alignment check
12	Machine check
13	SIMD floating point exception
14-1F	Reserved to Intel
20-FF	Available for external interrupts

17.2.3 Kernel

A Kernel is a part of the operating system. It is a part which runs in privileged mode i.e you have access to resources to registers and services which you dont have in user mode. Only time Kernel mode is used if a system call is done (e.g when an exception is called there is a context switch into kernel mode).

Kernel mode

- Exceptions cause a switch to ***kernel mode***
  - Also: supervisor mode, privileged mode, ring 0, etc.
- Things look **very different**:
  - Access to system state (virtual memory, etc.)
  - Some exceptions are disabled
  - Some new instructions and registers
  - MMU behavior may change
- Details vary between processors
  - Even in the same architecture

The Kernel

- Most operating systems have a ***kernel***
  = the part of the OS that runs in kernel mode
- Think of the kernel as:
  - A set of trap handling functions (always)
  - A set of threads in a special address space (sometimes)
  - Code to create the illusion of user-space processes (always)
- Many ways to structure the kernel and OS
  - Microkernels, monolithic kernels, multikernels, etc.
  - See the OS course next year...

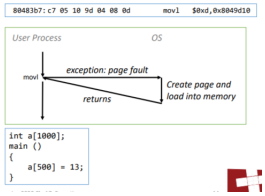
17.3 Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - Intentional
    - Examples: ***system calls***, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - ***Aborts***
    - unintentional and unrecoverable
    - Examples: parity error, machine check
    - Aborts current program

17.3.1 Fault Examples

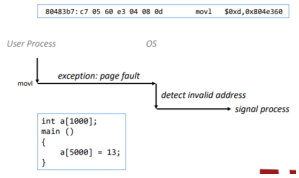
Fault example: page fault

- User writes to memory location
- That portion (page) of user's memory is currently on disk
- Page handler must load page into physical memory
- Returns to faulting instruction
- Successful on second try



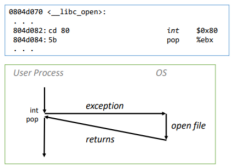
Fault example: invalid memory reference

- Page handler detects invalid address
- Sends SIGSEGV signal to user process
- User process exits with "segmentation fault"



Trap example: opening a file

- User calls:
  `open(filename, options)`
- Function open executes system call (e.g. `int` instruction)
- OS must find or create file, get it ready for reading or writing
- Returns integer file descriptor



A file descriptor is an integer which uniquely identifies a file

17.4 Asynchronous Exceptions (Interrupts)

- Caused by events *external* to the processor
  - Indicated by setting the processor's interrupt pin
  - Handler returns to "next" instruction
- Examples:
  - I/O *interrupts*
    - hitting Ctrl-C at the keyboard
    - arrival of a packet from a network
    - arrival of data from a disk
  - Hard reset interrupt
    - hitting the reset button
  - Soft reset interrupt
    - hitting Ctrl-Alt-Delete on a PC

- CPU **Interrupt-request line** triggered by I/O device
  - Might be edge- or level-triggered
- **Interrupt handler** receives interrupts
- **Maskable** to ignore or delay some interrupts
- Interrupt vector to dispatch interrupt to correct handler
  - Based on priority
  - Some **nonmaskable**
- Interrupt mechanism also used for exceptions

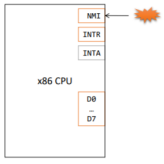
17.4.1 x86 interrupts

There are two interrupt pins:

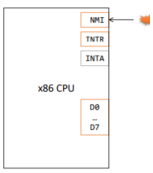
- INTR: interrupt request
- NMI: non-maskable interrupt

NMI

- NMI asserted ⇒
  - CPU completes current instruction
  - Issues Exception #2
  - *Always*.
- Cannot be disabled by the processor.
- Reserved for
  - Major hardware faults, e.g. memory parity error
  - "Watchdog" timer



- What caused the NMI?
- No obvious way to tell: the OS (`INT 0x2` handler) must poll potential sources
- Typical machine: o(100s) devices
  - ⇒ slow reaction, inefficient
- Might get a second NMI while first is still being handled!



