College of Engineering & Applied Sciences

CSPB 2400

Computer Systems

Exam Notes

Taylor Larrechea

2024

Exam 1 Notes

Compilation

The following are definitions of common compilation processes.

- **Preprocessing (CPP)**: The preprocessor handles directives for source code file inclusion, macro definitions, and conditional compilation.
- **Compilation (CC)**: The compiler takes the preprocessed source code and converts it into assembly code.
- **Assembly (AS)**: The assembler then takes this assembly code and translates it into machine code, producing object files.
- **Linking (LD)**: Finally, the linker combines these object files into a single executable program.

Memory Hierarchy

At the top, we have registers, which are the fastest type of memory within a computer. Below registers are levels of cache memory (L1, L2, and L3), each slower than the last but still faster than RAM (Random Access Memory). Further down are disks, like HDDs or SSDs, which provide more storage but at slower access speeds. Lastly, remote storage, which can be cloud storage or network-attached storage, offers the most space but has the slowest access speed due to its physical and network distance from the CPU.

Memory Hierarchy

`Registers` → `L1 Cache` → `L2 Cache` → `L3 Cache` → `RAM` → `Disks` → `Remote Storage`

Abstraction

In the context of computer systems, an abstraction is a simplification where complex details are hidden to reduce complexity and increase efficiency. It allows users and programs to interact with systems and devices at a higher level without concern for the underlying implementation details. This concept is central to computer science because it enables the development of complex systems and applications by breaking them down into more manageable parts. Each layer of abstraction provides a set of interfaces for the level above, ensuring that changes in one layer do not necessarily affect others.

- Files abstract the details of I/O devices, allowing users and programs to interact with data storage without needing to understand the specifics of the hardware.
- Virtual memory abstracts the physical memory, giving an application the impression of having a contiguous and large amount of memory while physically it could be fragmented and less than the virtual space.
- Processes abstract the execution of multiple tasks, giving the impression that there is more than one processor executing different tasks simultaneously when, in fact, the CPU switches between tasks to give the illusion of concurrency.

Propositional Logic

Propositional logic is a branch of logic that deals with propositions, which are statements that can be either true or false. It involves logical operations such as:

- **AND (Conjunction)**: A logical operator that results in true if both operands are true.
- **OR (Disjunction)**: A logical operator that results in true if at least one of the operands is true.
- **XOR (Exclusive OR)**: A logical operator that results in true only if one operand is true and the other is false.

A	B	A & B	A | B	A ^ B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bitwise Operations

Bitwise operators perform operations on binary representations of numbers:

- **& (AND)**: Sets each bit to 1 if both bits are 1.
- **| (OR)**: Sets each bit to 1 if one of the bits is 1.
- **^ (XOR)**: Sets each bit to 1 if only one of the two bits is 1.
- **~ (NOT)**: Inverts all the bits.
- **! (Logical NOT)**: Inverts the truth value (used with Boolean values, not bitwise).
- **» (Right Shift)**: Shifts the bits of a number to the right by a specified number of positions.
- **« (Left Shift)**: Shifts the bits of a number to the left by a specified number of positions.

Decimal To Binary

To convert a decimal number to binary, divide the number by 2 and record the remainder. Repeat this process with the quotient until the quotient is 0. The binary representation is the sequence of remainders read in reverse (from the last remainder obtained to the first).

Decimal To Hexadecimal

To convert a decimal number to hexadecimal:

1. Divide the decimal number by 16.
2. Record the remainder.
3. Continue dividing the quotient by 16 until you get a quotient of zero.
4. The hexadecimal number is the sequence of remainders read in reverse (from the last remainder to the first).
5. Each remainder corresponds to a hexadecimal digit: 0-9 for remainders 0-9 and A-F for remainders 10-15.

Binary To Decimal

To convert binary to decimal, each bit in the binary number is multiplied by the base (2) raised to the power of its position. Starting from the rightmost bit (least significant bit), the position starts at 0 and increases by 1 as you move left. Summing these products gives the decimal equivalent.

Binary To Decimal Formula

Mathematically, if $b_n b_{n-1} \ldots b_2 b_1 b_0$ is a binary number, its decimal equivalent D is:

$$D = \sum_{i=0}^{n} b_i \cdot 2^i$$

Here, b_i represents each binary digit (0 or 1), and n is the position of the digit from the right.

Binary To Hexadecimal

To convert binary to hexadecimal:

1. Group the binary number into sets of four digits (bits), starting from the right. If the leftmost group has less than four bits, add zeros to make a group of four.
2. Convert each 4-bit group to its hexadecimal equivalent, using the fact that each group represents a number from 0 to 15.
3. The hexadecimal number is the sequence of these hexadecimal digits read from left to right.

Hexadecimal To Decimal

To convert hexadecimal to decimal:

1. Assign each digit of the hexadecimal number a positional value, starting from 0 on the right.
2. Convert each hexadecimal digit to its decimal equivalent (0-9 stay the same, and A-F correspond to 10-15).
3. Multiply each digit by 16 raised to the power of its positional value.
4. Sum these values.

Hexadecimal To Decimal Formula

Mathematically, if $h_n h_{n-1} \ldots h_2 h_1 h_0$ is a hexadecimal number, its decimal equivalent D is

$$D = \sum_{i=0} h_i \cdot 16^i$$

Here, h_i represents each hexadecimal digit, and n is the position of the digit from the right.

Hexadecimal To Binary

To convert hexadecimal to binary:

1. Convert each hexadecimal digit to its 4-bit binary equivalent.
2. Each digit in hexadecimal corresponds to four binary digits, as hexadecimal is base-16 and binary is base-2.
3. Concatenate these binary groups to get the final binary number.

Binary, Decimal, Hexadecimal Conversions

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Left Shift

In this operation, the binary digits of a number are moved a certain number of places to the left. For every shift left, a zero is added to the rightmost end, and the leftmost bit is discarded. This operation effectively multiplies the original number by 2 for each shift position (since binary is base 2). For instance, shifting `1011` left by one position results in `0110`, which doubles the original number. The 3-bit left shift multiplies the number by 2^3 or 8.

Logical Right Shift

This operation shifts all the bits to the right by the specified number of positions. For non-negative numbers, logical and arithmetic right shifts yield the same result. It fills in the leftmost bits with zeros. This operation effectively divides a number by 2 raised to the number of shifts.

Arithmetic Right Shift

Used with signed numbers, it shifts values to the right but fills in the new leftmost bit with the sign bit (the original leftmost bit) instead of zeros. This preserves the sign of the number in two's complement form, which is used to represent negative numbers. This operation effectively divides a number by 2 raised to the number of shifts.

C Word Type Declarations

In C, different data types have different sizes when declared. The following is a table of typical data types and the number of bytes that the specific data type takes up in memory.

Data Type	32 Bit Architecture	64 Bit Architecture
`char`	1	1
`short int`	2	2
`int`	4	4
`long int`	4	8
`long long int`	8	8
`char*`	4	8
`float`	4	4
`double`	8	8

Unsigned Integer Representation

In unsigned integers (integers that can only be positive) the number conversion from binary to decimal is in the traditional sense.

Unsigned Integer Examples

Below are some examples of unsigned integers:

$$\texttt{0b00001101} = 13$$
$$\texttt{0b01110010} = 114$$
$$\texttt{0b00101011} = 43$$
$$\texttt{0b10101110} = 174$$

Signed Integer Representation

In signed integers (integers that can be negative) the most significant bit (MSB) is referred to as the 'sign' bit. 0 for positive and 1 for negative. The process of converting a binary number to a decimal is the same, but the MSB is calculated in the number as $-1 \cdot 2^n$ where n is the MSB. The rest of the digits are calculated in the same manner as an unsigned (positive) value.

