Computer Systems

Martin de Spirlet

Week	Unit	Title
1	1	Data Representation and Manipulation
2	2	Introduction to Computer Architecture
	3	Instructions, Assembly Language and Machine Code
3	4	Compilation, Interpretation and the JVM
	5	Subroutines and Stacks
4	6	The JVM and Java Bytecode
	7	Complexity of Algorithms
5	9	Introduction to Operating Systems
	10	Computer Architecture and OS Structures
6	11	Process Management
	12	Process Scheduling
7	14	Concurrency and Synchronisation

1 Data Representation and Manipulation

1.1 Binary

Computers consist of a collection of switches called transistors, which can either be on or off. This leads to binary representation, where off is 0 and on is 1. All data and instructions are stored in binary. Binary digits are known as 'bits'. It is possible to build computers that use other number systems, but it is less expensive and more reliable to use basic components with only two states.

1.2 Characters

Characters are represented as binary values using text encoding schemes such as ASCII and Unicode. Modern schemes support internationalisation and tend to use 8 bit, 16 bit or 32 bit to encode characters. Strings are sequences of characters.

1.3 Number Bases

1.3.1 Decimal to Binary, Octal and Hexadecimal

234_{10} :

$$\left.\begin{array}{r|l} 2\lfloor 234 & 0 \\ 2\lfloor 117 & 1 \\ 2\lfloor 58 & 0 \\ 2\lfloor 29 & 1 \\ 2\lfloor 14 & 0 \\ 2\lfloor 7 & 1 \\ 2\lfloor 3 & 1 \\ 2\lfloor 1 & 1 \end{array}\right\} = 11101010_2$$

234_{10} :

$$\left.\begin{array}{r|l} 8\lfloor 234 & 2 \\ 8\lfloor 29 & 5 \\ 8\lfloor 3 & 3 \end{array}\right\} = 352_8$$

234_{10} :

$$\left.\begin{array}{r|l} 16\lfloor 234 & 10 \\ 16\lfloor 14 & 14 \end{array}\right\} = EA_{16}$$

1.3.2 Binary, Octal and Hexadecimal to Decimal

101011_2 :

$$\begin{aligned} 1 \times 2^0 &= 1 \\ 1 \times 2^1 &= 2 \\ 0 \times 2^2 &= 0 \\ 1 \times 2^3 &= 8 \\ 0 \times 2^4 &= 0 \\ 1 \times 2^5 &= \underline{32} \\ &= 43_{10} \end{aligned}$$

724_8 :

$$\begin{aligned} 4 \times 8^0 &= 4 \\ 2 \times 8^1 &= 16 \\ 7 \times 8^2 &= \underline{448} \\ &= 468_{10} \end{aligned}$$

ABC_{16} :

$$\begin{aligned} C \times 16^0 &= 12 \\ B \times 16^1 &= 176 \\ A \times 16^2 &= \underline{2560} \\ &= 2748_{10} \end{aligned}$$

1.3.3 Binary to Octal and Octal to Binary

$1011010111_2 = 1327_8$:

1 011 010 111

1 3 2 7

$705_8 = 111000101_2$:

7 0 5

111 000 101

1.3.4 Binary to Hexadecimal and Hexadecimal to Binary

$1010111011_2 = 2BB_{16}$:

10 1011 1011

2 B B

$10AF_{16} = 1000010101111_2$:

1 0 A F

0001 0000 1010 1111

1.3.5 Octal to Hexadecimal and Hexadecimal to Octal

Conversions between octal and hexadecimal can be performed by first converting to binary.

1.4 Integers in Binary

Counting in powers of 2.

Prefix	Power of 2	Value
kibi	2^{10}	1024
mebi	2^{20}	1048576
gibi	2^{30}	1073741824
tebi	2^{40}	1099511628000

1.4.1 Overflow

In a computer, an integer is represented by a fixed number of bits. The maximum value that can be stored in an unsigned integer of n bits is $2^n - 1$. It is possible that the addition of two n bit numbers yields a result that requires $n + 1$ bits.

In Java, the 'overflow' bits are lost with no error. The remaining bits give the wrong answer. It is important to make sure the data type used is big enough for the values it will represent.

1.4.2 Two's Complement

In 8 bit arithmetic, $255 + 1$ appears to be 0 due to overflow. In this case, 255 is behaving like -1. This leads to the 'two's complement' representation of negative integers in binary.

The two's complement representation of $-x$ is equivalent to the unsigned binary representation of $2^n - x$. Another method to find the representation is to flip all the bits of the binary representation of x and add 1 to the least significant bit.

