CSE 351 Notes

Contents

1 Memory & Data

1.1 Base Representation

A numeral represents a number using a series of digits in a particular base. Each digit is one of a fixed set of symbols that represent values between 0 and $\text{base} - 1$

- The decimal value of the digit d in position i in base b is $d \times b^i$
- The digit in position i is the $(i+1)^{\text{th}}$ digit from the right
- A numeral with x digits in base b can represent at most b^x distinct values

1.2 Common Bases

- Base 2 representation is called binary, identified by the prefix 0b
- Base 10 representation is called decimal, identified by the lack of a prefix
- Base 16 representation is called hexadecimal, identified by the prefix 0x

1.3 Word Size

The word size of a machine is the number of bits used to represent an address in memory

1.4 Memory

Memory acts as a local pool of data storage for the CPU during the execution of instructions. Memory is byte-oriented in that we treat memory as a large array of bytes

1.5 Addresses

An address is a unique hexadecimal number assigned to each individual byte in memory

- An address is used to specify which data in memory to access

1.6 Aligned Addresses

The address of a chunk of memory is aligned if its address is a multiple of its size

1.7 Address Space

The address space is the set of all addresses in memory

- The size of the address space is equal to the number of distinct addresses

1.8 Significant Bits

- The left-most bit in a chunk of memory is the most-significant bit
- The right-most bit in a chunk of memory is the least-significant bit

1.9 Endianness

- In a big-endian machine, the most significant byte of data is stored at the lowest address
- In a little-endian machine, the least significant byte of data is stored at the lowest address

1.10 Pointers

Pointers are special variables that store addresses

- Since the length of an address is the word size, the size of a pointer is also the word size
- The address-of operator (`&`) returns the address of a variable in memory
- The dereference operator (`*`) accesses the data at the location of the pointer
- `type*` denotes the address of a variable of type `type`

1.11 Null

`null` is a symbolic constant that is used for pointers. `null` evaluates to zero and indicates a pointer to nothing

- Dereferencing `null` results in a runtime error

1.12 Pointer Arithmetic

Pointer arithmetic takes the size of the data type being pointed to into account by automatically scaling the arithmetic operation

- i.e. Given `int* p`, the operation `p += 1` increases the value of `p` by 4, the size of `int`
- i.e. Given `long* p`, the operation `p += 1` increases the value of `p` by 8, the size of `long`
- i.e. Given `int* p1` and `int* p2`, the operation `p2 - p1` returns the number of `int` values that fit between the two addresses

1.13 Arrays

An array is a set of contiguous locations in memory that store the same type of data object

- Declaring an array follows the syntax `type array_name[num];` and sets aside `num * sizeof(type)` bytes of consecutive memory
  - `num` is the number of elements in the array

1.14 Array Subscript Notation

`array_name[n]` refers to the n^{th} element of the array, and is actually accessed via `*(array_name + n)`

- In the C programming language, `n` can be smaller than 0 or larger than `num`
  - This will access an address beyond the scope of the array

1.15 Strings

Strings in the C programming language are represented by arrays of characters that are terminated by the null character

- i.e. `char str[] = "hello"` will allocate a character array of size 6 with the following content

'h'	'e'	'l'	'l'	'o'	'\0'

1.16 String Literals

A string literal is a sequence of characters surrounded by double quotes (i.e. `"hello world"`) and automatically stored in memory as an array of characters, terminated by '`\0`'

1.17 Data Representations

C Data Type	x86-64 (bytes)	x86-64 (hex digits)
`bool`	1	2
`char`	1	2
`short int`	2	4
`int`	4	8
`float`	4	8
`long int`	8	16
`double`	8	16
`long long`	8	16
`long double`	16	32
`pointer *`	8	16

1.18 Bitwise Operators

- AND $(\&)$ outputs a 1 if and only if both input bits are 1
- OR $(|)$ outputs a 1 if and only if at least one of the input bits is 1
- XOR $(\hat{})$ outputs a 1 if and only if exactly one of the input bits is 1
- NOT $(\sim)$ outputs the opposite of the input bit

1.19 Casting

Type casting is the conversion of data of one type into another type

- An implicit cast is done by the compiler when there is a type mismatch in an assignment statement or argument assignment, but there is a well-defined conversion between the types
- An explicit cast can be performed by the programmer by prepending the new data type in parentheses to the expression, i.e. `(type) data`

1.20 Unsigned Integers

Unsigned integers represent non-negative integers, identified by the suffix u

- An unsigned integer numeral with n bits can represent integers from 0 to $2^n - 1$
- UMin represents the smallest value that an unsigned integer numeral can take
- UMax represents the largest value that an unsigned integer numeral can take

1.21 Signed Integers

Signed integers represent positive and negative integers, identified by the suffix t

- A signed integer numeral with n bits can represent integers from -2^{n-1} to $2^{n-1} - 1$
- TMin represents the smallest value that a signed integer numeral can take
- TMax represents the largest value that a signed integer numeral can take
- The most common implementation of signed integers is Two's complement, where the weight of the most-significant bit is negative

1.22 Integer Casting and Extensions

- Casting between signed and unsigned integers does not change the representation of the data, just the interpretation of the data's value
- When casting from a shorter integer type to a longer type, the old data is extended
  - Zero extension pads unsigned data on the left with extra zeros to preserve the value of the numeral
    - i.e. $0\mathrm{b}\ 0111 \rightarrow 0\mathrm{b}\ 0000\ 0111$
    - i.e. $0\mathrm{b}\ 1111 \rightarrow 0\mathrm{b}\ 0000\ 1111$
  - Sign extension pads signed data on the left using copies of the most significant bit to preserve the sign and value of the numeral
    - i.e. $0\mathrm{b}\ 0111 \rightarrow 0\mathrm{b}\ 0000\ 0111$
    - i.e. $0\mathrm{b}\ 1111 \rightarrow 0\mathrm{b}\ 1111\ 1111$
- When mixing signed and unsigned values in an expression, the signed values are implicitly cast to unsigned

1.23 Integer Arithmetic

Arithmetic on fixed-width binary numbers is performed via modular arithmetic, where all bits of the result to the left of the fixed-width are truncated

- i.e. $0\mathrm{b}\ 1111 + 0\mathrm{b}\ 0001 = 0\mathrm{b}\ \underbrace{0001}_{\text{truncated}}\ 0000 = 0\mathrm{b}\ 0000$

Subtraction on fixed-width binary numbers is performed via addition, where $x - y = x + \sim y + 1$

- Uses the Two's complement negation technique, where $-x = \sim x + 1$

1.24 Arithmetic Overflow

Arithmetic overflow occurs when the result of a calculation does not fit in the current encoding scheme, leading to an incorrect result

- Unsigned overflow occurs when the result lies outside of the interval $[\mathrm{UMin}, \mathrm{UMax}]$
- Signed overflow occurs when the result lies outside of the interval $[\mathrm{TMin}, \mathrm{TMax}]$

1.25 Bit Shifting

The shift operators shift a bit vector in a specified direction by a specified amount n. This will cause n bits of the original bit vector to be truncated, while the vacated bits will be filled by either all zeros or all ones

- A left shift `x « n` adds n zeros to the right of the bit vector
  - The n most-significant bits of the original bit vector are truncated
- A logical right shift `x » n`, where `x` is unsigned, adds n zeros to the left of the bit vector
  - The n least-significant bits of the original bit vector are truncated
- An arithmetic right shift `x » n`, where `x` is signed, adds n copies of the most-significant bit of the original bit vector to the left of the bit vector
  - The n least-significant bits of the original bit vector are truncated

1.26 Scientific Notation

Scientific notation is a representation of a number that shows the multiplication of a mantissa with a power of the base

$$x = \alpha \times \beta^e$$

where α is the mantissa, β is the base, and e is the exponent

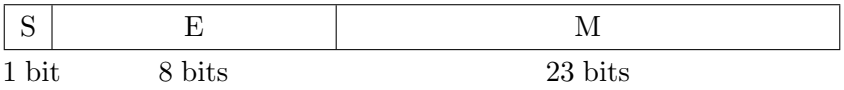
1.27 Floating Point

Floating point representation encodes the sign (`S`), mantissa (`M`), and exponent (`E`) of a number into three separate fields