Signed Integer Examples

Below are some examples of signed integers:

$$\texttt{0b10101001} = -87$$
$$\texttt{0b10101110} = -82$$
$$\texttt{0b10110011} = -77$$
$$\texttt{0b11011101} = -35$$
$$\texttt{0b11111100} = -4$$
$$\texttt{0b00001101} = 13$$
$$\texttt{0b00101011} = 43$$
$$\texttt{0b01110010} = 114$$

Binary Addition

The formula for binary addition is:

1. Start from the least significant bit (rightmost side).

2. Add the bits in each column.
3. If the sum in a column is 2 (10 in binary), write 0 and carry over 1 to the next left column.
4. If the sum is 3 (11 in binary), write 1 and carry over 1.
5. Proceed to the next column to the left, adding the carried over value.

Binary Subtraction

Binary subtraction is similar to decimal subtraction except it follows the rules of base 2. Here's a formulaic approach:

1. Start from the least significant bit (rightmost side).
2. Subtract the second number's bit from the first number's bit in each column.
   - $0 - 0 = 0$
   - $0 - 1 = 1$ (With a borrow of 1)
   - $1 - 0 = 1$
   - $1 - 1 = 0$
3. If the bit from the first number is less than the bit from the second number, borrow 1 from the next column to the left (which is equivalent to adding 2 in binary).
4. Continue the process for each column until the subtraction is complete.

Unsigned Integer Overflow

Because there are minimum and maximums for numbers represented in binary for a given number of bits, overflow can occur. The range of values for a binary number b will range from

$$0 \rightarrow 2^n - 1$$

where n is the number of digits in the binary representation. For example, an 8 bit unsigned integer can range from 0 to 255.

Signed Integer Overflow

Overflow applies the same for signed integers as it does for unsigned integers, but just slightly different. Because signed integers can represent negative values, the range of values for a binary number b will range from

$$-1 \cdot 2^{n-1} \rightarrow 2^n - 1$$

where n again is the number of digits in the binary representation. For example, an 8 bit signed integer can range from -128 to 127.

Floating Point Representation

For a floating point value, there are two main parts:

- **Integer Values**: These bits are in front of the decimal point - `XX.YYY ...`
- **Fractional Values**: These bits are after the decimal point = `...XX.YYY ...`

Fractional Binary To Decimal

To convert from a binary fractional number we add up the integer values like regular binary to decimal conversions. For the fractional bits, we add up the fractional bits with negative exponents indexing from -1 to n. For n fractional bits b_f, the fractional representation F is calculated with

$$F = \sum_{f=1}^{n} b_f \cdot 2^{-f}.$$

Take for example the following example.

Fractional Binary Example

Take for example the fractional binary number `11.0001` is going to be

$$\text{Integer Values} = 1 \cdot 2^1 + 1 \cdot 2^0 = 2 + 1 = 3$$

$$\text{Fractional Values} = 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} = \frac{1}{16} = 0.0625.$$

The final decimal value D is then

$$D = 3.0625_{10}.$$

IEEE Floating Point Representation

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation established by the Institute of Electrical and Electronics Engineers (IEEE). The standard defines formats for representing floating-point numbers (including negative zero and special "Not a Number" (NaN) values) and establishes guidelines for floating-point arithmetic in computer systems.

The key components of IEEE floating point representation are:

- **Sign Bit**: A single bit is used to denote the sign of the number. A 0 represents a positive number, and a 1 represents a negative number.
- **Exponent**: A set of bits that follow the sign bit, used to represent the exponent for the number. The exponent is stored in a biased form, meaning that a fixed value (the bias) is subtracted from the actual exponent to get the stored exponent. The bias is $2^{k-1} - 1$, where k is the number of bits in the exponent field. The exponent in IEEE 754 is used to represent both very large and very small numbers.
- **Mantissa (Significand)**: This is the fraction part of the floating-point number, representing the significant digits of the number. The binary point is assumed to be just to the right of an implicit leading bit (which is typically a 1, except for denormal numbers). The mantissa is normalized, meaning that it is scaled to be just less than 1 (for normalized numbers), which is represented by the implicit leading bit.
- **Special Values**: IEEE floating-point format can represent special values such as infinity (both positive and negative) and NaN (Not a Number), which are used to denote results of certain operations that do not yield a numerical value.

Denormalized And Normalized Values

Normalized and denormalized values play a crucial role in the IEEE floating-point representation. In the context of IEEE 754 standard, normalized values are represented with an implicit leading bit equal to 1, allowing for a higher precision and a wider range of representable numbers. On the other hand, denormalized values, also known as subnormal numbers, lack the implicit leading bit and are used to represent numbers close to zero, which fall below the normal range of the floating-point format. These denormalized numbers enable a graceful underflow, ensuring that very small numbers can still be represented with reduced precision.

Denormalized numbers can be summed up with the following:

- Denormalized numbers are used to represent values that are too small to be normalized (those that are closer to zero than what can be represented by a normalized value). These numbers do not have an implicit leading 1.
- In denormalized form, the exponent is all zeros, and the mantissa is allowed to begin with a series of zeros. This allows for representation of numbers closer to zero than is possible with normalized form, albeit with less precision.
- For example, a binary number `0.00101` (in denormalized form) cannot assume an implicit leading 1 and must store all bits explicitly.

Normalized numbers can be summed up with the following:

- In normalized representation, the floating-point number is scaled such that the leading digit of the mantissa is always a 1 (except for 0). Because this leading digit is always 1, it doesn't need to be stored, which effectively gives one more bit of precision. This is known as an "implicit leading bit."

- For a binary floating-point system, this means the mantissa (or significand) is always in the range of [1,2) (including 1 but excluding 2).
- For example, a binary number `1.101` (in normalized form) is represented with an implicit 1, so only `.101` needs to be stored.
- The exponent is adjusted accordingly to represent the correct scale of the number and is not all zeros or all ones.

IEEE To Decimal

When converting from IEEE floating point binary to decimal, there are specific values that we need to calculate to in order to convert the number to decimal representation. Here is the recipe for doing so:

- **Bias**: $b = 2^{k-1} - 1$ where k is the number of bits in the exponent.
- **Exponent**: $e =$ value of exponent in decimal.
- **Exponent With Bias**: $\mathbf{E = e - b}$ (Normalized) i.e. $e \neq 0$, $\mathbf{E = 1 - b}$ (Denormalized) i.e. $e = 0$.
- **Mantissa**: $\mathbf{M = 1 + f}$ (Normalized) i.e. $e \neq 0$, $\mathbf{M = f}$ (Denormalized) i.e. $e = 0$.
- **Sign**: s: 1 if sign bit is 0, -1 if sign bit is 1.