Using two's complement, a signed binary integer of n bits can hold any value from -2^{n-1} to $2^{n-1}-1$, inclusive. The most significant bit represents -2^{n-1}. If the number of bits in a signed number is increased, the new bits are given the same value as the most significant bit. This is known as sign extension.

In Java, all integers are signed.

1.5 Real Numbers in Binary

1.5.1 Fixed Point Decimal to Binary

The integral part is converted as usual. The fractional part is converted by doubling as follows.

$$
\begin{aligned}
0.537_{10} = 0.100010_2 :& \\
0.537 \times 2 &= \underline{1}.074 \\
0.074 \times 2 &= \underline{0}.148 \\
0.148 \times 2 &= \underline{0}.296 \\
0.296 \times 2 &= \underline{0}.592 \\
0.592 \times 2 &= \underline{1}.184 \\
0.184 \times 2 &= \underline{0}.368
\end{aligned}
$$

1.5.2 Fixed Point Binary to Decimal

$$
\begin{aligned}
1101.0101_2 :& \\
1 \times 2^{-4} &= \ 0.0625 \\
0 \times 2^{-3} &= \ 0 \\
1 \times 2^{-2} &= \ 0.25 \\
0 \times 2^{-1} &= \ 0 \\
1 \times 2^{0} &= \ 1 \\
0 \times 2^{1} &= \ 0 \\
1 \times 2^{2} &= \ 4 \\
1 \times 2^{3} &= \ \underline{8} \\
&= 13.3125_{10}
\end{aligned}
$$

1.5.3 Floating Point Numbers

Fixed point is convenient and intuitive, but has two major problems.

1. Numerical precision — only values that are multiples of the smallest used power of two can be represented.
2. Numerical range — fractional precision comes at the expense of numerical range.

Floating point representation in binary is similar to scientific notation in decimal. In binary, real numbers can be represented in the form $\pm m \times 2^e$. Floating point numbers consist of a sign bit ($\pm$, 0 for positive or 1 for negative), a mantissa (m, the significant bits) and an exponent (e, two's complement signed binary).

S	Offset exponent, e'	Normalised mantissa, m'

The stored representation of a floating point number.

Since the mantissa is normalised (unless the value is small enough to represent without an exponent), the leading bit is almost always 1. It is therefore omitted from the stored representation. The exponent has a bias (offset) of $2^{n-1} - 1$ added to it for engineering purposes.

Floating point data types in Java.

Type	Bits				Bytes	Exponent bias
	Sign	Mantissa	Exponent	Total		
`float`	1	23	8	32	4	127
`double`	1	52	11	64	8	1023

With the 52 bit mantissa of a `double` (and the additional hidden bit), $2^{53} \approx 8 \times 10^{15}$. Hence, the `double` data type can be used to represent 15 significant decimal digits.

In Java, special values are returned for floating point overflow and other unusual circumstances. For example, if the result is too large for the `double` data type, `Double.POSITIVE_INFINITY` or `Double.NEGATIVE_INFINITY` are returned. If the result is indistinguishable from zero but known to be negative, `-0.0` is returned. If the result is not a real number, `Double.NaN` is returned.

1.5.4 Numerical Precision

Floating point arithmetic loses accuracy in its less significant digits. This is an issue even in values of type `double`.

It is a bad idea to use floating point representation for money — i.e. with integral pounds and fractional pence — as this will result in rounding errors. Taking £0.10 as an example, $0.10_{10} = 0.0\overline{0011}_2$.

Since currency is inherently integral, integer types with suitable range, such as `int`, `long` or `java.math.BigInteger`, should be used to represent multiples of the smallest denomination.

2 Introduction to Computer Architecture

2.1 Anatomy of a Computer System

A computer is a complex system (machine) that can be instructed to carry out sequences of arithmetic or logical operations automatically via computer programming.

A computing device must be able to

- load a program (input interface),
- process instructions in the correct order (track progress, storage and decoding of instructions),
- access data according to its instructions (local storage),
- perform computations (calculation 'engine'),
- make decisions according to its computations (control mechanism), and
- send results to an external device (output interface).

Thus, all computers, regardless of their implementing technology, have five basic subsystems.

1. Memory
2. Control unit
3. Arithmetic logic unit (ALU)
4. Input unit
5. Output unit

2.1.1 Memory

Memory locations have finite capacity. Data may not fit in a memory location. Blocks of four or eight bytes are used so often as a unit that they are known as memory 'words'.

Computer memory is known as random access memory (RAM). This simply means that the computer can refer to memory locations in any order.