- Sign field
  - As with signed integers, `0` means positive and `1` means negative
  - Mathematically, $\text{sign} = (-1)^{\text{S}}$
- Exponent field
  - The exponent is an integer value encoded in biased notation
    - The value of the exponent is shifted by the bias of $2^{w-1} - 1$ where w is width of the exponent field, and then converted to unsigned
  - Mathematically, $\text{E} = \text{exponent\_value} + \text{bias}$ and $\text{exponent\_value} = \text{E} - \text{bias}$
- Mantissa field
  - $\text{mantissa} = 1.\text{M}$, where M is an unsigned binary numeral of fixed length with trailing zeros
  - i.e. a mantissa of 1.10111_2 is encoded as $\text{M} = 0\text{b}\ 101\ 1100\ 0000\ 0000\ 0000\ 0000$

There are two levels of precision of normalized scientific binary notation

- `float` data type



- `double` data type

1.28 Floating Point Special Values

value	E	M
± 0	0b0...0	0b0...0
$\pm \infty$	0b1...1	0b0...0
NaN	0b1...1	non-zero
denormalized number	0b0...0	non-zero
normalized number	everything else	any value

A denormalized number uses an implicit leading 0 and a fixed exponent of $1 - \text{bias}$. This enables greater precision for smaller numbers

1.29 Floating Point Arithmetic

- Addition/Subtraction
  1. Shift the binary point of the smaller number so that its exponent matches the larger one
  2. Line up the binary points of the mantissas
  3. Perform the operation, the result will have the larger exponent
  4. Encode the result in floating point, possibly to a special case
- Multiplication
  1. Sum the exponents to get the final exponent
  2. Multiply out the mantissas
  3. Encode the result in floating point, possibly to a special case

1.30 Floating Point Arithmetic Issues

- Arithmetic on ∞ and NaN may compute without raising warnings, leading to unexpected values
- Rounding breaks the associative, distributive and commutative properties of floating point arithmetic
- Equality comparisons may yield unexpected results when two different values are rounded to the same value, or two same values are rounded to different values
- Casting between an integral data type and a floating point data type changes the bit representation

2 x86-64 Instruction Set

2.1 Instruction Sets

An instruction set architecture (ISA) is the combination of parts of the processor design that a programmer needs to understand to write assembly code. It consists of

- The system's state
  - All the information that defines the culmination of past instructions
- The instruction set
  - The list and format of all instructions that the CPU can execute
- The instruction's effect
  - The effect each instruction has on the system state

2.2 Instructions

In x86-64 assembly, instructions are specified using a short instruction name followed by 0-3 operands separated by commas. Most instructions fall into one of the following categories

- Data transfer
- Arithmetic and logical operations
- Control flow

2.3 Size Specifier

In x86-64 assembly, each instruction includes a size specifier letter at the end, corresponding to one of the following

- `b` for 'byte', equivalent to 1 byte
- `w` for 'word', equivalent to 2 bytes
- `l` for 'long word', equivalent to 4 bytes
- `q` for 'quad word', equivalent to 8 bytes

2.4 Operands

In x86-64 assembly, the operands fall into one of the following categories

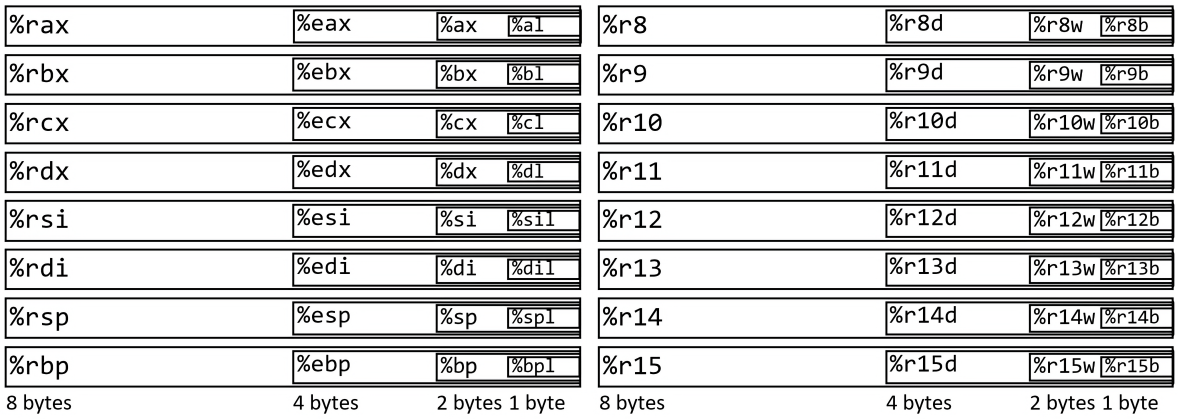
- Immediates
  - Constant integer data, identified by the prefix $
- Registers
  - The name of any of the 16 general-purpose registers, identified by the prefix %
- Memory
  - A specified address that is usually dereferenced, identified by parentheses () enclosing '%'

2.5 Binary Instructions

Binary instructions take two operands and are of the form `instr src, dst`. These operands can be used in any combination except

- The destination operand cannot be an immediate
- The operands cannot both be memory

2.6 Registers



A register is a location in the CPU that stores a 'word size' amount of data. There are a fixed number of registers and they are referred to by name. Smaller divisions of a register always refer to the least-significant bytes of the register

- `Reg[R]` returns the value of the register `R`

2.7 Memory Addressing Modes

Memory operands can be expressed in the form `D(Rb,Ri,S)`

- Displacement `D`
  - Constant displacement value, must be an immediate or constant
  - When omitted, a default value of 0 is used
- Base register `Rb`
  - Name of the register whose value will act as the 'base' of the address calculation
  - When omitted, a default value of 0 is used
- Index register `Ri`
  - Name of the register whose value will be scaled and added to the base
  - When omitted, a default value of 0 is used
- Scale factor `S`
  - Scales the value in `Ri` by the specified number, which can be 1, 2, 4, or 8
  - When omitted, a default value of 1 is used

The computed address is `Reg[Rb]+Reg[Ri]*S+D`

2.8 Address Computation Instructions

The load effective address instruction, denoted `lea`, loads the actual address value instead of dereferencing the address. It is used in the form `lea D(Rb,Ri,S), R`

- The source operand must be a memory operand and its destination operand must be a register operand

2.9 Condition Codes

Condition codes are status bits that are part of the CPU state. They indicate information about the most recently executed assembly instructions

- Carry flag, denoted `CF`
- Zero flag, denoted `ZF`
- Sign flag, denoted `SF`
- Overflow flag, denoted `OF`

2.10 Condition Codes Implicit Setting

Condition code flags can be set implicitly by arithmetic and logical operations. They indicate whether the result had unsigned overflow (`CF`), was zero (`ZF`), was negative (`SF`), or had signed overflow (`OF`)

- While the instruction stores the result in the destination operand, the values of the condition codes are being automatically updated

2.11 Condition Codes Explicit Setting

Condition code flags can be set explicitly by two special instructions. The results of these instructions are never stored; only the values of the condition codes are updated

- Compare, denoted `cmp`
  - Produces a result equivalent to the `sub` instruction
- Test, denoted `test`
  - Produces a result equivalent to the `and` instruction

2.12 Jump and Set Instructions

Jump Instruction	Set Instruction	Condition	Condition Description
`jmp target`	–	`1`	no condition
`je target`	`sete dst`	`ZF`	equal to zero
`jne target`	`setne dst`	`~ZF`	not equal to zero
`js target`	`sets dst`	`SF`	negative
`jns target`	`setns dst`	`~SF`	not negative
`jg target`	`setg dst`	`~(SF^OF)&~ZF`	signed and > 0
`jge target`	`setge dst`	`~(SF^OF)`	signed and ≥ 0
`jl target`	`setl dst`	`(SF^OF)`	signed and < 0
`jle target`	`setle dst`	`(SF^OF)|ZF`	signed and ≤ 0
`ja target`	`seta dst`	`~CF&~ZF`	unsigned and > 0
`jb target`	`setb dst`	`CF`	unsigned and < 0

- The jump instructions will jump our program to the specified target if the condition is met
- The set instructions will set the value of the 1-byte `dst` register to the value of the condition

2.13 Extension Instructions

Extension instructions are similar to a regular `mov` instruction, except that the source operand is shorter than the destination operand

- Zero extension, denoted `movz`
  - Pads data on the left with extra zeros
- Sign extension, denoted `movs`
  - Pads data on the left with copies of the most significant bit

2.14 Conditionals

A conditional is made up of two assembly instructions

- The instruction that sets the condition codes
  - These instructions can set the condition codes implicitly or explicitly
- A conditional jump instruction
  - Conditional jumps are determined by the previous result compared to 0