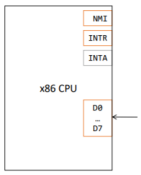
A "watchdog" timer is a timer which must be periodically reset, if this timer reaches zero, the processor enters a "panic state" (blue screen) and stops working. Exceptions are indexed from 0. Since there is only one pin there is no obvious

way to tell which device caused the interrupt, and hence it must check each device for the source of the interrupt (called polling). NMI interrupts are not so common, IRQ interrupts are more common

IRQ

Interrupt requests are sent into the INTR pin. The CPU then acknowledges that the interrupt was recieved on the INTA pin (hence we decide which device we deal with) when the device confirms it sends data to the data bits. The Data bits allow the device to send us its interrupt vector, allowing us to except different kinds of interrupts.

- Interrupt request ⇒
  - Can be disabled by `IE` status flag (`CLI`/`STI` instructions to clear/set interrupt flag)
  - If enabled, complete current instruction, then:
- CPU acknowledges using INTA pin
- Interrupt vector is then supplied to the CPU via the data bus
- CPU issues exception # from the vector

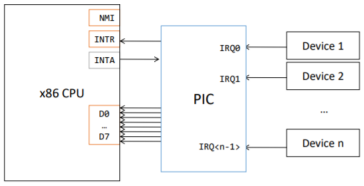


17.5 Interrupt controllers

17.5.1 Programmable Interrupt Controllers (PIC)

Instead of having Devices which directly connect to the INTR pin like we have for the NMI pin, we have the devices connected to the PIC. This allows us to manage which devices asked for the interrupts and then transmit the proper data to the CPU

Programmable Interrupt Controllers



- Map physical interrupt pins (from device, slot, socket, etc.) to interrupt vectors
  - Mapping picked by the OS ⇒ flexible
- Buffer simultaneous interrupts
  - Deliver each interrupt vector separately
  - Won't lose some device's interrupt
- Prioritize interrupts
  - Allow some device IRQs to interrupt others'
  - Again, selected by the OS
- Selectively mask any individual device's interrupts
  - Useful for high-interrupt rate devices
  - Useful at boot!

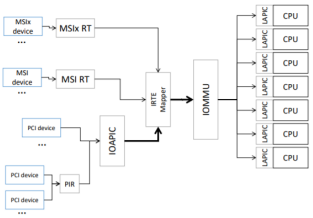
In the case of multiple interrupts we can use a buffer to store them.

17.5.2 Modern PICs

Modern PICs

- Every processor in a modern PC has a "Local APIC"
  - "Local Advanced Programmable Interrupt Controller"
  - PIC → APIC → LAPIC → xAPIC → x2APIC → …
- Additional capabilities:
  - Inter-processor interrupts
  - Programmable timer
  - Sophisticated interrupt scheduling
  - Etc.
- Integrated onto same die as the core itself
- Also: IOxAPIC / MSI-X
  - Off-chip, attached to device I/O interconnect

Modern PCs (simplified!)

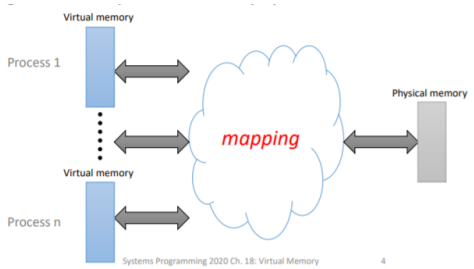


Chapter 18

Virtual Memory

18.1 Address Translation:

Each Process gets its own private memory space



18.1.1 Address Spaces:

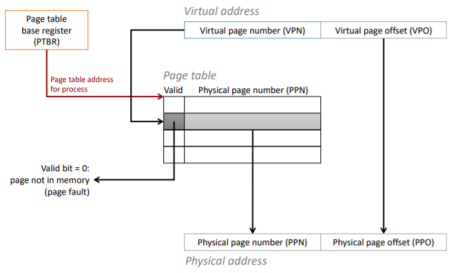
- Linear address space: Ordered set of contiguous non-negative integer addresses:
  {0, 1, 2, 3 ... }
- Virtual address space: Set of N = 2^n virtual addresses
  {0, 1, 2, 3, ..., N-1}
- Physical address space: Set of M = 2^m physical addresses
  {0, 1, 2, 3, ..., M-1}
- Clean distinction between data (bytes) and their attributes (addresses)
- Each object can now have multiple addresses
- Every byte in main memory:
  one physical address, one (or more) virtual addresses

One physical address can map to multiple addresses in Virtual memory.
Virtual Memory requires the OS and hardware:

- **MMU (Memory Managing Unit):** This hardware is dedicated to translating virtual to physical addresses using a table in hardware which is managed and populated by hardware. Before memory can be accessed the virtual address needs to be translated to physical address. The memory hierarchy also applies to this.