Compiling this to calculate the decimal representation D we have

$$D = s \cdot 2^E \cdot M.$$

IEEE To Decimal Example

Consider the IEEE floating point binary number `0b101001100000` that has 1 sign bit, 5 exponent bits, and 6 fractional bits.

$$\begin{aligned}
\textbf{Sign: } s: &\text{ Sign bit is a 1 } \therefore\ s = -1 \\
\textbf{Bias: } b: &\ b_n = 5 \ \therefore\ b = 2^{5-1} - 1 = 2^4 - 1 = 16 - 1 = 15 \\
\textbf{Exponent: } e: &\ e = 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 9 \\
\textbf{Biased Exponent: } E: &\ e \neq 0 \ \therefore\ \text{Norm.} \ \therefore\ E = e - b = 9 - 15 = -6 \\
\textbf{Fraction: } f: &\ e \neq 0 \ \therefore\ \text{Norm.} \ \therefore\ f = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + \cdots + 0 \cdot 2^{-6} = 1/2 \\
\textbf{Mantissa: } M: &\ e \neq 0 \ \therefore\ \text{Norm.} \ \therefore\ M = 1 + 1/2 = 3/2 = \texttt{0b1.1} \\
\textbf{Decimal Value: } &D = s \cdot 2^E \cdot M = -1 \cdot 2^{-6} \cdot 3/2 = -1 \cdot \frac{1}{64} \cdot \frac{3}{2} = -\frac{3}{128} = -0.0234375.
\end{aligned}$$

Therefore the decimal value is then

$$101001100000_2 = -\frac{3}{128} = -0.0234375_{10}.$$

Decimal To IEEE

We can convert decimal values to IEEE as well. Below is a recipe for doing so.

- Determine the sign bit (s) based on the sign of the decimal value. If the value is positive, the sign bit is 0; if the value is negative, the sign bit is 1.
- Convert the absolute value of the decimal number into binary form.
- Calculate the bias (b) for the number of exponents.
- Calculate the maximum number of shifts (α) when writing the binary number in scientific notation.
  - $\alpha = 1 - b$
- Normalize the absolute value of the binary number in the form $1.xxx \cdot 2^y$

- If $y > \alpha$
  - $E = y$, value is normalized.
- If $y < \alpha$
  - Normalize the absolute value of the binary number in the form $0.xxx \cdot 2^{\alpha}$. $E = \alpha$, value is denormalized.
- Calculate the exponent of the binary number and represent it in binary.
  - If the value is **normalized**:
    - $e = E + b$
  - If the value is **denormalized**:
    - $e = 0$
- Grab the mantissa (the digits after the decimal place in normalized binary number), these are the fraction bits (f).
- Stitch the results together: `s + e + f`

Decimal To IEEE Example

Consider the decimal number $d = -0.046875$, approximate this value to floating point IEEE with 4 exponent bits and 5 fractional bits.

Sign: This number is negative therefore $s = 1$

Bias: $b = 2^{k-1} - 1 = 2^{4-1} - 1 = 7$ (k is the number of exponent bits)

Alpha: $\alpha = 1 - b = 1 - 7 = -6$ (Max number of shifts is -6)

Absolute Value: $|d| = 0.046875 = 3/64$

Abs. Value In Binary: $b_d = 1/32 + 1/64 = 1 \cdot 2^{-5} + 1 \cdot 2^{-6} =$ `0b0.000011`

Normalized Binary: $\hat{b}_d = 1.1 \cdot 10^{-5}$ ($-5 > \alpha \therefore E = -5$)

Normalized Exponent: $e = E + b = -5 + 7 = 2$ ($e > 0$ therefore Norm.)

Normalized Exponent In Binary: $e =$ `0010`

Mantissa: $M =$ `1000`, Values to right of normalized binary value

The finally binary representation is then

$$-0.046857_{10} = \texttt{0b1001010000}_2.$$

Rounding

Rounding a value to IEEE is performed the same as rounding with decimals. Round the decimal value to the corresponding decimal value, and then convert to binary representation.

Exam 2 Notes

Registers

Registers are small, fast storage locations directly inside the CPU that are used to hold data temporarily during the execution of programs. In the x86 architecture, which has evolved over the years from 16-bit to 32-bit (x86) and then 64-bit (x86-64 or AMD64) versions, registers have specific roles and sizes that influence their usage in various operations.

- **General-Purpose Registers (GPRs)**: Initially, x86 CPUs had 8-bit and 16-bit GPRs, but this expanded to 32-bit in the 386 and later processors, and 64-bit in x86-64 processors. These registers include:
  - `AX, BX, CX, DX` (**Accumulator, Base, Count, Data**): Used for arithmetic, data storage, loop counters, and more. In 32-bit mode, they are extended to `EAX, EBX, ECX,` and `EDX,` and further to `RAX, RBX, RCX,` and `RDX` in 64-bit mode.
  - `SI, DI, BP, SP` (**Source Index, Destination Index, Base Pointer, Stack Pointer**): Used for string operations, stack management, and memory addressing. Extended to `ESI, EDI, EBP, ESP` in 32-bit, and `RSI, RDI, RBP, RSP` in 64-bit.
  - **8 new general-purpose registers** (`R8` to `R15`) are available in x86-64, providing additional flexibility.
- **Segment Registers**: Used in real mode and protected mode for memory segmentation, helping with memory management by dividing memory into smaller segments.
  - `CS, DS, ES, FS, GS, SS` (**Code, Data, Extra, more segment registers**): They are primarily used to hold the segments' base addresses used by the CPU to access memory.
- **Instruction Pointer (IP)**: The IP (or `EIP` in 32-bit, `RIP` in 64-bit) register points to the next instruction to be executed. It's automatically updated by the CPU.
- **Flag Registers**: The `FLAGS` register (`EFLAGS` in 32-bit, `RFLAGS` in 64-bit) contains flags that indicate the status of the processor and the outcome of various operations, such as the Zero flag, Carry flag, etc.
- **Control Registers**: Used in protected mode to control operations such as memory management, task switching, and more. `CR0, CR2, CR3,` and `CR4` are examples.
- **MMX, XMM, and YMM Registers**: Used for SIMD (Single Instruction, Multiple Data) operations to perform parallel processing on multiple data points. These are beyond the basic x86 registers and are used for advanced multimedia and arithmetic operations.

Assembly Instructions

In the context of assembly language, there are common operations that can be used for registers and memory addresses. Below are some common operations found in assembly.

Instruction	Description	Syntax
`ADD`	Performs addition operation	`ADD destination, source`
`CALL`	Calls a procedure	`CALL procedure_label`
`CMP`	Compares two values	`CMP operand1, operand2`
`DEC`	Decrements the value of a register/memory	`DEC destination`
`INC`	Increments the value of a register/memory	`INC destination`
`JE`	Jump if equal (zero flag set)	`JE label`
`JNE`	Jump if not equal (zero flag clear)	`JNE label`
`JMP`	Unconditional jump to a label	`JMP label`
`LEA`	Computes address of memory operand and stores in register	`LEA register, memory_reference`
`MOV`	Transfers data from one location to another	`MOV destination, source`
`POP`	Pops a value from the stack	`POP destination`
`PUSH`	Pushes a value onto the stack	`PUSH source`
`RET`	Returns from a procedure	`RET`
`SUB`	Performs subtraction operation	`SUB destination, source`
`TEST`	Tests bits by performing a bitwise AND	`TEST operand1, operand2`

Signed Data

When dealing with signed data, the CPU uses the sign flag (`SF`), overflow flag (`OF`), and zero flag (`ZF`) to determine the outcome of a comparison. Here are some of the conditional jump instructions tailored for signed comparisons:

- `JG` (**Jump if Greater**): Jumps if the result of a subtraction is positive, and no overflow occurs (`SF=OF` and `ZF=0`).
- `JL` (**Jump if Less**): Jumps if the result is negative considering signed operands (`SF` $\neq$ `OF`).
- `JE` (**Jump if Equal**): Jumps if the operands are equal (`ZF=1`), applicable to both signed and unsigned comparisons.
- `JGE` (**Jump if Greater or Equal**): Jumps if a signed number is greater than or equal to another (`SF=OF`).
- `JLE` (**Jump if Less or Equal**): Jumps if a signed number is less than or equal or if the result is zero (`ZF=1` or `SF` $\neq$ `OF`).