2.15 Labels

The target to a jump instruction is known as a label. A label is a symbolic representation of an instruction's address and is indicated by an identifier followed by a colon

- i.e. `main: movq %rdi, %rax`

2.16 If-Else Statements

If-Else statements are constructed with labels and jump statements such that at most one branch gets executed

- Jump targets in if-else statements are usually to later addresses in the code

i.e. the following if-statement could be compiled to the following assembly

```
1    if (x < 3) {
2      y += 2;
3    } else {
4      y = 10;
5    }
6    ...
```

```
1    .MAIN:    cmpq  $2, %rax
2              jg    .ELSE
3              addq  $2, %rbx
4              jmp   .ENDIF
5    .ELSE:    movq  $10, %rbx
6    .ENDIF:   ...
```

2.17 Loops

Loops are constructed with labels and jump statements where one of the jump targets goes backwards to the beginning of the loop body

- The differences between repeat-until loops, for loops, while loops, etc are
  - When you evaluate the test conditional, either at the top or bottom of the loop
  - Efficiency, in terms of how many jump instructions you need to evaluate

A repeat-until loop is implemented in assembly as follows

```
1    .LOOPTOP:    <body code>
2                 <CC instr>
3                 j* LOOPTOP
4    .LOOPDONE:   ...
```

- The instruction that sets the condition codes is placed after the body code such that the loop runs at least once

A while loop is implemented in assembly as follows

```
1    .MAIN:      <CC instr>
2                j*' .LOOPDONE
3    .LOOPTOP:   <body code>
4                <CC instr>
5                j* .LOOPTOP
6    .LOOPDONE:  ...
```

- The instruction that sets the condition codes is placed before the body code such that the program may never enter the loop

2.18 Switch Statements

Switch statements are specialized syntax for chains of if-else statements, but they are more efficiently implemented in assembly using jump tables with indirect jump instructions

2.19 Jump Tables

A jump table is a data structure used to help branch to different parts of a program. Since all instructions have addresses associated with them, a jump table is an array of pointers where the pointers are to blocks of code

2.20 Program Counter

The program counter, denoted `%rip`, is a special register that holds the address of the next instruction to execute in your program and gets updated every time an instruction is executed. Normally it advances through consecutive assembly instructions, but this behavior can be altered by jump instructions

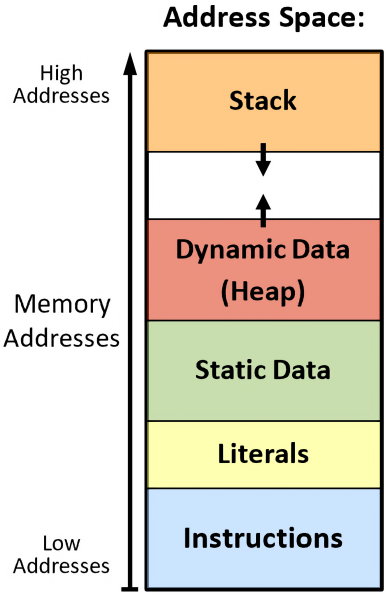
2.21 Indirect Jumps

The indirect jump, denoted `jmp *loc`, is a special instruction that set the program counter to a specified address

- The asterisk differentiates between an indirect jump from a normal jump
  - Tells the program that the address to jump is found in `loc`, instead of given directly as a label
  - The location `loc` is specified as a register or memory operand

3 Stack & Procedures

3.1 Memory Layout



Data:	Management:	Permissions:
Local variables and procedure context	Managed "automatically" (by compiler/assembly)	Writable Not executable
Variables allocated with `new` or `malloc`	Managed "dynamically" (by programmer)	Writable Not executable
Static variables (including globals)	Managed "statically" (initialized at process start)	Writable Not executable
Large literals/constants (*e.g.* "example")	Managed "statically" (initialized at process start)	Read-only Not executable
Program code	Managed "statically" (initialized at process start)	Read-only Executable

3.2 Stack Pointer

The stack pointer, denoted `%rsp`, stores the address representing the top of the stack (bottom of the pictured stack)

- If the stack pointer is manipulated, all addresses less than `Reg[rsp]` are no longer considered part of the stack

3.3 Stack Manipulation

- Stack pointer manipulation
  - The stack pointer can be changed directly via `subq` and `addq` instructions, which allocate and deallocate respectively. These instructions do not affect any of the data in memory, just which data are considered part of the stack
- Stack data manipulation
  - Data can be transferred to and from the stack via `push src` and `pop dst` instructions
  - `push src` decrements `%rsp` and copies the data from the source operand into the newly-allocated space
  - `pop dst` copies the data from `%rsp` into the destination operand and increments `%rsp` to deallocate that space

3.4 Procedure Calling Conventions

Procedure calling conventions are an established set of rules to guarantee that procedures can properly pass data and control one another. The procedure doing the calling is called the caller, and the procedure being called is called the callee

3.5 Return Address

The return address is the address of the caller's next instruction to execute. It indicates how to return to the caller

- Procedure call, denoted `call label`
  - The call instruction passes control to another procedure
  - This will automatically push the return address onto the stack and then update the program counter to the address of the specified label
- Procedure return, denoted `ret`
  - The return instruction is used when a procedure wants to return control
  - This will automatically pop the return address off the stack and then update the program counter to the popped address

3.6 Procedure Data Passing

To pass arguments, the first six arguments must be placed in the registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`. All additional arguments must be placed on the stack in reverse order such that the 7^{th} argument is pushed last

- A useful mnemonic to help memorize the register names is 'Diane's Silk Dress Cost $89'

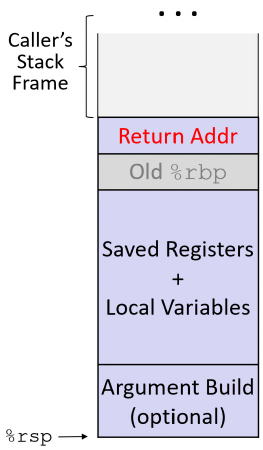
To return values, the return value must be placed in the register `%rax`. For return values larger than a word size, a pointer to the return value is placed in `%rax` instead

3.7 Stack Frames

Stack frames hold the local state of each procedure instantiation. This enables recursion, which requires support for multiple simultaneous instantiations of individual procedures

- Stack frames only exist for a limited time, from when the procedure is called until it returns
- Any space allocated during the execution of the procedure must be deallocated in a parallel manner to ensure that `ret` correctly pops off the return address and completely deallocates the stack frame
- Stack discipline is the process of managing stack frames properly

3.8 Stack Frame Details



- The return address is pushed onto the stack by the `call` instruction and marks the beginning of the stack frame
- Old assembly code uses `%rbp` as the frame pointer
  - The frame pointer indicates the beginning of the current stack frane
  - The old value of `%rbp` would need to be saved if it is being used as a frame pointer, but this is not required in x86-64 assembly
- Old register values that needed to be saved would be pushed, followed by allocated space for local variables that are not being stored solely in registers
- If this procedure calls another procedure, it may need to push more register values and arguments onto the stack
  - Arguments are put on a stack before the `call` instruction is executed such that they appear on the stack above the return address that starts the callee's stack frame

3.9 Register Saving Conventions

%rax	Return value	Caller-saved	%r8	Argument #5	Caller-saved
%rbx		Callee-saved	%r9	Argument #6	Caller-saved
%rcx	Argument #4	Caller-saved	%r10		Caller-saved
%rdx	Argument #3	Caller-saved	%r11		Caller-saved
%rsi	Argument #2	Caller-saved	%r12		Callee-saved
%rdi	Argument #1	Caller-saved	%r13		Callee-saved
%rsp	Stack pointer	Callee-saved	%r14		Callee-saved
%rbp		Callee-saved	%r15		Callee-saved

Register saving conventions describe how register reuse is dealt with in order to avoid accidentally overwriting another procedure's data

- All procedures have access to all of the general-purpose registers during execution
- Registers are designated as either callee-saved or caller-saved
- Register values can be saved by pushing them onto the stack or copying them into a register of the opposite save type

3.10 Callee-Saved Registers

In a callee-saved register, it is the callee's responsibility to save the old value before using/manipulating the register. The callee then restores the old value before returning to the caller

- The saving is the first thing done by the callee, and the restoring is the last thing done before returning control
- From the perspective of the caller, it assumes that the values it has in the callee-saved registers will remain the same when control is passed back to it

3.11 Caller-Saved Registers

In a caller-saved register, it is the caller's responsibility to save the old value before passing control to the callee. The caller then restores the old value after the callee returns control

- The saving is done before calling the callee and the restoring is done after the call
- From the perspective of the callee, it assumes that it has free reign to change the values in the caller-saved registers without worrying about their old values

3.12 Register Saving and Stack Frames

These saved values form the 'Saved Registers' portion of the stack frame

- Callee-saved register values are typically close to the return address, since they must be saved before the procedure can use those registers
- Caller-saved register values typically come after the local variables, since they are only saved when a procedure call is made

3.13 Recursion

As each recursive call is made, a new stack frame is created to hold that instance's local procedure context

- The register saving conventions prevent each instance from corrupting each other's data
- Recursive procedures must be written from the perspective of both a caller and a callee.