18.1.2 Address Translation with page table

Page: A page is a block of memory. This is the granularity of memory (corresponds to a cache block).



The page table is stored in a register in the CPU. The page table can be indexed with the virtual the virtual address. The page table tells of if a particular page is valid. The Virtual page offset specifies which byte in a page we want. The

Virtual page number needs to be translated into a physical page number. The Virtual page offset remains the same for the physical address.

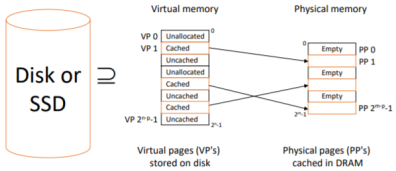
18.2 Uses of virtual memory

18.2.1 Benefits

- **Efficient use** of limited main memory (RAM)
  - Use RAM as a cache for the parts of a virtual address space
    - some non-cached parts stored on disk
    - some (unallocated) non-cached parts stored nowhere
  - Keep only active areas of virtual address space in memory
    - transfer data back and forth as needed
- **Simplifies** memory management for programmers
  - Each process gets the same full, private linear address space
- **Isolates** address spaces
  - One process can't interfere with another's memory
    - because they operate in different address spaces
  - User process cannot access privileged information
    - different sections of address spaces have different permissions

18.2.2 VM as a tool for caching

- Virtual memory: array of $N = 2^n$ contiguous bytes
  - think of the array (allocated part) as being stored on disk
- Physical main memory (DRAM) = cache for allocated virtual memory
- Blocks are called pages; size = 2^p



Pages can be in 3 different states:

- **Unallocated:** A page which hasent yet been used by the program. These are not present on disc.
- **Cached:** A page which has already been allocated. A Cached page resides in main memory
- **Uncached:** A page which has already been allocated. But does not reside in main memory. This page can be found on disc.

When trying to access and Uncached page, we will get a page fault. And the page fault handler will bring in that page from memory and retry the instruction, which will then be mapped to memory.

18.2.3 DRAM cache organization

SRAM: Memory type used for L1,L2,L3 caches.

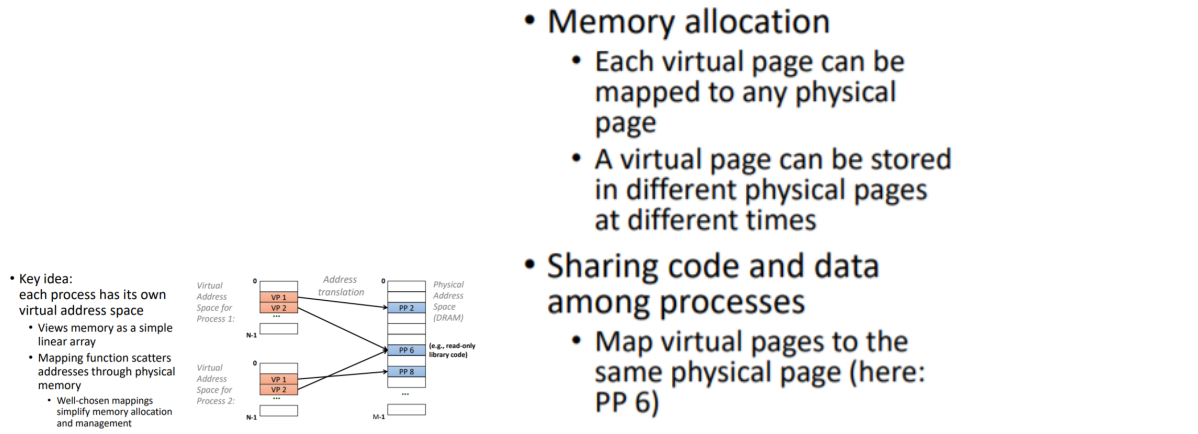
- DRAM cache organization driven by the enormous miss penalty
  - DRAM is about ***10x*** slower than SRAM
  - Disk is about ***10,000x*** slower than DRAM
    - For first byte, faster for next byte
- Consequences
  - Large page (block) size: typically 4-8 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a "large" mapping function – different from CPU caches
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through

- Virtual memory works because of locality
- At any point in time, programs tend to access a set of active virtual pages called the *working set*
  - Programs with better temporal locality will have smaller working sets
- If (working set size < main memory size)
  - Good performance for one process after compulsory misses
- If (SUM(working set sizes) > main memory size)
  - *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously

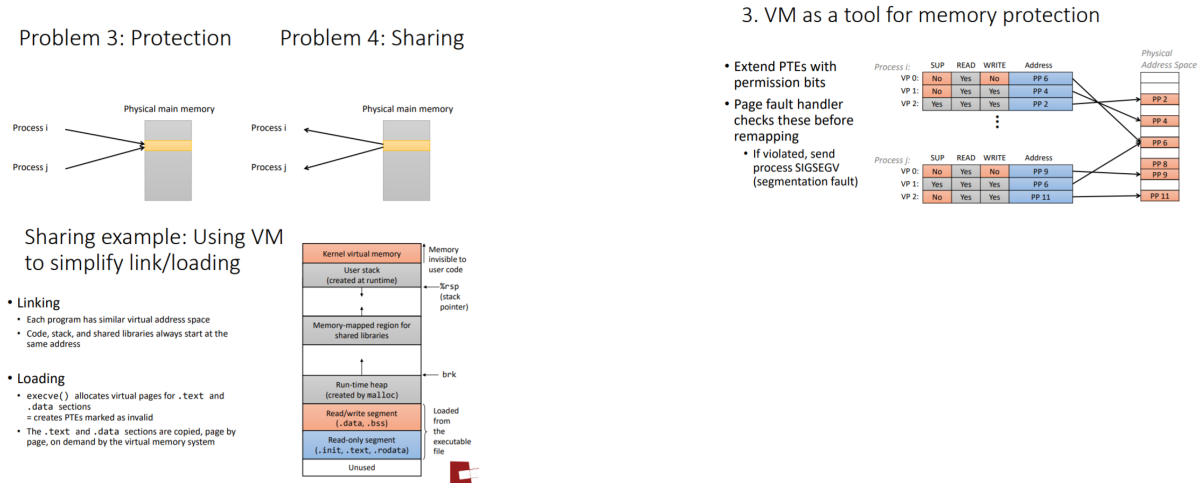
Paging/Swapping Bringing in pages from Disk. The reason that the page sizes are so large, is that accessing disc is so slow. So we try to compensate by bringing in more data when accessing it.

18.2.4 Memory Management

The idea of mapping multiple processes to physical memory.



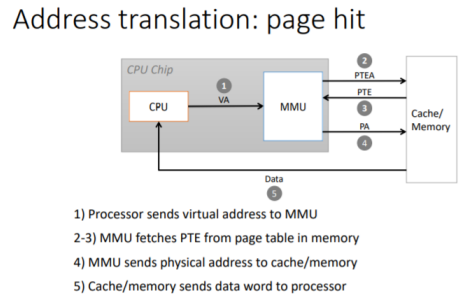
18.2.5 Protection and Sharing



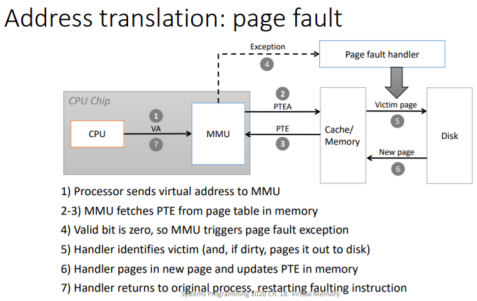
Supervisor mode permission are pages which can only be accessed by the kernel.

18.3 The address translation process

18.3.1 Page Hit:



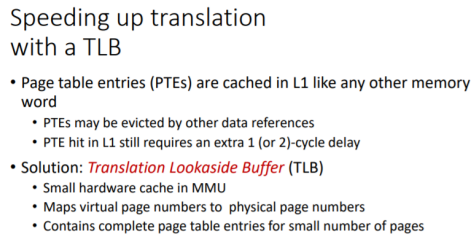
18.3.2 Page Fault:



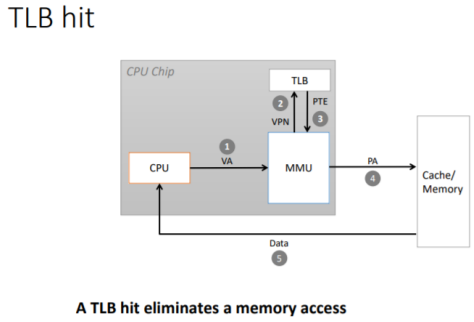
The Page fault handler is implemented in software. The cache can be indexed by virtual or physical addresses. Typically the cache is indexed by physical addresses, it is a bit slower because of translation, but is simpler to implement.