Unsigned Data

For unsigned data comparisons, the CPU relies on the carry flag (`CF`) and zero flag (`ZF`) to make jump decisions:

- `JA` (**Jump if Above**): Jumps if the first operand is greater than the second operand in an unsigned comparison (`CF=0` and `ZF=0`).
- `JB` (**Jump if Below**): Jumps if the first unsigned operand is less than the second (`CF=1`).
- `JAE` (**Jump if Above or Equal**): Jumps if the first operand is greater than or equal to the second operand in an unsigned comparison (`CF=0`).
- `JBE` (**Jump if Below or Equal**): Jumps if the first operand is less than or equal to the second operand in an unsigned comparison (`CF=1` or `ZF=1`).

Assembly Instruction Example

Below is an example of some assembly instructions and their outcomes. Assume the register `%rax` holds the value 10, and `%rcx` holds the value 4.

```
  leal (%rax, %rax, 2), %rdx :  %rdx = %rax + 2(%rax) = 3(%rax) = 3(10) = 30
   leal 4 (%rcx, %rax), %rdx :  %rdx = 4 + %rcx + %rax = 4 + 4 + 10 = 18
      leal (, %rcx, 4), %rdx :  %rdx = 0 + 4(%rcx) = 4(4) = 16
leal 4 (%rax, %rcx, 8), %rdx :  %rdx = 4 + %rax + 8(%rcx) = 4 + 10 + 8(4) = 14 + 32 = 46
```

Please note, the parenthesis in the RHS of the computations above are being used as multiplication symbols.

Memory Addressing

Memory addressing is the scheme used by a CPU to locate and access data in the computer's memory. Each byte in memory has a unique address, much like houses on a street. Instructions in assembly language use these addresses to specify where data should be read from or written to.

Operands

In the context of assembly language, an operand can be considered as an argument to an instruction that specifies what data is to be operated on. Operands can be immediate values (directly provided in the instruction), register values (which hold a small amount of data within the CPU for quick access), or memory addresses (which point to locations in RAM).

Register Values And Arithmetic Computations

Registers are used for a variety of purposes in assembly language, including but not limited to holding operands for arithmetic computations. Here's how register values are typically involved in arithmetic computations with memory addresses:

- **Register as Direct Operand**: A register can hold one of the operands for an arithmetic operation. For example, `ADD EAX, EBX` adds the contents of `EBX` to `EAX` and stores the result in `EAX`.
- **Memory Addressing Modes**: When combined with arithmetic operations, several addressing modes can be used to refer to memory addresses:
  - **Immediate Addressing**: Using a literal number. For instance, `ADD EAX, 5` adds 5 to the contents of `EAX`.
  - **Direct Addressing**: Using a direct memory address. For example, `ADD EAX, (0x0040)` adds the value at memory address `0x0040` to `EAX`.
  - **Indirect Addressing**: Using a register to hold the memory address. For example, `ADD EAX, (EBX)` adds to `EAX` the value at the memory address contained in `EBX`.
  - **Based Addressing with Displacement**: Using a register plus an offset. For example, `ADD EAX, (EBX + 8)` adds to `EAX` the value at the memory address `EBX` plus 8 bytes.
- **Computation Results**: After an arithmetic operation is performed, the result can be stored back in a register or in a memory location, depending on the instruction used.

CPUs use memory addressing to:

- Retrieve instructions to be executed.
- Access data operands for instruction execution.
- Store the results of computations.

Memory Addressing Example

Below are some simple examples of memory addressing and arithmetic operations with registers and memory addressing:

```
addl 16 (%ebp), %ecx :  Reg[ecx] = Reg[ecx] + Mem[Reg[ebp]] + 16
  addq $0x11, (%rax) :  Mem[Reg[rax]] = 17 + Mem[Reg[rax]]
  subl $0x11, (%eax) :  Mem[Reg[eax]] = Mem[Reg[eax]] - 17
```

When registers are enclosed by parenthesis, this means we are accessing that register in memory. And thus, the operations must deal with the value in memory and not just the value of the register.

Exam 3 Notes

Amdahl's Law

Amdahl's Law is a principle that helps in understanding the potential speedup in the overall performance of a system when only part of the system is improved. It's particularly relevant in the context of parallel computing and optimizing performance through hardware upgrades or software optimization.

The formula for Amdahl's law is

$$S = \frac{1}{(1-\alpha) + \alpha/k}$$

where in the aforementioned formula the variables are

- S: The maximum possible speedup of the entire process.
- α: The proportion of the process that benefits from the improved system performance.
- k: The ratio of how much a process can be sped up.

Sector Access

Accessing a sector in a disk involves reading from or writing to a specific, small, fixed-size portion of the disk. Disks, whether they are hard disk drives (HDDs) or solid-state drives (SSDs), are organized into platters (for HDDs) or blocks (for SSDs), which are further divided into tracks and sectors. Let's focus primarily on HDDs for the classic understanding of disk sectors, though the general concept applies to SSDs with some differences due to their lack of moving parts.

Sector Definition

A sector is the smallest storage unit on a disk that can be read from or written to. Historically, sectors on hard drives have been 512 bytes in size, but newer hard drives use a larger sector size of 4096 bytes (or 4K), known as Advanced Format (AF). Each sector has its own unique address, which the disk controller uses to read or write data.

Accessing A Sector

To access a sector, the disk's read/write head must be positioned over the correct track (for HDDs, this is a circular path on the surface of a platter) and then wait for the disk to rotate until the desired sector is under the head. This process involves two main components:

1. **Seek Time**: The time it takes for the read/write head to move to the correct track. Seek time can vary significantly, depending on how far the head needs to move.
2. **Rotational Latency**: Once the head is over the correct track, it must wait for the disk to rotate the correct sector under the head. The average rotational latency depends on the rotation speed of the disk, measured in revolutions per minute (RPM).

The formula for calculating the time it takes to access a sector is made up of three parts: average seek time, average rotation, and average transfer time. Formally as an expression this is

$$T_\alpha = T_\sigma + T_\rho + T_\tau = T_\sigma + \frac{1}{2} \cdot \left(\frac{60}{\text{Rot. Rate}}\right) \cdot 1000 + \frac{60}{\text{Rot. Rate}} \cdot \left(\frac{1}{\text{Sectors / Track}}\right) \cdot 1000$$

where σ is the average seek time, ρ is the average rotation, and τ is the average sectors per track.

Pipeline Speed Up

Pipeline speed up is a concept in computer architecture that refers to the increase in processing speed that can be achieved by using an instruction pipeline. The basics of pipelining are

- **Instruction Pipeline**: It's like an assembly line in a factory. Each stage of the pipeline completes part of the instruction. While one stage is processing one part of an instruction, another stage can process a different part of another instruction.
- **Stages**: Common stages in a simple instruction pipeline include instruction fetch, instruction decode, execute, memory access, and write-back.