2.1.2 Control unit

The control unit of a computer is where the fetch/execute cycle occurs. It fetches a machine instruction from memory and performs other operations of the fetch/execute cycle accordingly.

2.1.3 Arithmetic Logic Unit (ALU)

The arithmetic logic unit carries out each machine instruction with a separate circuit. It contains various circuits for arithmetic and logic, such as addition, subtraction, multiplication, comparison and logic gates.

2.1.4 Input and Output Units

These components are the wires and circuits through which information travels into and out of a computer. Without input or output, a computer is useless.

Peripherals connect to the computer through input/output (I/O) ports. They are not considered parts of the computer.

2.2 The Fetch-Execute Cycle

A program is loaded into memory and the address of the first instruction is placed in the program counter (PC).

The fetch-execute cycle proceeds as follows.

1. Instruction fetch (IF)
2. Instruction decode (ID)
3. Data fetch (DF) / operand fetch (OF)
4. Instruction execution (EX)
5. Result return (RR) / store (ST)

2.2.1 Instruction Fetch (IF)

The instruction at the memory address given by the PC is copied to the instruction register of the control unit. The PC is incremented to point at the next instruction to be fetched.

2.2.2 Instruction Decode (ID)

The ALU is prepared for the operation specified by the instruction. The decoder finds the addresses of the instruction operands. These addresses are passed to the ALU circuit that fetches the operands from memory in the next stage. The decoder also finds the destination address for the result. This is passed to the RR circuit.

2.2.3 Data fetch (DF)

The operands are copied from the specified memory addresses into the ALU circuits. The data remains in memory and is not destroyed.

2.2.4 Instruction Execute (EX)

The ALU performs the operation on its operands. The result is held in its circuitry.

2.2.5 Result Return (RR)

The result of the EX stage is stored at the specified destination memory address. The cycle begins again.

2.3 Machine Instructions

A computer "knows" very few instructions. The decoder hardware in the control unit recognises, and the ALU performs, of the order of 100 instructions. Everything that a computer does must be

reduced to some combination of these primitive instructions. Computers can carry out millions of these instructions per second.

3 Instructions, Assembly Language and Machine Code

3.1 The von Neumann Architecture

3.1.1 Central Processing Unit (CPU)

The central processing unit (CPU) consists of a control unit, ALU and registers. These registers include

- the program counter (PC) (holds the address of the next instruction),
- the instruction register (holds the instruction currently being decoded or executed),
- the address register (holds a memory address from which data will be fetched, or to which data will be returned), and
- the accumulator (holds temporarily the results of ALU computations).

3.1.2 System Bus

The system bus includes

- a control bus (carries commands from the CPU and returns status signals from devices),
- an address bus (carries information about the device with which the CPU is communicating, namely physical memory addresses), and
- a data bus (carries the data being processed).

3.1.3 Memory

In the von Neumann architecture, the memory contains both program instructions and data.

3.1.4 Clock

A clock cycle is a signal that oscillates between high and low. It is used to coordinate actions. The rate of the fetch-execute cycle is determined by the clock. Instructions begin execution on the rising edge of the signal. The time between rising edges is known as the 'clock period'. The time between the rising and falling edges of the signal is known as the 'clock width'. The number of clock cycles per second is used to determine the speed of a CPU.

Modern computers attempt to start an instruction on each clock tick. Since each stage of the fetch-execute cycle is handled by separate circuitry, the fetch unit is freed to start the next instruction before the previous instruction is complete. This process is known as 'pipelining'. When the pipeline is filled, five instructions are in process at a time, each at a different stage of the fetch-execute cycle Additionally, one instruction is finished on each clock tick, making it appear as though the computer is running one instruction per tick.

3.2 The Harvard Architecture

The main difference between the von Neumann and Harvard architectures is their storage of instructions and data in memory.

A von Neumann (or 'stored-program') machine stores its instructions and data in a shared memory. This means that an instruction fetch and a data operation cannot occur at the same time because they share a common bus. This is known as the 'von Neumann bottleneck' and can impinge on the performance of the system. However, the architecture allows for self-modifying code that can tune its performance at runtime.

A Harvard machine stores its instructions and data in separate memories. This makes the machine more complex, but allows instruction fetch and data operation to occur at the same time.

3.3 MIPS R4000

MIPS R4000 uses the modified Harvard architecture. Instructions and data are stored in the same memory (as in von Neumann), but they have separate interfaces (as in Harvard).

MIPS R4000 was one of the first 64 bit microprocessors. It features integrated caches (primary on-chip and secondary off-chip), an integrated floating point unit and a deep pipeline. It uses MIPS III — a compact instruction set with a regular format — which makes it easy to convert from high-level code to machine code.