18.4 Translation lookaside buffer

Caching for the page table translations.

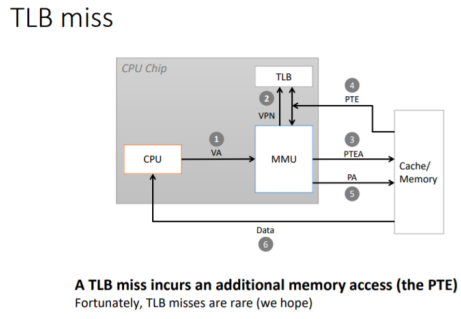


18.4.1 TLB hit



When considering multiple processes we can either have identification bits for the processes. An alternative is to flush the TLB when context switching.

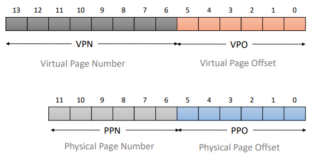
18.4.2 TLB Miss



18.5 Simple Memory system Example

A simple memory system

- Addressing
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes



18.5.1 Page Table

Simple memory system: page table

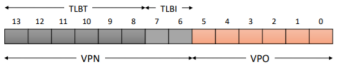
- Only show first 16 entries (out of 256)

VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

18.5.2 TLB

- 16 entries
- 4-way associative

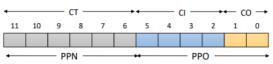


Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

- **TLBT (TLB-tag):** Used to check wether the page we are actually accessing is stored in the TLB
- **TLBI (TLB- index):** Used to index into the right set of the TLB

18.5.3 Cache

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped

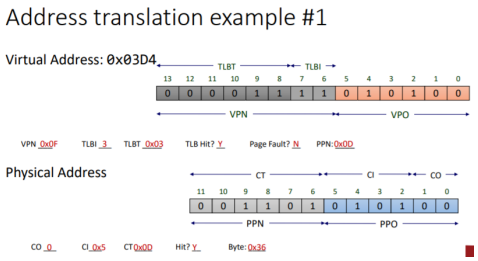


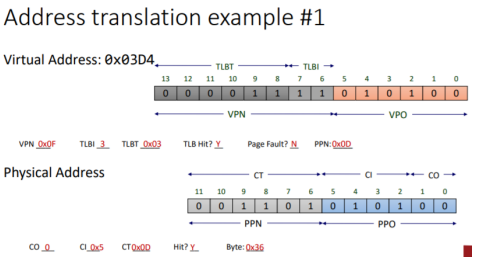
Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

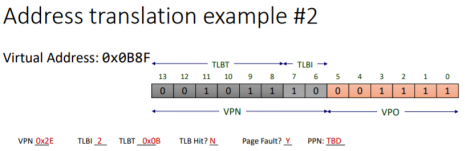
Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

- **PPN:** Physical Page Number
- **PPO:** Physical Page Offset
- **CO:** Cache Offset: Number of bits is determined by the cache block size
- **CI:** Cache Index: Tells us which set to go to.
- **CT:** Cache Tag: Tells us if we have a hit.

18.5.4 Address translation examples



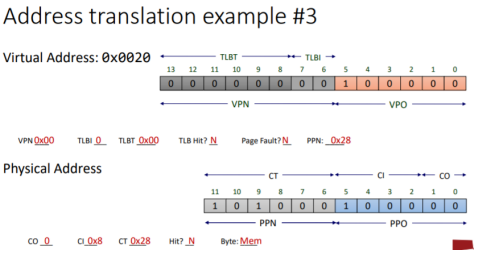




For the last example we dont have a hit in the TLB. So we check if the page table entry 00 which has PPN 28. Hence we dont have a page fault. A page fault is not the same as a miss. A fault refers to a miss in DRAM for the page itself.