Combinatorial Logic

Combinatorial logic functions are a foundational concept in digital circuit design. They are logic circuits whose outputs depend only on the current state of their inputs and not on any prior history (in contrast to sequential logic circuits, which have outputs that depend on a combination of current input states and historical input states).

Solving Combinatorial Logic Problems

When solving a combinatorial logic problem (in the context of what was seen in the quizzes) the goal is to maximize the clock speed. When a register is added between stages, it allows the stages to operate independently in a pipelined fashion. To maximize the clock speed, we want to minimize the delay of the longest pipeline in the process. The steps for determining the highest possible clock speed is:

1. Add a register between the first two stages, calculate the delay for the first delay plus the delay of the newly added register, calculate the delay for the rest of the process, and keep track of the largest delay in this scenario.
2. Add a register between the last two stages, calculate the delay for first two delays plus the delay of the newly added register, calculate the delay for the rest of the process, and keep track of the largest delay in this scenario.
3. Determine the smallest delay between the two largest delays found in step 1 and 2, and calculate the clock speed with the smallest delay with the following formula

$$\text{C.S} = \frac{1}{\alpha \cdot 10^{-12}}$$

where α is the smallest of the two largest delays in ps (pico seconds $10^{-12}(s)$). The resulting calculation is in Hz (hertz $(1/s)$).

Memory Aliasing

Memory aliasing refers to a situation in computer systems where two or more different memory addresses refer to the same physical memory location. This can occur in various contexts and can have both intentional and unintentional consequences. The common causes for memory aliasing are

- **Pointers In Programming**: In languages like C or C++, if two or more pointers point to the same memory address, changing the memory value through one pointer affects the value seen by all pointers aliasing that address.
- **CPU Caches**: Multiple cache lines may map to the same memory location, especially in systems with virtually indexed, physically tagged caches.
- **Memory-Mapped I/O**: Devices mapped to the same address space can cause aliasing, where different device registers are accessed using the same memory addresses.
- **Virtual Memory Systems**: Different virtual addresses may map to the same physical address through the page table mechanism, either within the same process or across different processes.
- **Compiler Optimizations**: When the compiler tries to optimize code, it assumes that different variables occupy different memory locations. Aliasing can break these assumptions and lead to incorrect optimizations.

Memory aliasing has several implications when it happens, here are some examples of these implications

- **Correctness**: It can lead to bugs that are difficult to track down because the same memory is being manipulated from multiple reference points.
- **Performance**: Aliasing can hinder certain optimizations because the compiler must assume that operations affecting one alias could affect all aliases.
- **Consistency**: In multi-threaded environments, memory aliasing complicates the coherence protocols that ensure memory consistency across different CPU cores and caches.

Program Optimizations

We can optimize code in numerous ways, cutting down on executions, function calls, etc.

Machine Independent Optimization

Machine independent optimizations are code transformations that improve performance and are not specific to any particular machine architecture. These optimizations generally improve the efficiency of the code by reducing the number of instructions, improving algorithmic complexity, or enhancing data access patterns. They are typically performed by the compiler at a high level, without considering the specifics of the underlying hardware.

```
// Without
int compute_area(int width, int height) {
    int area1 = width * height;
    int area2 = width * height; // Redundant computation
    return area1 + area2;
}
// With
int compute_area(int width, int height) {
    int area = width * height; // Compute once
    return area + area; // Reuse the result
}

```

Loop Unrolling

Loop unrolling is an optimization technique that aims to increase a program's execution speed by reducing or eliminating the overhead of loop control. By executing more than one iteration of the loop per cycle through the loop control code, it can also improve the opportunities for other optimizations, such as instruction pipelining in the processor.

```
// Without
for (int i = 0; i < N; ++i) {
    dest[i] = src[i] + 1;
}
// With
// Handle the main part of the loop in steps of 4 to reduce loop overhead.
for (int i = 0; i < N; i += 4) {
    dest[i] = src[i] + 1;
    dest[i+1] = src[i+1] + 1;
    dest[i+2] = src[i+2] + 1;
    dest[i+3] = src[i+3] + 1;
}

// Handle the remainder of the elements that didn't fit into groups of 4
for (int i = N - N % 4; i < N; ++i) {
    dest[i] = src[i] + 1;
}

```

Unrolling And Multiple Accumulators

Loop unrolling with multiple accumulators is a further enhancement of the loop unrolling optimization. In this technique, not only is the loop unrolled to reduce the loop overhead, but multiple accumulator variables are also used to hold intermediate results. This can reduce dependencies between loop iterations and allow for more parallelism, especially on hardware that can perform multiple operations simultaneously.

```
// Without
int sum = 0;
for (int i = 0; i < N; ++i) {
    sum += array[i];
}
// With
int sum0 = 0, sum1 = 0, sum2 = 0, sum3 = 0;
for (int i = 0; i < N; i += 4) {
    sum0 += array[i];
    sum1 += array[i + 1];
    sum2 += array[i + 2];
    sum3 += array[i + 3];
}
int totalSum = sum0 + sum1 + sum2 + sum3;

// Handle the remainder of the elements that didn't fit into groups of 4
for (int i = N - N % 4; i < N; ++i) {
    totalSum += array[i];
}

```

Strength Reduction

Strength reduction is an optimization technique where more expensive operations are replaced with equivalent but less costly operations. It's particularly effective in loops where an expensive operation, like multiplication or division, can be replaced with addition or subtraction.

```
// Without
for (int i = 0; i < N; ++i) {
    array[i * 4] = i * 4; // Multiplication inside loop
}
// With
for (int i = 0, j = 0; i < N; ++i, j += 4) {
    array[j] = j; // Replace multiplication with addition
}

```

Parallel Accumulators

Parallel accumulators are an optimization technique used to minimize dependencies between successive iterations of a loop that would otherwise limit the degree of parallelism achievable. By using separate accumulator variables in a loop that performs a reduction (like summing values), a program can take advantage of parallel execution capabilities of modern processors.

```
// Without
int sum = 0;
for (int i = 0; i < N; ++i) {
    sum += array[i];
}
// With
int sum1 = 0, sum2 = 0;
for (int i = 0; i < N / 2; ++i) {
    sum1 += array[2 * i];
    sum2 += array[2 * i + 1];
}
int totalSum = sum1 + sum2;

// Handle the case where N is odd
if (N % 2 != 0) {
    totalSum += array[N - 1];
}

```

Common Subexpression Elimination

Common subexpression elimination (CSE) is an optimization technique that identifies instances of identical expressions being evaluated multiple times, and eliminates the redundancy by computing the expression once and reusing the result. This reduces the computation time and can also decrease code size, improving cache performance.

```
// Without
int width = 5;
int height = 10;
int area = width * height; // Computed here
int perimeter = 2 * (width + height);
int doubleArea = 2 * (width * height); // Computed again - a common subexpression
// With
int width = 5;
int height = 10;
int area = width * height; // Compute once
int perimeter = 2 * (width + height);
int doubleArea = 2 * area; // Reuse the computed area

```

Inlining

Inlining is an optimization where the compiler replaces a function call with the actual code of the function. It eliminates the overhead associated with function calls such as parameter passing, return value computation, and stack frame management. This is especially beneficial for small, frequently called functions.

```
// Without
int square(int num) {
    return num * num;
}

```

```
int main() {
    int total = 0;
    for (int i = 0; i < N; ++i) {
        total += square(i);
    }
    return total;
}
// With
int main() {
    int total = 0;
    for (int i = 0; i < N; ++i) {
        int temp = i * i; // Inlining 'square(i)'
        total += temp;
    }
    return total;
}

```

Spatial Locality

Spatial locality is a principle that helps to optimize how computer systems access and store data, and it's particularly relevant in the context of memory hierarchies, including caches. The concept of spatial locality refers to the tendency of a processor to access data locations that are physically close to recently accessed locations. The formal definition of spatial locality is

- **Spatial Locality**: The principle that if a particular storage location is accessed, the locations with nearby addresses are likely to be accessed soon. This is due to the structure of most programs, where data is often organized sequentially in memory (like arrays or adjacent fields in a structure).