3.3.1 Registers

The CPU contains 32 registers denoted `$0`–`$31` (or `r0`–`r31`). The registers can be either 32 bit or 64 bit wide, depending on the mode of operation. `$0` (`r0`) always stores zero. `$31` (`r31`) is the link register used by jump and link instructions. It should not be used by other instructions.

The ALU takes input from up to two registers and sends output to registers only.

3.3.2 Deep Pipeline

The typical MIPS pipeline consists of the following five stages.

1. Instruction fetch (IF)
2. Instruction decode (ID) and register fetch (RF)
3. Execute (EX)
4. Memory access (MEM)
5. Write back (WB)

MIPS R4000 is super-pipelined; its pipeline contains eight stages rather than the regular five. The additional stages exist due to the extension of the IF and MEM stages to account for cache overheads. When the pipeline is full, eight instructions are in process at a time — i.e. the pipeline is eight-deep.

3.4 Assembly Language

Binary is the only form in which a computer can be given instructions. Computers can be programmed to translate instructions expressed in other forms into binary code. This is known as 'assembly'.

Assembly language is a human-readable alternative to machine language. A computer can scan assembly code and convert words to binary. The binary is assembled into instructions.

High-level languages are compiled to assembly language, which is then assembled into binary.

3.5 Instruction Sets

Every machine has a unique instruction set architecture (ISA). This is the set of primitive instructions that the CPU can execute. MIPS R4000 uses the MIPS III ISA. Each MIPS III instruction has a 32 bit representation. It can be represented as human-readable assembly code, or binary machine code. There are three types of MIPS III instruction.

I-type requires one or two registers and an operand.
R-type requires three registers.
J-type requires an operand.

4 Compilation, Interpretation and an Overview of the Java Virtual Machine (JVM)

4.1 Levels of Programming Languages

High-level programming languages, such as Java, C, C++ and C# are close to the English language and, therefore, easy for humans to read and write. Code written in these languages is more concerned with problem-solving than implementation. Programmers are less likely to make errors when programming in these languages and do not need to worry about memory management and other complexities.

Low-level languages, such as machine code, use instructions stored in memory (opcodes) and refer to specific locations in memory. Code written in these languages reflects the processes being used rather than the problem being solved. Machine language is difficult for humans to read and write. Since it runs directly on hardware, it is also hardware-specific.

Assembly language is slightly higher than machine code. It uses mnemonics so that it can be more easily read or written by humans. This is used to translate high-level languages to machine code.

4.2 Language Processing Systems

To run on a computer, a program must be supplied in machine code. Programs written in high-level languages are converted to binary through a series of phases.

1. Preprocessor (prepares high-level code for the compiler)
2. Compiler/interpreter (produces assembly/intermediate code)
3. Assembler (produces relocatable machine code)
4. Linker/loader (produces object code)

The preprocessor organises and prepares the source code for the compiler. This includes importing packages, macro processing and file inclusion.

The compiler reads the entire source code in one go and creates tokens. It checks the meaning of these tokens and generates intermediate assembly code. Alternatively, the interpreter reads the source statement by statement, converting each statement to intermediate code that is executed immediately.

The assembler calculates relative addresses for jumps and produces relocatable machine code.

The linker combines assembled parts into a whole. Alternatively, the loader places the code directly into memory.

4.3 High-Level Program Execution

A program written in a high-level language can be run in two different ways.

1. Compiled into a program in the native machine language of the target machine
2. Directly interpreted and executed via simulation on the target machine

4.4 Compilation

A compiler converts source code into assembly, object or machine code that does the same thing as the original. Object code is usually relocatable, so it can later be linked or loaded.

This is done just once for each program. Hardware features can be exploited to optimise object code so that it will run more quickly.

However, this is more difficult than interpretation, as the compiler must understand the entire program in order to convert it. Compilers are also hardware-dependent, so cannot run on different platforms.

A compiler runs on the same platform as the target machine. A cross-compiler can produce code that will run only on a separate platform.

A compiler can be divided into a front-end and a back-end. Both perform their operations in a sequence of phases that each generate a data structure to be used by the following phase.

Front-end:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generation

Back-end:

5. Optimisation
6. Code generation

4.4.1 Lexical Analysis

During lexical analysis, the compiler breaks the source code into meaningful words known as 'lexemes' and generates a sequence of tokens from the lexemes. A lexeme is a word recognised by the compiler according to the language specification. This includes keywords, identifiers, operators etc. A token is an object describing a lexeme. Along with the value of the lexeme, it includes information about the type of the lexeme (keyword, identifier, operator etc.).