18.6 Multi-level page tables

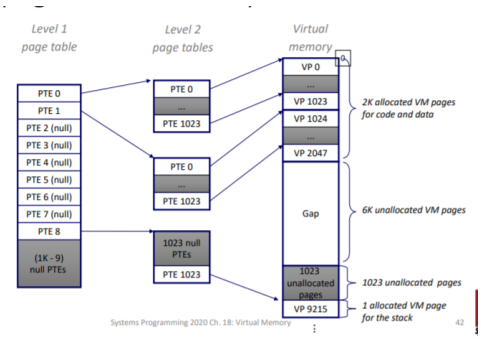
18.6.1 Terminology



18.6.2 Linear Page Table size

- Given:
  - 4KB (2^{12}) page size
  - 48-bit address space
  - 8-byte PTE
- How big a page table do you need?
- Answer: 512 GB!
  - $2^{48} / 2^{12} * 2^3 = 2^{39}$ bytes

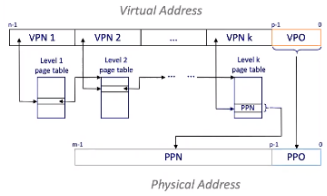
18.6.3 2-level page table hierarchy



We use the first bits of the VM address to index the Page directory table. This points to a different Level of smaller page tables. The next bits index the page table that was pointed to by the page directory table which in turn points to an

Address in VM. This reduces page Table size because a large range of addresses are represented by much fewer numbers and we only access allocate page table space on demand. The user tries to access a location in memory. If that hasn't been accessed before then the Page Directory Table will have a nuller pointer and hence a 2nd level page table is created.
This idea can be extended to a k-level page table: We use the virtual page number to index into the multi level page tables. We take the virtual page number which is the top bits. The bottom bits are the virtual page offset. We divide the top bits by k (i.e the number of levels we have). The top 1/k bits will be used to index the first page table. The address we get from the first table will be the base of the second level table. And the next bits in the VPN will serve as a index from the second level page table. This continues until we reach the final level table which will give the PPN

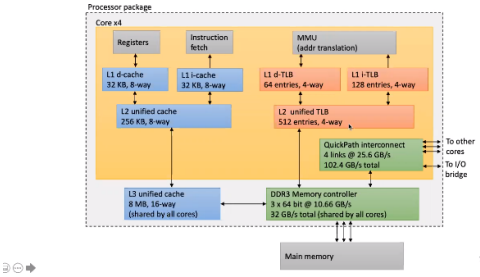
Translating with a k-level page table



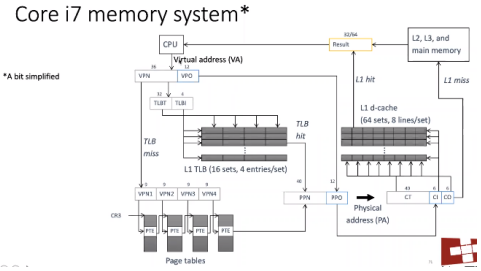
A miss on a multi level page table has a longer miss penalty called a "page lock".

18.7 Case study of the core i7 processor

A typical core i7 processor



Core i7 memory system*



The size of the physical adress space is proportional to the amount of RAM that is expected to be accessible on the machine. If one of the levels is a null pointer, then we create this level and continue this for the whole hierarchy for this address range we are trying to access.

x86-64 paging

- 48-bit virtual address
  - 256 terabytes (TB)
  - Not yet ready for full 64 bits
    - Nobody can buy that much DRAM yet
    - Mapping tables would be huge
    - Multi-level array map may not be the right data structure
- 52-bit physical address = 40 bits for PPN
  - Requires 64-bit table entries
- Keep older x86 4KB page size (including for page tables)
  - (4096 bytes per PT) / (8 bytes per PTE) = 512 entries per page

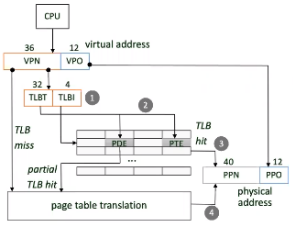
18.7.1 Core i7 TLB

- TLB entry (not all documented, so this is speculative):

  - **V:** indicates a valid (1) or invalid (0) TLB entry
  - **TLBTag:** disambiguates entries cached in the same set
  - **PPN:** translation of the address indicated by index & tag
  - **G:** page is "global" according to PDE (Page Directory Entry), PTE – do not evict on context switch
  - **S:** page is "supervisor-only" according to PDE, PTE
  - **W:** page is writable according to PDE, PTE
  - **D:** PTE has already been marked "dirty" (once is enough)
- Structure of the data TLB:
  - 16 sets, 4 entries/set

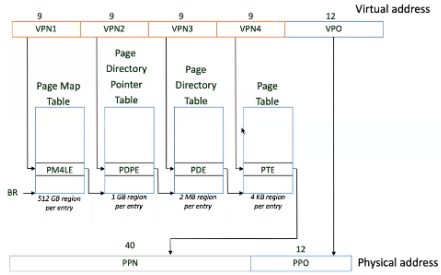
the dirty bit indicates that the value has been changed and has a different value than whats on disc. Hence when evicting this entry we also update the disc. The Page Directory refers to the higher levels of page tables. The Page Directory entry is the highest level in the page table.