3.14 Stack Frame Contents

In an ideal case, all of the procedure's work is being done in the registers such that the minimal stack frame contains just a return address. However, a procedure needs to grow its stack frame in the following situations

- It has too many local variables to save in caller-saved/callee-saved registers
- It has local variables that do not fit in registers, i.e. arrays
- It uses the address-of operator to compute the address of a local variable
  - Registers do not have addresses
- It calls a procedure that takes more than six arguments
- It uses data in caller-saved registers before and after a procedure call
- It modifies/uses callee-saved registers

4 Hardware/Software Interface

4.1 C.A.L.L Process

The process of building and running an executable has four phases

1. Compiling
2. Assembling
3. Linking
4. Loading

Compilation is the process of building an executable from a source file and is comprised of the first three phases. For the purposes of this class, we will be considering C source files

4.2 Compiling

The compiler translates a text file of source code into a text file of assembly

- The first part of compiling is a preprocessor step that handles preprocessor directives (i.e. commands that start with `#`)
- The rest is a complicated process of interpreting the semantic meaning of the source code and translating it into assembly code

Using the `gcc` flag `-S` will stop the compilation process after compiling and output an assembly `.s` file

4.3 Assembling

The assembler converts a text file of assembly code into a binary object file

- Binary object files contain object code, which consists of incomplete machine code and information tables
- The machine code is incomplete since it lacks the addresses associated with the labels of the finalized executable
- The object file contains the bytes for the instructions, static data, and literals found in the source file
  - The addresses of these labels are updated at runtime when memory is allocated to the program
  - In order to update the addresses of these labels, the assembler generates a symbol table and a relocation table
    - The symbol table holds the list of globally-accessible labels (what addresses the program has)
    - The relocation table holds the list of addresses to be updated (what addresses the program needs)

Using the `gcc` flag `-c` will stop the compilation process after assembling and output an object `.o` file

4.4 Linking

The linker combines all the object and static library files needed to produce the final executable.

- Each object file contains its own symbol table, relocation table, data segment, and instructions, so they are combined into a larger, complete data and code segment for the executable
- References are resolved by running through each relocation table and finding the corresponding entry in a symbol table from any of the object files being linked
  - If there are any unresolved symbols due to undefined functions or missing source files, the linking will fail and an error message will be displayed

The `gcc` compiler stops the compilation process after linking and outputs an executable file

4.5 Loading

The loader takes an executable file and starts up a running process from it

- This includes setting up its memory sections and initializing the register values
- The loader is primarily handled by the operating system

4.6 Disassembling

Disassembly is the process of reading low-level binary files and translating them into high-level instructions

- Object files, with their incomplete object code, and executables, with their machine code, can be disassembled

i.e. a disassembled file resembles the following

```
0000000000401126 <main>:
  401126:   48 83 ec 08             sub    $0x8,%rsp
  40112a:   bf 10 20 40 00          mov    $0x402010,%edi
  40112f:   e8 fc fe ff ff          callq  401030 <puts@plt>
  401134:   b8 00 00 00 00          mov    $0x0,%eax
  401139:   48 83 c4 08             add    $0x8,%rsp
  40113d:   c3                      retq
  40113e:   66 90                   xchg   %ax,%ax
```

- The left column shows the addresses of the functions and instructions in hex
- The middle column shows the bytes of the disassembled file
- The right column shows the interpreted assembly instructions from the bytes of the disassembled file
- Note that the last instruction `xchg` is not part of main and does not make sense. This is likely because it is not actually part of an assembly instruction. The disassembler simply interprets the bytes in the binary file as best as it can, including any bytes that might not be part of any function

4.7 Arrays in C

- Declaring an array `T arr[N];` is guaranteed to allocate enough contiguous space to hold the specified `N` elements of size `sizeof(T)`
- Separate array declarations are not guaranteed to be adjacent
- The array subscript operator `arr[i]` uses pointer arithmetic to access the correct memory address

4.8 Arrays in Assembly

- The name of an array is a label/placeholder for the starting address of the array
- Array subscript notation is syntactic shorthand for address referencing
  - `arr[i]` is equivalent to `*(arr + i)` and is equivalent to `Mem[arr + i*sizeof(T)]`
  - `Mem[arr + i*sizeof(T)]` is easily specified in an x86-64 memory operand
    - `D(,Ri,S)` where `D` is the array label, `Reg[Ri]` is the array index, and `S=sizeof(T)`
    - `(Rb,Ri,S)` where `Reg[Rb]` is the array address, `Reg[Ri]` is the array index, and `S=sizeof(T)`

4.9 Multidimensional Arrays

A multidimensional array is a contiguous chunk of memory large enough to hold all of the necessary elements

- Arrays in C use row-major ordering, placing consecutive elements in each row next to each other
- Accessing an element is done by an address calculation followed by a single memory access

4.10 Multilevel Arrays

A multilevel array is created by adding extra levels of arrays of array pointers. Each individual array is guaranteed to use contiguous memory, but the multilevel array as a whole is spread apart in memory

- Accessing an element is done by repeatedly accessing the address of each sub-level array

4.11 Structs in C

A struct is a user-defined, structured group of variables. A struct definition is formatted as

```
struct struct_tag {
    type_1 field_1;
    ...
    type_N field_N;
};
struct struct_tag var_name;
```

- The user-chosen struct tag is part of the name of the new data type we are defining, which is the two-word name `struct struct_tag`
  - The struct tag can be omitted if the struct is not used elsewhere in the code
- Fields can be accessed from an instance using the '`.`' operator, i.e. `strct.x`
- Fields can be accessed from a pointer
  - By dereferencing and using the '`.`' operator, i.e. `(*ptr).x`
  - By using the `->` operator, i.e. `ptr->x`

4.12 Typedef in C

Typedef allows us to create aliases to other data types, such as two-word struct data type names

- Typedef statements are of the form `typedef <data type> <alias>;`

i.e. for structs, a typedef statement can be used after or combined with the struct definition

```
struct point_st {
    int x;
    int y;
};
typedef struct point_st Point;
Point pt1;
```

```
typedef struct {
    int x;
    int y;
} Point;
Point pt1;
```

4.13 Struct Layout

The size and layout of a struct instance in C is automatically determined by the compiler following strict rules set by

- The user-defined ordering of the struct fields
- Alignment requirements

4.14 Internal Fragmentation

The struct layout will follow the ordering of fields in the definition of the struct, but with padding inserted between fields such that each individual field is aligned

- The alignment requirement on each of the fields of the struct is the alignment requirement of its size
- Additional padding is added between fields such that each individual field starts at a multiple of its size
- Later fields will be at higher addresses
- Unused padding between fields is known as internal fragmentation

4.15 External Fragmentation

The overall size of a struct is also subject to alignment requirements to guarantee that the individual fields are properly aligned, and that consecutively allocated structs are also properly aligned

- The alignment requirement on the overall size of the struct is the alignment requirement of its largest field $K_{\max}$
- After placing each field of the struct, additional padding is added to the end of the struct to make the size of the struct a multiple of $K_{\max}$
- Unused padding between struct instances is known as external fragmentation

5 Buffers

5.1 Buffers

A buffer is a region of memory that temporarily stores data

- Buffers are usually arrays

5.2 Buffer Overflow

A buffer overflow occurs when data is written past the end of a buffer onto adjacent memory locations

- This is possible in C since there is no automatic bounds checking

5.3 Stack Smashing

Stack smashing is a form of buffer overflow where data is written past the end of a local array in the stack

- Since array elements are laid out in increasing address order, buffer overflow naturally moves towards higher addresses
- Since the stack grows downwards, stack smashing will eventually overwrite the return address in the current frame and data in previous stack frames