Cache

A cache in computing is a smaller, faster storage layer that stores copies of data from a more substantial, slower storage layer. The primary goal of a cache is to increase data retrieval performance by reducing the time it takes to access data. Caches are ubiquitous in computer systems, found in web browsers, operating systems, and most importantly, within the CPU to speed up access to memory.

Caches consist of fundamental parameters that are used to determine quantities like the Cache Offset, Cache Index, and Cache Tag.

The following are fundamental parameters regarding caches:

$S = 2^s$	(Number of Sets)
$s = \log_2(S)$	(Number of *Set Index* Bits)
E	(Number of Lines Per Set)
$B = 2^b$	(Block Size (Bytes))
$b = \log_2(b)$	(Number of *Block Offset* Bits)
$M = 2^m$	(Maximum Number of Unique Memory Addresses)
$m = \log_2(m)$	(Number of Physical (Main Memory) Address Bits)
$t = m - (s + b)$	(Number of *Tag* Bits)
$C = B \cdot E \cdot S$	(Cache Size (Bytes))

Cache Offset

The cache offset, also known as the block offset, refers to the position of a byte or word within a cache block or line. When data is loaded into the cache, it's not loaded in single bytes but rather in blocks of contiguous bytes. The offset specifies the exact byte within this block where the desired data is located. It's used to pinpoint the data within the cache line once the correct cache line is identified.

Cache Index

The cache index helps in determining which cache line (or slot) within the cache is to be used for storing and retrieving a specific block of data. The cache is divided into several lines, and the index specifies which line data is stored in or should be looked for. The index is derived from the memory address being accessed, usually by taking certain bits from the middle of the address.

Cache Tag

The cache tag stores information about the data's identity in the cache line. It's used to verify that the data in the cache line is the same as the data being requested by the CPU. When a memory address is accessed, a part of it is used to form the cache tag. The cache controller compares this tag with the tags in the cache to determine if the requested data is present (a hit) or not (a miss).

Cache Hit Or Miss

- **Cache Hit**: A cache hit occurs when the data requested by the CPU is found in the cache. This means the CPU can directly read from the cache without having to access the slower main memory, leading to faster data retrieval.
- **Cache Miss**: A cache miss occurs when the requested data is not found in the cache. This forces the CPU to fetch the data from the main memory, which is a slower process. A cache miss also triggers the process of loading the requested data into the cache for future access, possibly replacing existing data based on the cache's replacement policy.

Exam 4 Notes

Linking

Linking is a fundamental process in software development that combines various pieces of compiled code into a single executable program. In UNIX-like systems, linking can be either static or dynamic:

- **Dynamic Linking**: Libraries are linked at runtime, which allows multiple programs to share the code of a single library file on disk, reducing overall memory usage.
- **Static Linking**: The linker combines all the necessary library routines and modules into a single executable file at compile time.

Symbol Resolution In Linking

During the linking process, the linker must resolve references to symbols (variables, functions, etc.) that are defined in one module but used in another. This resolution can become complex when multiple definitions of the same symbol are found across different modules. Here, the concepts of strong and weak symbols are crucial.

Strong Symbols

- **Definition**: Strong symbols are typically defined as functions and initialized global variables. The linker uses strong symbols to resolve ambiguities when the same symbol appears multiple times.
- **Characteristics**:
  - Have higher precedence during symbol resolution.
  - If there are multiple strong symbols of the same name across different modules, it leads to a linking error, as each symbol is assumed to represent different entities.

Weak Symbols

- **Definition**: Weak symbols are generally used for uninitialized global variables and provide a way for the linker to resolve symbols when multiple definitions occur without causing errors.
- **Characteristics**:
  - If a weak symbol and a strong symbol of the same name exist, the strong symbol is preferred, and no error is raised.
  - If multiple weak symbols of the same name exist, any of them might be chosen by the linker, and the choice does not result in a linking error.

Rules For Symbol Resolution

Given these definitions, UNIX linkers typically follow a set of rules to handle duplicate symbols:

- **Any Weak If Only Weak Present**: If multiple weak symbols with the same name exist and no strong symbol is present, the linker arbitrarily chooses one. This flexibility is useful for linking against large libraries where weak symbols can act as placeholders or defaults.
- **No Multiple Strong Symbols**: If more than one definition of a strong symbol is found, it results in a linker error. This ensures that each strong symbol is unique.
- **Strong Over Weak**: If both strong and weak symbols with the same name exist, the strong symbol takes precedence. This rule allows for optional features in libraries where a weak symbol provides a default implementation, and a strong symbol in the user's code provides a custom implementation.

Symbol Tables

Symbol tables are a crucial aspect of the compilation process in programming. They serve as a repository where information about the identifiers (symbols) used in a program is stored. These identifiers can be variable names, function names, constants, and data types. The symbol table is used by the compiler and linker to ensure that all symbols are correctly identified and accessible.

Functions Of Symbol Tables

- **Scope Management**: They help in managing the scope of variables. Local variables in different functions can use the same name without conflict because the symbol table will treat them as separate entries.
- **Storage Of Symbol Information**: Symbol tables store details such as the name of the symbol, its type, scope (local or global), and sometimes its memory location.
- **Type Checking**: Symbol tables are used during semantic analysis by compilers to check for type inconsistencies in operations.

Components Of Symbol Tables

- **Symbol**: The variable (symbol) that is present in one of source files.
- **Entry**: A boolean value that indicates if a symbol is present in the table.
- **Type**: The type of symbol that is / isn't present in the table: `static`, `extern`, `global`, `local`.
- **Location (Module)**: Where the symbol is defined in reference to the source files.
- **Section**: Location of where the symbol is stored in compilation.

Types

Symbols are categorized into main types:

- `extern`: The `extern` keyword is used to declare a variable or function and indicates that its definition is in another file or translation unit.
  - **Usage**: `extern` is primarily used when you need to access a variable or function defined in another source file or to declare the variable in a header file that multiple source files include.
  - **Characteristics**:
    - Does not allocate memory by itself.
    - Requires an external definition with a matching type.
    - Useful in managing global variables across different files.
- `global`: Global variables and functions are those defined outside any function (usually at the top of the source file) and can be accessed from any function in the program.
  - **Usage**: Global symbols are accessible throughout the program from any translation unit that includes a declaration of them. This is default for functions in C and C++.
  - **Characteristics**:
    - Stored usually in the global data segment.
    - Persistent for the lifetime of the application.
    - Can cause issues like name clashes and are generally discouraged in modern programming due to their impact on code maintainability and testing.
- `local`: Local variables are declared inside a function or block and can only be accessed within that function or block (scope-limited).
  - **Usage**: Local variables are used to store temporary state or intermediate results within a function.
  - **Characteristics**:
    - Stored on the stack (typically).
    - Automatically allocated and deallocated when the function is called and returns, respectively.
    - Not visible outside the function or block where they are declared.
- `static`: The `static` keyword can modify both local and global variables. It alters the storage duration of the variable it qualifies.
- **Usage**:

  - **Global Static**: When used outside any function, it restricts the scope of the variable to the file in which it is declared, making it a private global.
  - **Local Static**: When used within a function, the variable retains its value between function calls.