18.7.2 Translating with the i7 TLB

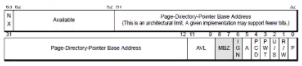


1. Partition VPN into TLBT and TLBI.
2. Is the PTE for VPN cached in set TLBI?
3. *Yes:* Check permissions, build physical address
4. *No:* Read PTE (and PDE if not cached) from memory and build physical address

18.7.3 x86-64 paging

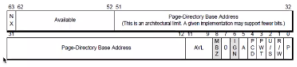


Page-Map Level 4



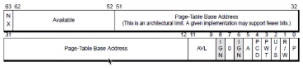
- **Page-Directory-Pointer Base Address**: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)
- **Avail**: These bits available for system programmers
- **A:** accessed (set by MMU on reads and writes, cleared by software)
- **PCD:** cache disabled (1) or enabled (0)
- **PWT:** write-through or write-back cache policy for this page table
- **U/S:** user or supervisor mode access
- **R/W:** read-only or read-write access
- **P:** page table is present in memory (1) or not (0)

Page Directory Pointer Entry (level 3)



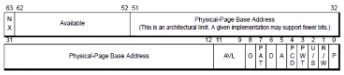
- **Page-Directory Base Address**: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)
- **Avail**: These bits available for system programmers
- **A:** accessed (set by MMU on reads and writes, cleared by software)
- **PCD:** cache disabled (1) or enabled (0)
- **PWT:** write-through or write-back cache policy for this page table
- **U/S:** user or supervisor mode access
- **R/W:** read-only or read-write access
- **P:** page table is present in memory (1) or not (0)

Page Directory Entry (level 2)



- **Page table physical base address**: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)
- **Avail**: These bits available for system programmers
- **A:** accessed (set by MMU on reads and writes, cleared by software)
- **PCD:** cache disabled (1) or enabled (0)
- **PWT:** write-through or write-back cache policy for this page table
- **U/S:** user or supervisor mode access
- **R/W:** read-only or read-write access
- **P:** page table is present in memory (1) or not (0)

Page Table Entry (level 1)



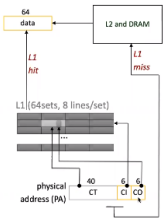
- **Physical Page base address:** 40 most significant bits of physical page address (forces pages to be 4 KB aligned)
- **Avail:** available for system programmers
- **G:** global page (don't evict from TLB on task switch)
- **PAT:** Page-Attribute Table – allows the OS to control how individual pages are cached
- **D:** dirty (set by MMU on writes)
- **A:** accessed (set by MMU on reads and writes)
- **PCD:** cache disabled or enabled
- **PWT:** write-through or write-back cache policy for this page
- **U/S:** user/supervisor
- **R/W:** read/write
- **P:** page is present in physical memory (1) or not (0)

18.8 Core i7 cache

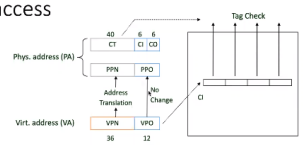
18.8.1 cache access

L1 cache access

- Partition physical address: CO, CI, and CT
- Use CT to determine if line containing word at address PA is cached in set CI
- **No:** check L2 (then L3...)
- **Yes:** extract word at byte offset CO and return to processor



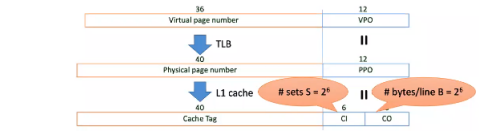
Speeding up L1 access



- Bits that determine CI *identical* in virtual and physical address
- Can index into cache *while* address translation taking place
- Generally we hit in TLB, so PPN bits (CT bits) available next
- "Virtually indexed, physically tagged"
- Cache carefully sized to make this possible

18.8.2 Cache size

Why the cache doesn't get bigger



- $\log_2$(cache size) = bits(CI) + bits(CO) + $\log_2$(associativity)
- For Core i7: 6 + 6 + 3 = 15 ⇒ 32kB
- For performance: bits(CI) + bits(CO) ≤ bits(VPO)

As seen in the formula we can increase the cache size by increasing the associativity. This has the tradeoff that we must do more tag comparisons, and hence more hardware is needed to keep this in parallel.