5.4 Code Injection

A code injection occurs when buffer overflow is used to write instructions into the buffer and then modify the return address to execute the injected code

- Designed exploit code is compiled to its binary/machine code representation and put at the beginning of the buffer
- Based on the space available, any necessary padding is added to reach the current frame's stored return address
- The address of the beginning of the buffer is used to replace the return address

When done successfully, the `ret` instruction pops the address of the buffer into `%rip`, instead of the return address, and the program executes the exploit code

- Even though the exploit code is no longer part of the stack, the data remains there until it is overwritten by future stack frames
- Code injection should not alter data in the caller's stack frame to ensure the program continues executing as normal
- Code injection attacks rely on knowledge of the address of the buffer on the stack and the amount of space allocated between the array and the return address

6 Memory & Caches

6.1 IEC Prefixes

The IEC prefixes refer to powers of 1024, unlike the SI system prefixes which refer to powers of 1000

SI Size	Prefix	Symbol	IEC Size	Prefix	Symbol
10^3	kilo	K	2^{10}	kibi	Ki
10^6	mega	M	2^{20}	mebi	Mi
10^9	giga	G	2^{30}	gibi	Gi
10^{12}	tera	T	2^{40}	tebi	Ti
10^{15}	peta	P	2^{50}	pebi	Pi
10^{18}	exa	E	2^{60}	exbi	Ei
10^{21}	zetta	Z	2^{70}	zebi	Zi
10^{24}	yotta	Y	2^{80}	yobi	Yi

6.2 Caches

Caches are memory with short access time used to store frequently or recently used data and instructions

- Caches hold a small subset of the data in memory
- Caches are slower to access than registers, but faster to access than memory

6.3 Cache Mechanics

Data is transferred between caches and memory in blocks

- Blocks are machine-specific fixed unit of transfer between a cache and the storage level below
- Blocks are much larger than a word

6.4 Cache Hit & Miss

When accessing memory, the CPU will always check the caches first

- Cache hit
  - A cache hit occurs when the data the CPU is looking for is found in the cache
  - The data can be returned quickly
- Cache miss
  - A cache miss occurs when the data the CPU is looking for is not found in the cache
  - The CPU must fetch the data from memory and copy it into the cache
  - Depending on the state of the cache, the CPU will invoke the cache's placement or replacement policies to determine where the block will go in the cache
  - After the data is placed in the cache, it is returned to the CPU

6.5 Principle of Locality

The principle of locality states that programs tend to use data at addresses equal to or near those that have been used recently

- Temporal locality states that recently referenced items are likely to be referenced again in the near future
  - In the event of a cache miss, the CPU copies the data into the cache so that future accesses to the data will result in cache hits
- Spatial locality states that items with nearby addresses tend to be referenced close together in time
  - In the event of a cache miss, the CPU copies an entire block's worth of data into the cache so that future accesses to nearby addresses will result in cache hits

6.6 Cache Performance Metrics

- Hit Time
  - How long a cache hit takes. This is the time to check the cache and return data to the CPU
  - Hit Time depends on the cache hardware
- Miss Penalty
  - How long a cache miss takes. This is the time to fetch a block of data from memory
  - Miss Penalty depends on cache and memory hardware
- Hit Rate and Miss Rate
  - The fraction of your program's memory accesses that result in a cache hit or a cache miss
  - Hit Rate $+$ Miss Rate $= 100\%$
  - The Hit Rate and Miss Rate depends on your program

The performance metric used is called Average Memory Access Time (AMAT), defined as

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$$

6.7 Multilevel Caching

Modern systems typically use multiple levels of cache, with each successive level sitting lower/-closer to memory and being larger and slower to access

- Block size may differ between different levels of cache
- A memory access can miss in one level and then hit in the next, causing the block to be copied into a higher level
- AMAT is computed for the overall system caching, taking all levels into account

6.8 Memory Hierarchy

The memory hierarchy is a classification of the main forms of computer storage

- This includes both local data storage (i.e. registers and RAM) and remote/external data storage (i.e. web servers)
- Each level of the hierarchy can be thought of as a cache of the level below it
  - Each level of the hierarchy is a faster way to access a subset of the available data from the level below
- Faster storage technologies cost more per byte, and therefore are used in lower capacities
  - Each level of the hierarchy has a smaller storage capacity than the level below it

When programs exhibit good locality, the CPU has access to the storage capacity of the lower levels at an average access time closer to those of the higher levels

6.9 Block Size

The block size, denoted as K, refers to the number of bytes in each block

- The block size is always a power of 2

6.10 Cache Size

The cache size, denoted as C, refers to the number of bytes that the cache can store

- The cache size is always a multiple of block size
- The cache size may be given in units of bytes or in units of blocks

6.11 Block Index

The address space of memory is pre-determined such that blocks are made up of adjacent bytes and do not overlap with each other

- Addresses 0 to $K-1$ forms one block, addresses K to $2K-1$ forms another block, and so on
- The block index of an address A is given by $\lfloor \frac{A}{K} \rfloor$

6.12 Block Offset

Each block has K bytes in it, which correspond to K ordered positions within the block

- The block offset of an address A is given by $A\%K$

6.13 Cache Placement

Cache placement refers to the strategy of placing blocks on a cache miss. This affects where we look for blocks on a cache access, and affects which blocks are displaced from the cache

- Cache placement strategies must be fast in order to reduce AMAT

6.14 Direct-Mapped Cache Placement

Direct-mapped cache placement uses a hash function to determine where in the cache to place a specific block. This gives a very fast algorithm that utilizes every spot in the cache

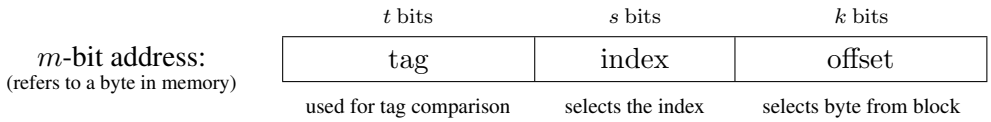
- $h(\text{block number}) = \text{block number } \% \ (C/K)$

6.15 Direct-Mapped Cache Replacement

Direct-mapped cache placement is deterministic such that the cache access always hashes to the same spot. This means that no special replacement policy is needed – we always kick out the existing block in that spot

6.16 Cache Address

A cache access is handled by analyzing the requested address via its TIO address breakdown



The widths of the address fields are computed as

- $k = \log_2(K)$, the number of bits needed to represent each byte of a block
- $s = \log_2(C/K)$, the number of bits needed to represent each block index in the cache
- $t = m - s - k$, the number of remaining available bits that uniquely identify this data

When handling a cache access, the values of the fields are used as follows

1. Check the cache index indicated by the index field
2. Compare the stored tag against the value of the tag field. If they match, then its a cache hit. Otherwise, its a cache miss
3. Once the proper block has been found/fetched, the value of the offset field tells us where within the block the requested data starts from

6.17 Associativity

In a set associative cache, each block is allowed to fit into a specified set of locations, called lines

- An E-way set associative cache uses sets of size E
  - A block can be placed in E different lines
- A direct-mapped cache is a 1-way associative cache
- A fully-associative cache is a C/K-way set associate cache
- The number of sets in an E-way set associative cache is $S = (C/K)/E$

Associativity fixes the problem of having to alternate between different blocks that map to the same index. Without associativity, the blocks would keep evicting each other and we lose the benefits of temporal locality

6.18 Associative Cache Access

The TIO address breakdown still indicates which set the block should be found in. However, the associativity changes where the TIO address breakdown splits the tag and index fields

- $s = \log_2(S)$, the number of bits needed to represent each set index in the cache
- $t = m - s - k$, the number of remaining available bits that uniquely identify this data

6.19 Associative Cache Replacement

On a cache miss, there are multiple locations within the designated set where the block can be placed

- If any of the lines in the set are invalid/empty, then we can place our block there
- If the set is full, then the least recently used (LRU) block is replaced in order to maximize temporal locality

6.20 Cache State

Caches are hardware such that there is always data present, either program data or mystery data. In order to tell the difference, a valid bit is included