- **Characteristics**:
  - Local static variables are initialized only once, and they exist until the end of the program.
  - Global static variables are only accessible within the same translation unit (source file), protecting against namespace pollution.

Sections

For each symbol, there is a location where the symbol is stored upon compilation:

- `.bss` **(Block Started by Symbol)**: The section where uninitialized `static` variables, and `global` or `static` variables that are initialized to zero are stored. This section is used for declaring variables that are not initialized by the programmer. By default, the system initializes them to zero.
- `COMMON`: The section where uninitialized `global` variables are stored. This is a special section used in the context of weak linkage and tentative definitions. If a global variable is declared but not initialized (a common practice in C for external variables), it is typically placed in the `COMMON` section. This allows the linker to handle multiple tentative definitions.
- `.data`: The section where initialized `global` and `static` variables are stored. Both `global` and `static` variables in this context retain their values through execution.
- `.rodata` **(Read-Only Data)**: This section stores constant values and string literals that should not be modified, making them read-only at runtime.
- `.text`: This section contains the executable code of the program. It is where the machine instructions reside.

Exceptional Flow Control

Exceptional Control Flow (ECF) involves mechanisms that alter the normal sequential execution order of programs. ECF mechanisms include signals, process context switching, and system calls like `fork()`. These are essential for operating systems to perform efficient multitasking, handle asynchronous events, and manage multiple processes.

`fork()` System Call

One example of ECF is the `fork()` system call. This system call produces children from a parent process (and sometimes a child from an already existing child). The core tenants of the `fork()` system call are:

- **Control Flow Implications**: After a `fork()`, the child process may execute the same or different code based on the return value. This leads to complex flow control scenarios, especially when multiple `fork()` calls are nested or combined with other control flow mechanisms like loops and conditionals.
- **Functionality**: The `fork()` system call is used to create a new process by duplicating the calling process. The new process is referred to as the child process, while the original process is the parent.
- **Memory Handling**: Initially, both processes share the same physical memory pages, but typically a copy-on-write mechanism is used where pages are duplicated only if either process attempts to modify them.
- **Memory Sharing**: Initially, the child process shares the same memory segments (code, data, and stack) with the parent process. Modern operating systems typically use a copy-on-write mechanism where the actual copying of the memory pages is deferred until one of the processes attempts to modify a page.
- **Process Tree**: When `fork()` is executed, it returns twice: once in the parent process (returning the child's PID) and once in the child process (returning 0). This dual return allows both processes to execute the same subsequent code but often follow different branches depending on the return value.

- **Return Value**: `fork()` returns twice, once in the parent process and once in the child process. In the parent, it returns the PID of the newly created child, while in the child, it returns 0. If `fork()` fails, it returns -1 in the parent.

Signal Handling

In UNIX and UNIX-like systems, signals are one of the primary methods for exceptional control flow, providing a way for processes to interrupt or notify each other asynchronously. Signals can indicate events like division by zero, termination requests, or user-defined conditions. They are used for a variety of purposes including process control, inter-process communication, and handling asynchronous events.

The key concepts of signal handling are:

- **Signal Delivery**: When a signal is generated, the operating system delivers it to the target process. This process then interrupts its current task to handle the signal.
- **Signal Generation**: Signals can be generated by errors, explicit requests via system calls (like kill), or hardware exceptions.
- **Signal Handlers**: Processes can register signal handlers, specific functions that are executed when signals are received. If a signal handler is not set, a default action is taken (usually terminating the process).

`kill()` System Call

The `kill()` function is a critical tool in UNIX and UNIX-like operating systems, used primarily for sending signals to processes or groups of processes. It enables a process to communicate with other processes through predefined signals, which can indicate various system events or requests.

Overview Of `kill()`

- **Function Prototype**: `int kill(pid_t pid, int sig);`
- **Parameters**:
  - `pid`: The process ID of the target process to which the signal is sent. This can also be a special value to target a group of processes.
  - `sig`: The signal number to be sent. This can be any of the standard signals defined in `<signal.h>`, like `SIGKILL`, `SIGTERM`, `SIGSTOP`, `SIGUSR1`, etc.
- **Return Value**: Returns 0 on success, and -1 on failure, setting `errno` to indicate the error.

Usage Of `pid` Parameter

- **Positive Value (`pid` > 0)**: Sends the signal to the process with the specified process ID.
- **Negative One (`pid` $== -1$)**: Sends the signal to all processes for which the calling process has permission to send signals, except for the system processes and the process sending the signal.
- **Negative Value (`pid` < 0)**: Sends the signal to all processes in the process group whose ID is the absolute value of `pid`.
- **Zero (`pid` $== 0$)**: Sends the signal to all processes in the sender's process group, which typically includes all processes started from the same terminal.

Error Handling

On failure, `kill` sets `errno` to one of the following values:

- `EINVAL`: The signal number is invalid.
- `EPERM`: The process does not have permission to send the signal to any of the target processes.
- `ESRCH`: The target process or process group does not exist.

The `kill()` function is an essential aspect of UNIX system programming, providing robust capabilities for process control and inter-process communication. Understanding its behavior and implications is crucial for effective system management and application development in UNIX environments. This function illustrates the powerful, albeit low-level, mechanisms available in UNIX systems for managing the process lifecycle and system resources.

`signal()` System Call

The `signal` system call is essential for setting up signal handling in UNIX and UNIX-like operating systems. It allows processes to manage how they respond to the various signals they might receive, which are typically used to indicate system events, errors, or external interruptions.

Overview

- **Function Prototype**: `void (*signal(int sig, void (*func)(int)))(int);`
- **Parameters**:
  - `sig`: The signal number to handle. This is an integer representing one of the system-defined signals like `SIGINT`, `SIGTERM`, `SIGCHLD`, etc.
  - `func`: A pointer to a function that will handle the signal, or a special constant like `SIG_IGN` for ignoring the signal or `SIG_DFL` for default handling.
- **Return Value**: Returns the address of the previous signal handler for the specified signal, or `SIG_ERR` in case of error.

Key Features

- **Default Behavior**: Setting the handler to `SIG_DFL` restores the default action for the signal, which often results in the process being terminated or stopped.
- **Ignoring Signals**: By passing `SIG_IGN` as the handler, the process can ignore the specified signal (except for signals that cannot be caught or ignored, like `SIGKILL` and `SIGSTOP`).
- **Signal Handlers**: A signal handler is a function designated to be called when a specific signal is received. It should have the following prototype: `void handler(int sig);`

Signal Handling

- **Asynchronous Events**: Signals are asynchronous by nature, meaning they can interrupt the process at almost any point in its execution.
- **Atomic Operations**: Signal handlers should perform atomic operations or operate in a context where interruptions do not cause inconsistency or data corruption. They are typically used to set flags or handle simple state changes.

The `signal()` system call provides fundamental capabilities for handling asynchronous events in UNIX systems. It allows programs to define custom responses to various signals, which can enhance program robustness, enable graceful exits, and manage system resources effectively. Understanding and using `signal` effectively is crucial for advanced system programming and developing resilient applications in a UNIX environment.