18.9 Caches revisited

Until now we have assumed the cache only sees a virtual or physical address. But indexing and tagging can use different addresses:

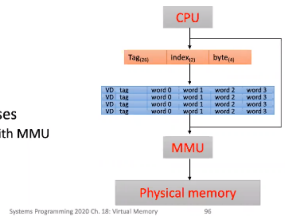
- Virtually-indexed, virtually tagged (VV)
- Virtually-indexed, physically tagged (VP)
- Physically-indexed, virtually tagged (PV)
- Physically-indexed, physically tagged (PP)

Virtually-indexed means you index the cache based on bits available from the virtual address i.e before translation. Virtually tagged means that the tag you use in the cache to check if this is really a hit comes from either the virtual address in a virtually tagged cache or from the physical address in a physically tagged cache after translation.
PV is complex and rare and hence wont be discussed in further detail.
PP is the most straightforward implementation. Its disadvantage though is that we must wait for the translation to happen before we can access the tag.

18.9.1 Virtually-indexed, virtually tagged

Virtually indexed, virtually tagged

- Also called:
  - Virtually addressed
- Also (misleadingly!)
  - Virtual cache
  - Virtual address cache
- Only uses virtual addresses
  - Operates concurrently with MMU



Systems Programming 2020 Ch. 18: Virtual Memory 96

Virtually-indexed cache issues

- Homonyms: same names for different data
- VA used for indexing is context dependent
  - Same VA refers to different PAs
  - Tag does not uniquely identify data
  - OS must prevent this!
- Homonym prevention:
  - Flush cache on context switch
  - Force non-overlapping address-space layout
  - Tag VA with address-space ID (ASID)

With a virtually indexed cache, we can access the cache at the same time as the MMU does the translation.

Summary: VV caches with keys

- Add *address space identifier* (ASID) part of tag
- On access compare with CPU's ASID register
- Removes homonyms, creates synonyms (different VAs map to same PA)
  - Potentially better context switching performance
  - ASID recycling still requires a cache flush
- Doesn't solve synonym problem
  - (but that's less serious)
- Doesn't solve write-back problem

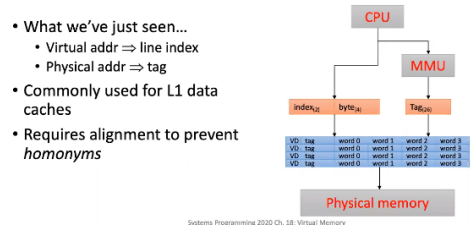
ASIDs much more common in *TLBs*!

Summary: VV caches

- Fastest: don't rely on TLB for retrieving data
  - Still need TLB lookup for protection
  - Or other mechanism to provide protection
- Suffer from synonyms and homonyms
  - Requires flush on context switch
    - Makes context switches expensive
    - May even be required on kernel ⇒ user switch
  - ... or guarantee of no synonyms or homonyms
- Require TLB lookup for writeback!
- Used for i-cache on many architectures
  - Alpha, Pentium 4, etc.
- Rare as d-cache today...
  - Used on i860, ARM7/ARM9/StrongARM/Xscale...

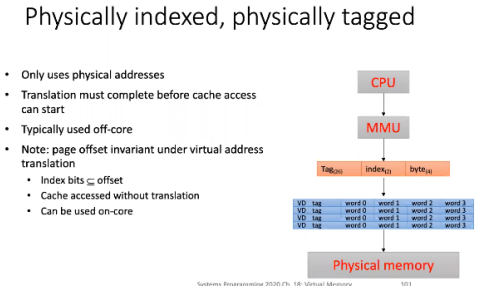
Synonyms are a problem because we might have multiple occurences of the same block in cache. This is a problem when writing because this would cause inconsistencies because the same block would be written and not written.

18.9.2 Virtually-indexed, physically tagged



The CPU can start indexing the cache, while the MMU does the translation, but we must wait for the MMU to generate the tag in order to start comparisons in the cache.

18.9.3 Physically-indexed, physically tagged



Slow but easy to use as there are no homonyms or synonyms. Frequently used as L2 or L3 cache.

18.9.4 Write Buffers