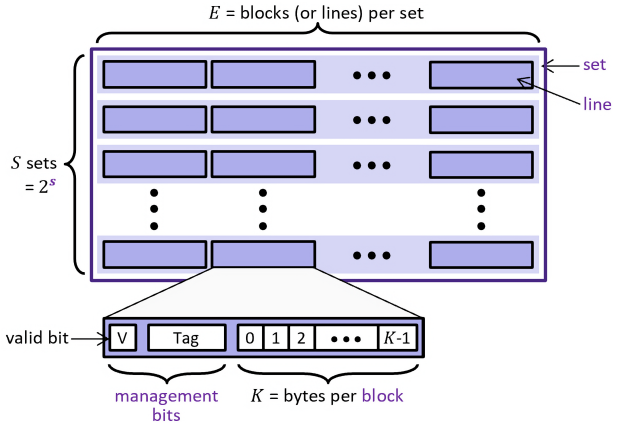
- The management bits consist of the valid bit and the tag bits
- The cache line consists of the management bits and the rest of the TIO address
- A cache with no valid program data is called a cold cache

6.21 Cache Line

A cache line consists of a block and its corresponding management bits

6.22 Cache Organization

The geometry of a cache can be completely described by the number of sets S, the associativity E, and the block size K. The cache size C is the product of these terms where $C = S \times E \times K$



6.23 Cache Misses

Understanding the cause of misses can help identify inefficiencies in code. Cache misses are generally categorized as follows

- Compulsory misses, if we access a block for the first time, then we are guaranteed to have a cache miss
- Conflict misses, if too many references map to the same set, then this causes misses when we revisit earlier references
- Capacity misses, if the total amount of data we are using exceeds the size of the cache, then the cache does not have the capacity to keep all of the blocks at once and the evicted blocks cause misses

Different cache parameters have different effects on each type of miss

- Larger block size reduces compulsory misses since more data is brought into the cache with each miss
- Higher associativity reduces conflict misses by allowing more blocks to coexist in the same set
- Larger cache size or reducing our working set of data helps reduce capacity misses

6.24 Cache Writes

When writing to memory, the CPU will always check the caches first

- Write hit
  - A write hit occurs when the data is already in the cache
  - A write through cache writes the change into the block and in the level below
  - A write back writes the change into the block in the cache only, but makes a note that this block has been updated (it holds more recent data than the level below)
    - This requires an additional management bit called the dirty bit to be stored for each line in the cache
    - The updated data is only written to the level below when a dirty line is evicted from the cache
- Write miss
  - A write miss occurs when the data is not in the cache
  - A write allocate cache will load the requested block into the cache before executing a write hit
  - A no-write allocate cache will skip the cache and send the write directly to the level below

A cache must have a write hit and a write miss policy

- A write back + write allocate cache combination tries to avoid going to the lower level as much as possible
- A write through + no-write allocate cache combination is primarily concerned with making sure that multiple copies of data at different levels remain consistent with each other

6.25 Cache Friendly Code

- Our goal for spatial locality is to use the smallest memory strides possible
- Our goal with temporal locality is to maximize our use of whatever data has been loaded into the cache before it gets evicted
- With cache blocking, operations on a large data structure are grouped such that the chunks fit in the cache

7 System Control Flow & Processes

7.1 Exceptional Control Flow

Exceptional control flow is a mechanism by which the CPU reacts to changes in system state. This allows the CPU to transfer control between processes and the operating system, as well as allow it to react to external signals like input and output devices

7.2 Exceptions

An exception is the transfer of execution/control to a part of the operating system called the kernel in response to some event. The exception will invoke some kernel code called an event handler that will deal with the event. There are three possible outcomes upon the completion of the event handler

1. Re-execute the instruction that caused the initial event
   - The event has been handled, but the instruction did not complete as intended
2. Execute the next instruction
   - The event has been handled, and the instruction has been completed as intended
3. Abort the process
   - The event could not be handled so the process needed to exit

7.3 Asynchronous Exceptions

Asynchronous exceptions are caused by events external to the processor. These are also known as interrupts. Since they have nothing to do with the process' instructions, interrupt handlers always return control to the next instruction

7.4 Synchronous Exceptions

Synchronous exceptions are caused by executing an instruction and can have a variety of intentions and outcomes

- Traps
  - Intentional
  - Caused by instructions that transfer control to the operating system
  - Operating system performs some desired function with a privileged resource, i.e. disk, network
  - When these succeed, the trap handlers return control to the next instruction
- Faults
  - Unintentional, but possibly recoverable
  - Caused by unexpected outcomes from regular instructions, i.e. bad arithmetic, bad memory accesses
  - If the fault is recoverable, the fault handler will return control to the current instruction (retry)
  - If the fault is not recoverable, the fault handler will abort
- Aborts
  - Unintentional and unrecoverable
  - Caused by catastrophic events where the only option is to abort, i.e. a hardware failure

7.5 Processes

A process is an instance of a running program/executable. The operating system provides processes with two abstractions to maintain an interface between the process and underlying hardware

- Logical control flow
  - Each process believes that it has exclusive use of the CPU for executing its instructions
  - This is accomplished via context switching
- Private address space
  - Each process believes that it has exclusive use of memory
  - This is accomplished via virtual memory

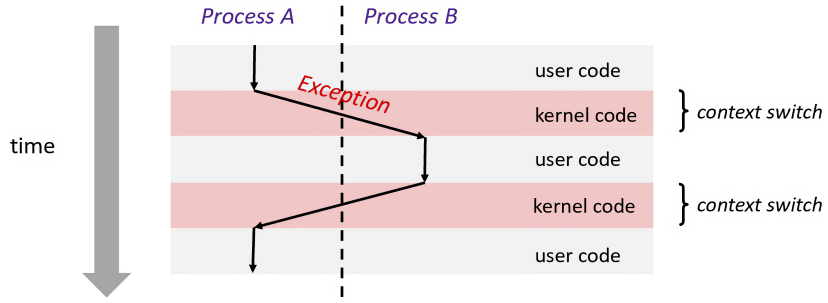
7.6 Concurrency

Two processes are said to run concurrently if their instruction executions overlap in time, i.e. the periods between when they start and stop have some overlap

- This can be accomplished with a single CPU by switching back and forth between the instructions of multiple processes

7.7 Context Switching

Switching between processes is accomplished via context switching, which is managed by the operating system and exceptional control flow



1. Save current registers in memory
2. Schedule next process for execution
3. Load saved registers and switch address

7.8 Fork-Exec Model

The fork-exec model is a process management model used by Linux. Instead of spawning new processes from scratch, the fork-exec model splits it into fork and exec

- `fork()` is a system call (a type of trap) that duplicates the process that called it
  - This spawns a child process that starts in the same execution state as the parent process
  - `fork()` returns the child PID in the parent process, and return `0` in the child process
- `exec*()` is a family of functions that replace the current process data with a fresh instance of the specified program
  - The entire family is `execv`, `execl`, `execve`, `execle`, `execvp`, `execlp`
  - The actual system call is `execve`. The others are wrappers for it

When a new process is started in Linux, what actually happens is that a currently running process duplicates itself by calling `fork()` and then the child process calls some form of `exec*()`

- In order to distinguish processes, each process is assigned a unique process ID (PID)

7.9 `exec*()`

The `exec*()` family of functions replaces the current process data with a fresh instance of the specified program. This is the loading portion of the C.A.L.L process

- Data: static data and literals are copied from the program executable
- Code: code is copied from the program executable
- Heap: the heap is empty since no dynamically-allocated memory have been requested
- Stack: another program sets up the `main` stack frame and handles the command-line arguments and environment variables passed into `exec*()`
- Registers: `main` often takes two inputs stored in `%rdi` and `%rsi`. `%rsp` has to be set to the top of the stack, and `%rip` has to be set to the beginning of `main`

The setup of the stack and registers are taken care of by library functions, i.e. on Linux there is `__libc_start_main`

7.10 Process Termination

There are three ways that a process terminates

- Executing the `return` statement from `main`
- Explicitly calling the library function `exit()`
- Via an abort from an exception handler

`return` and `exit()` provide a status code by value and by argument respectively

7.11 Process Reaping

The system resources for a terminated process persists until the process is reaped, which is when the status code is read and the process and its resources are deleted

- A terminated process that is not reaped is called a zombie process
- The parent process is responsible for reaping a child process
  - Implicit reaping: the child process is reaped when the parent process terminates
  - Explicit reaping: the parent process invokes the `wait()` or `waitpid()` system calls
    - These system calls suspend execution of the parent process until a child process terminates

7.12 `init`

If a parent process terminates before its child process, then the child process is said to be orphaned