Exceptional Control Flow (ECF) is a fundamental concept in systems programming, where the normal linear execution sequence of a program is interrupted or altered using mechanisms such as interrupts, signals, traps, system calls, and context switches. These mechanisms allow an operating system to perform more complex tasks like handling multiple processes, managing asynchronous events, and sharing resources among different programs efficiently.

Virtual Memory

Virtual Memory is a fundamental concept in modern computing systems, primarily designed to decouple the user's view of memory from the physical limitations of the available main memory. It provides an abstraction that allows each process to act as though it has its own contiguous block of addresses that it can read and write to,

independently of the actual physical memory available.

Key Features Of Virtual Memory

- **Abstraction Of Physical Memory**: Virtual memory abstracts the underlying physical memory hardware, allowing an operating system to use hardware like disk storage to extend available memory.
- **Memory Management**: It simplifies memory management by allowing the system to move memory pages between physical memory and disk transparently.
- **Process Isolation**: Each process operates in its own virtual address space, which isolates it from other processes and the operating system, enhancing security and stability.

Components Of Virtual Memory

- **Physical Address Space**: The actual RAM available on the system.
- **Virtual Address Space**: This is the contiguous address space that processes use to access memory. Each process has its own virtual address space, which is mapped to the physical memory.

Address Translation

- **Page Table**: A data structure used by the operating system to store mappings from virtual pages to physical frames.
- **Virtual Addresses** are translated to **Physical Addresses** using data structures like page tables.

Pages And Page Tables

- **Page**: A fixed-size block of memory. Virtual memory is divided into pages, which are mapped into physical memory frames of the same size.
- **Page Table**: Holds the mapping of virtual pages to physical frames. Modern systems often use a multi-level page table structure to optimize memory usage and access time.

Virtual Memory Translation

Problems that consist of taking a memory addresses and translating them into their virtual and physical addresses respectively consist of some common variables:

$$
\begin{aligned}
N = 2^n &\quad \text{(Number of virtual address bits } (n)\text{)} \\
M = 2^m &\quad \text{(Number of physical address bits } (m)\text{)} \\
P = 2^p &\quad \text{(Page size in bytes } (P)\text{)} \\
S = 2^s &\quad \text{(Number of TLB index bits } (s)\text{)}
\end{aligned}
$$

When translating an address into both the virtual address and physical address, there are specific values for both the virtual and physical addresses:

$$
\begin{aligned}
\text{Physical Page Offset (PPO)} &= \text{First } p \text{ bits from, right to left of address in binary} \\
\text{Physical Page Number (PPN)} &= \text{Address found from TLB or Page Table, valid bit must be set} \\
\text{Physical Address (PA)} &= \text{PPN + PPO (Concatenated, not summed)} = m \text{ in size} \\
\text{Virtual Page Offset (VPO)} &= \text{First } p \text{ bits from, right to left of address in binary} \\
\text{Virtual Page Number (VPN)} &= \text{Bits from } p \text{ to } n-1 \text{ of address in binary} \\
\text{Virtual Address (VA)} &= \text{VPN + VPO (Concatenated, not summed)}, = n \text{ in size} \\
\text{TLB Index (TLBI)} &= (p, p+s-1) \text{ bits from Virtual Address} \\
\text{TLB Tag (TLBT)} &= (p+s, n-1) \text{ bits from Virtual Address}
\end{aligned}
$$

The procedure for translating an address into both its virtual and physical address representation is:

1. Translate the original address from hexadecimal `0xXXXX` into binary.
   - Add padding (trailing zeros) if original address is smaller in total bits than virtual or physical address.
2. Calculate the number of offset bits p.
3. Calculate the VPO and PPO of the address in binary, convert to hexadecimal.
4. Calculate the VPN of the address in binary, convert to hexadecimal.
5. Calculate the number of TLB Index bits s.
   - Calculate the TLBI from the VA, translate to hexadecimal.
   - Calculate the TLBT from the VA, translate to hexadecimal.
6. Refer to the TLB to determine if there is a TLB hit.
   - A valid TLB hit occurs when an index (referred to as the set #), has a valid tag (V = Y) in any of the ways.
     - If there is a TLB hit, the PPN can be found from the TLB.
     - If there is no TLB hit, we must refer to the Page Table to see if there is a valid PPN.
7. Refer to the Page Table with the aforementioned VPN to determine the PPN.
   - The valid tag must be set in this case.
8. Calculate the PA if there was indeed a PPN either from step 6 or 7.

Virtual Memory is a sophisticated system that underpins modern operating systems, allowing them to efficiently use both the physical memory (RAM) and secondary storage (like HDDs or SSDs) to manage the execution of multiple processes. This system enhances the computer's multitasking capabilities, improves security and isolation among processes, and abstracts complex memory management details from the user and application programs.

Allocation `malloc`

There are different forms memory allocation, the key concepts of memory allocation are:

- **Boundary Tag Method**:
  - This method involves using the headers and footers of blocks as "tags" to hold information about the block size and allocation status. These tags help in merging adjacent free blocks during deallocation (coalescing), effectively reducing fragmentation.
  - Both the header and footer of a block generally contain the same information so that blocks can be merged efficiently whether the allocator is moving forward or backward through the heap.
- **Implicit List Allocator**:
  - An implicit list uses the blocks themselves to keep track of memory allocations. Each block on the heap typically contains a header and possibly a footer that store metadata about the block, such as its size and whether it's allocated or free.
  - The allocator traverses the heap from the beginning to find a free block that fits the size requirement of a malloc call (first-fit, best-fit, etc.).
- **Malloc And Free Operations**:
  - **`malloc(size)`**: Allocates a block of at least **`size`** bytes and returns a pointer to the allocated memory. The allocator might also include additional bytes for alignment and metadata (headers and footers).
  - **`free(ptr)`**: Deallocates the block of memory pointed to by `ptr`, potentially merging it with adjacent free blocks to form larger free blocks and reduce fragmentation.

Best Fit

The Best Fit method searches the entire free list and takes the smallest block that is adequate to fulfill the request. This approach aims to find the closest matching block in size to the requested memory.

Advantages

- **Efficient Utilization Of Space**: It tends to use memory more efficiently over the long term, as it leaves larger blocks of memory available for future allocations.
- **Reduced Fragmentation**: By allocating the smallest block that meets the size requirement, Best Fit minimizes the leftover space in memory blocks, reducing external fragmentation.

Disadvantages

- **Maintenance Overhead**: Best Fit requires more complex bookkeeping to keep track of the sizes of all free blocks, which can add overhead.
- **Performance**: Scanning the entire list to find the optimal block can be slower, especially if the free list is long. This comprehensive search increases allocation time.
- **Possible Internal Fragmentation**: Although it reduces external fragmentation, it may increase internal fragmentation if the best fit block is slightly larger than needed.

First Fit

The First Fit method of memory allocation scans the heap from the beginning and allocates the first block that is large enough to satisfy the request. It stops searching as soon as it finds a block of sufficient size.

Advantages

- **Simplicity**: The algorithm is straightforward and easy to implement.
- **Speed**: First Fit generally performs faster than Best Fit for allocation because it stops searching as soon as it finds a free block that fits, rather than continuing to search for a better fit.

Disadvantages

- **Fragmentation**: It may lead to higher fragmentation compared to Best Fit because it can leave smaller unusable spaces scattered throughout the heap. This happens as it might skip smaller blocks that are a better fit in favor of the first block that fits the request.
- **Suboptimal Allocation**: Over time, it could result in larger free blocks at the end of the heap being underutilized.