- A background process called `init` is responsible for reaping an orphaned child process
- `init` always has a PID of 1
- `init` is called `systemd` on more recent Linux systems

8 Virtual Memory

8.1 Virtual Memory

Virtual memory is the process abstraction of a private address space which hides the actual amount of RAM installed on the machine from processes

- Virtual address space: the set of bytes that each process thinks it has available from the physical address space
  - A byte in the virtual address space is identified by an n-bit virtual address
  - The virtual address space is $N = 2^n$ bytes
- Physical address space: the set of bytes actually available in physical memory that must be shared across all running processes
  - A byte in the physical address space is identified by an m-bit physical address
  - The physical address space is $M = 2^m$ bytes
- Swap space: a region of disk that can temporarily hold excess memory
  - This is needed since the total amount of virtual address space used by all running processes can exceed the amount of physical address space
- Indirection: a mapping from the processes' virtual address to the physical address in memory where the data is actually stored
  - Program instructions such as `movq (%rdi), %rax` reference virtual addresses

8.2 Pages

A page is a fixed-length contiguous block of memory that represents the smallest unit of data for memory management

- Physical page numbers (PPNs) uniquely identify a page in physical memory
- Virtual page numbers (VPNs) uniquely identify a page in virtual memory
- The page size is $P = 2^p$, where p is the number of bits needed to represent every byte in a page
  - The page size is the same in physical and virtual memory
- Memory is split into non-overlapping pages, each with a unique page number

8.3 Page Tables

A page table is a lookup table that converts a requested virtual address for a process to its corresponding physical address

- The VPN is used as an index to find the page table entry (PTE) that contains the VPN's corresponding PPN
- Page tables must have an entry for every possible VPN in the virtual address space

8.4 Page Table Entries

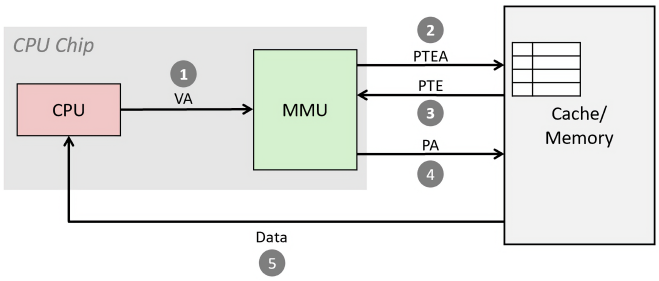
A page table entry consists of a PPN and its corresponding management bits

- Valid bit
  - Indicates whether this translation exists
  - i.e. whether this virtual page is actually stored in physical memory
- Dirty bit
  - Indicates whether the virtual page has been updated
  - Only relevant if a copy of the page exists in the swap space as well
- Access rights bits

8.5 Memory Protection

- Between processes
  - Virtual memory enables both protection from different processes meddling with each other's memory, and sharing of memory when desired
  - Since every process has the same set of virtual page numbers, a page table must be maintained for each process
  - Memory protection between processes is possible by ensuring that VPNs from two different processes do not map onto the same PPN
  - Memory sharing between processes is possible by mapping VPNs from different processes to the same PPN
- Within processes
  - Virtual memory protects against a process misusing its own address space via access rights
  - Access rights are extra management bits stored in each page table entry and mimic the Linux file access permissions
    - Read bit (R), whether this process is allowed to read the data on this page
    - Write bit (W), whether this process is allowed to change the data on this page
    - Execute bit (X), whether this process is allowed to use data on this page when fetching instructions from `%rip`

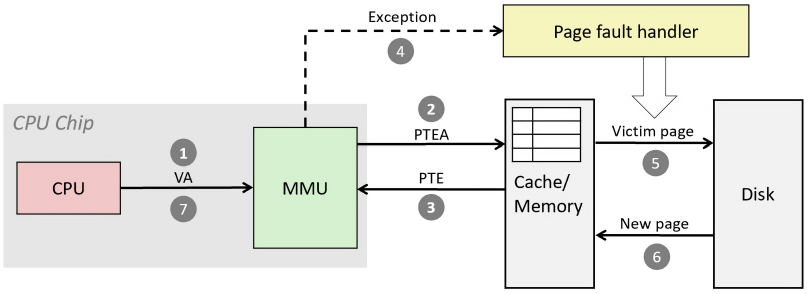
8.6 Page Hit



1. The CPU sends a virtual address to the memory management unit (MMU)
2. The MMU uses the virtual address' VPN and the value in the page table base register (PTBR) to look up the corresponding page table entry (PTE) in the page table, which is stored in memory
3. The requested PTE is returned to the MMU. The valid bit must be 1 for it to be a page hit
4. The MMU then translates the virtual address to a physical address and uses the physical address to request the desired data from memory
5. The cache/memory returns the requested data to the CPU

During a page hit, the MMU requests data from cache/memory twice: once for the PTE, and again for the requested data

8.7 Page Fault



1. The CPU sends a virtual address to the memory management unit (MMU)
2. The MMU uses the virtual address' VPN and the value in the page table base register (PTBR) to look up the corresponding page table entry (PTE) in the page table, which is stored in memory
3. The requested PTE is returned to the MMU. The valid bit must be 0 for it to be a page fault
4. The MMU triggers a page fault exception
5. The page fault handler identifies a victim page from physical memory. If this victim page is dirty, the page fault handler pages it out to disk
6. The handler pages in the requested page and updates the corresponding page table entry (PTE) in the page table
7. The handler returns control to the original process, which restarts the faulting instruction for a guaranteed page hit

During a page fault, the MMU requests data from cache/memory thrice: once for the PTE, and twice when the instruction is restarted

8.8 Translation Lookaside Buffer

The translation lookaside buffer (TLB) is a small hardware cache in the memory management unit which reduces the number of memory accesses needed to the page table during address translation

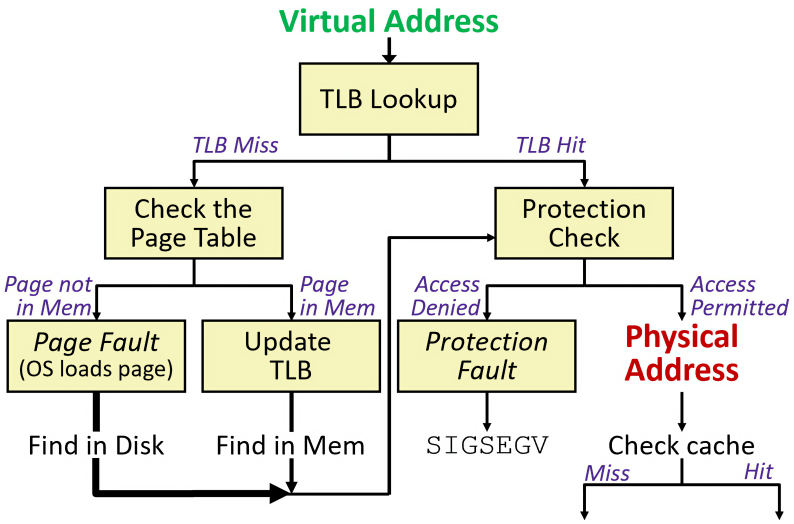
- The TLB caches page table entries, which consist of the PPNs and their corresponding management bits
- Like a normal cache, the TLB has a defined size, associativity, and placement and replacement policies
- In order to access the TLB, the VPN field is split into a TLB index (TLBI) and a TLB tag (TLBT)
  - $s = \log_2(S)$, the number of bits needed to represent each set in the TLB cache
  - $t = m - s$, the number of remaining available bits that uniquely identify this page entry
- A TLB entry consists of a valid bit, a TLB tag, and a page table entry

8.9 Translation Lookaside Buffer Mechanics

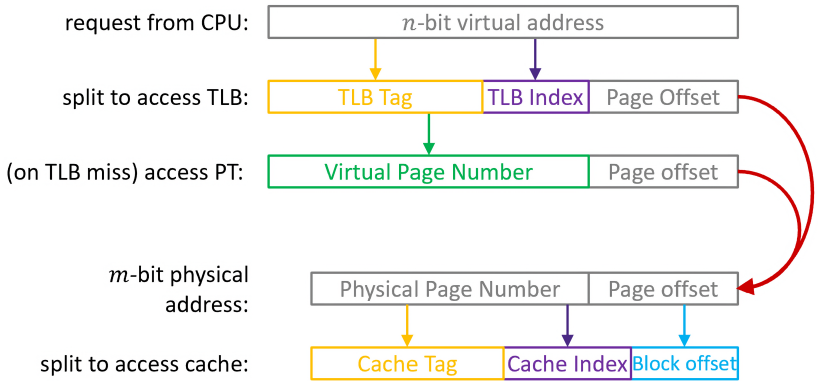
The MMU sends the requested VPN to the TLB first before accessing the page table

1. Check if the requested virtual address is in the TLB
   - TLB Hit: entry found in TLB, so return PPN
   - TLB Miss: entry not found in TLB, so fetch from the page table in memory
     - Page Hit: page is in physical memory, so load the page table entry into the TLB
     - Page Fault: page is not in the physical memory, so trigger a page fault exception, which should load the page into memory, update the entry in the page table, and load it into the TLB
2. Check the cache using the translated physical address
   - Cache Hit: block found in cache, so return data to the CPU
   - Cache Miss: block not found in cache, so fetch block from memory and return it to the CPU

8.10 Address Translation Flowchart



8.11 Address Manipulation Flowchart



9 Memory Allocation

9.1 Memory Allocation

- Dynamic memory allocation
  - Dynamically allocated data are intended for memory whose characteristics, such as size or lifetime, can only be determined at runtime
  - Dynamically allocated data is stored in the heap
  - i.e. input strings
- Automatic memory allocation
  - Automatically allocated data are intended for memory which have a known lifetime tied to their stack frames
  - Automatically allocated data is stored in the stack
  - i.e. local variables
- Static memory allocation
  - Statically allocated data are intended for memory which are fixed in size and persist for the entire lifetime of the process
  - Statically allocated data is stored in the static data
  - i.e. global variables

9.2 Dynamic Memory Allocators

A dynamic memory allocator manages the size of the heap, indicated by a `brk` pointer, and tracks the current allocations. The heap is grouped into a collection of variable-sized heap blocks which are marked as either allocated or freed

- Implicit allocators
  - The programmer is only responsible for allocations and the allocator implicitly handles the deallocations
  - i.e. garbage collection in Java
- Explicit allocators
  - The programmer is responsible for both allocations and deallocations
  - i.e. manual memory management in C

9.3 Memory Allocator Interface

Applications and dynamic memory allocators must adhere to the following interface

- Applications
  - Can issue arbitrary sequences of allocation and deallocation requests
  - Must never access memory that is not currently allocated
  - Must never try to deallocate memory that is not currently allocated
- Allocators
  - Cannot control the number or size of allocated blocks
  - Must respond immediately to allocation requests
  - Must satisfy alignment requirements during allocation
  - Can only allocate from freed memory
  - Cannot move allocated blocks

9.4 Performance Goals

Dynamic memory allocators have two performance goals that frequently conflict with each other

1. Throughput, the allocator wants to complete as many allocation and deallocation requests as possible per unit time
2. Memory utilization, the allocator wants to use the heap space as efficiently as possible

9.5 Allocator Implementations

- An implicit free list traverses its blocks via arithmetic by using the size of each block
- An explicit free list traverses its free blocks using a linked list data structure

9.6 Dynamic Memory Allocation in C

`void* malloc(size_t size)` will request a continuous block of at least `size` bytes of uninitialized memory from the allocator

- `malloc` returns a pointer to the beginning of the allocated space, or `NULL` in case of an allocation failure
- Often `size` is calculated by use of the `sizeof()` macro
  - i.e. `int* ptr = (int*) malloc(n * sizeof(int))` allocates the space for an array of n `int`'s
- `calloc` is an allocation function that initializes the allocated heap block to all zeros
- `realloc` changes the size (up or down) of a previously allocated block

9.7 Dynamic Memory Deallocation in C

`void free(void* p)` will deallocate the entirety of the heap block pointed to by p

- p must hold the same address that was returned by the allocating function, p cannot point to an arbitrary byte within the heap block
- If p is `NULL`, then nothing happens
- Invalid calls to `free` will result in a system exception
  - i.e. address of the middle of a block, address of a previously deallocated block

9.8 Heap Fragmentation

Poor memory utilization in the heap is caused by fragmentation, which consists of all the parts of the heap that are not currently being used to store program data

9.9 Internal Fragmentation

Internal fragmentation is the wasted space inside of allocated heap blocks

- The allocator stores metadata, called the header, within each allocated and free block
- The allocator may add padding for alignment purposes
- This means that the allocator must always allocate a block size that is greater than the requested space
- The requested space is called the payload, which has size `malloc(size)`
- All of the space within a block that is not the payload is considered internal fragmentation

9.10 External Fragmentation

External fragmentation is the unused space between allocated heap blocks

- External fragmentation does not include free space on the heap past the end of the last allocated block
- External fragmentation is caused by the specific pattern of allocation and deallocation requests
- External fragmentation can cause situations where an allocation request cannot be specified despite there being enough aggregate free heap memory

9.11 Dynamic Memory Allocation Strategies

- First fit, search the list from the beginning and return the first free block that is large enough
- Next fit, search the list starting from where the last search finished and return the first free block that is large enough
- Best fit, search through the whole list and return the free block that is large enough to fulfill the request but causes minimal external fragmentation

9.12 Allocating a Free Block

The process of fulfilling an allocation request is as follows

1. Compute the necessary block size based on the payload, metadata, and padding for alignment
2. Search for a suitable free block using the allocator's allocation strategy
   - If found, continue
   - If not found, return `NULL`
3. Compare the necessary block size against the size of the chosen block
   - If equal, continue
   - If not, split off the excess and convert it into a new free block
4. Allocate the block and return the address of the beginning of the payload

9.13 Memory Alignment

The allocator has a set alignment that affects heap blocks in two ways

- The address of the beginning of the payload must be a multiple of the alignment size
- The total size of the block will be padded out to be a multiple of the alignment size

9.14 Minimum Block Size

A block maintained by our dynamic memory allocator must include any necessary metadata and satisfy alignment requirements, which leads to the notion of a minimum block size. The minimum block size affects the computation of the necessary block size in step 1, and splitting in step 3

- Even if the request is very small, the computed necessary block size will be padded out to the minimum block size
- If the remaining free space after splitting is less than the minimum block size, then it will be included as padding in the allocated block

9.15 Freeing an Allocated Block

The process of fulfilling a deallocation request is as follows

1. Flip the is-allocated flag for the specified block
2. Coalesce neighboring free blocks into a larger single free block
    - If the preceding and following blocks are allocated, continue
- If the preceding and following blocks are free, bi-directional coalesce
    - If the preceding block is allocated and the following block is free, forwards coalesce
    - If the preceding block is free and the following block is allocated, backwards coalesce

9.16 Boundary Tags

The header and footer metadata are known as boundary tags

- The header stores the size and is-allocated flag at the front of the heap block
- The footer stores the size and is-allocated flag at the end of the heap block

9.17 Explicit Free List

An explicit free list traverses its free blocks using a doubly-linked list data structure

- Only free blocks have `next` and `prev` pointers
- Saves time by checking only free blocks

9.18 Garbage Collection

Garbage collection is a form of automatic memory management that is often used by implicit dynamic memory allocators to reclaim heap-allocated memory. To identify memory that is no longer needed, memory is viewed as a directed graph where

- Each allocated heap block is a node in the graph
- Each pointer is an edge in the graph
- Locations not in the heap that contain pointers into the heap are called root nodes
  - i.e. registers, stack, static data

Any heap node that is reachable from a root node is considered still in use by the process. Any heap node that is not reachable from root nodes is considered no longer in use and eligible to be freed. The following assumptions are made

- The memory allocator knows which data are pointers and which are not
- All pointers to heap nodes point to the start of the payload

9.19 Mark and Sweep

A particular implementation of garbage collection is mark and sweep. This requires the use of a mark bit which is stored in the boundary tags. When the garbage collector is invoked

1. The garbage collector recursively follows all root nodes into the heap and set the mark bit for all reachable heap blocks
2. The garbage collector sweeps through the heap from start to finish, and checks the mark bit for each block
   - If the mark bit is set, then clear it and move on
   - If the mark bit is not set, then free the block

9.20 Memory-Related Issues in C

Memory-related issues in C may cause compiler warnings, runtime errors, or security flaws

- Bad/incorrect pointer arithmetic
- Dereferencing a non-pointer
- Improper or lack of bounds checking on input or array indices
- Failing to free memory, leading to a memory leak
- Using a deallocated variable
- Using uninitialized memory
- Trying to free a freed block
- Trying to access a freed block
- Using the wrong allocation size