4.4.2 Syntax Analysis (Parsing)

During syntax analysis, the compiler interprets the structure of the source code. This is known as 'parsing' and involves grouping tokens into higher-level constructs. This phase produces an abstract syntax tree (AST).

This is also the phase where syntax errors are detected. If no error is found, the compiler continues to the semantic analysis phase.

4.4.3 Semantic Analysis

During semantic analysis, the compiler interprets the meaning of the AST. The compiler checks that the program is consistent with the rules of the source language.

The compiler can deal with ambiguity in a number of ways.

- Type inference (the compiler annotates nodes in the AST with inferred type information)
- Type checking (the compiler checks that all values assigned to variables are of the correct type)
- Symbol management (the compiler uses a symbol table to determine whether variables have been declared or whether multiple variables with the same name exist in the same scope)

Semantic analysis produces an annotated AST.

4.4.4 Intermediate Code Generation

During intermediate code generation, the compiler produces code in an intermediate language between that of the source code and machine language.

4.4.5 Optimisation

During optimisation, the compiler simplifies or removes unnecessary code. This allows the program to run more quickly or use fewer resources.

4.4.6 Code Generation

During code generation, the compiler maps the optimised intermediate code to the target machine language.

4.5 Interpretation

An interpreter is a program that follows the source code and performs appropriate actions accordingly. A CPU can be viewed as a hardware implementation of an interpreter for machine code.

Interpreters begin execution immediately and facilitate interactive debugging and testing. A user can read and modify the values of variables and invoke procedures from the command line. Interpreted languages are not hardware-dependent. However, interpreted programs have slower execution than compiled programs.

Compilers are compute-intensive and require more preparation time. Additionally, compiled programs are hardware-dependent. However, they can run very quickly.

4.6 Combined Compilation and Interpretation

Combined compilation and interpretation is a method by which programs are compiled to an intermediate language that can be interpreted efficiently. Execution of programs produced by this method is slower than pure compilation, but quicker than pure interpretation.

Compilation for any platform requires only a single compiler that is independent of the target CPU. Interpretation of the intermediate language is delegated to the target CPU.

Java was conceived as a language that uses combined compilation and interpretation. The `javac` compiler converts `.java` source code to `.class` bytecode, which is interpreted by the Java Runtime Environment (JRE) on the Java Virtual Machine (JVM).

4.7 Compilation and Execution on Virtual Machines

A virtual machine executes an instruction stream using software rather than hardware. Virtual machines emulate hardware using software and are, therefore, hardware-independent. Virtual machines are used in languages such as Java, Pascal and C#.

Java compilers generate bytecode that is interpreted by the JVM. This requires a JVM to exist on the target machine. The JVM may translate portions of bytecode to machine code at runtime via just-in-time (JIT) compilation if it finds those portions are used frequently.

5 Subroutines and Stacks

5.1 How Subroutines Work

5.1.1 Call and Return Instructions

When execution jumps from line n to a subroutine, there needs to be a way to instruct the computer to continue execution on line $n + 1$ once the subroutine is complete.

Two hypothetical instructions that could be used to achieve this are `call`, which jumps to the subroutine and stores the return address (the current PC value) somewhere suitable, and `ret`, which reads the return address from where it was stored and loads it into the PC.

5.1.2 Method 1 — Storing Return Address as First Byte

Using this method, the return address is stored as the first byte of the subroutine. The instruction `call N` would store the return address at location N and begin executing the subroutine at location $N + 1$. The instruction `ret N` would load the return address stored at location N into the PC.

The disadvantage of this method is that only one return address can be stored for each subroutine. Consequently, a subroutine could not call itself, as this would require two return addresses (one for the original call and another for the recursive call).

FORTRAN — the first commercially available high-level programming language — disallowed recursion in order to implement this method.

5.1.3 Introduction to Stacks

A stack is a data structure that can flexibly store a variable number of bytes. Stacks employ 'last in, first out' (LIFO) semantics. Elements are 'pushed' to or 'popped' from the top of the stack. A stack pointer (SP) is a CPU register that points to the top of the stack. It is not necessary to know where the bottom of the stack is, but programmers must make sure they do not attempt to pop from an empty stack.

When a value is pushed to the stack, it is written to memory at the address given by the SP. The SP is then incremented.

When a value is popped from the stack, the SP is decremented, then the value is read from memory at the address given by SP. Popped values remain in memory and are later overwritten by pushed values.

5.1.4 Method 2 — Storing Return Address on a Stack

Using this method, the return address is pushed to a stack when the subroutine is entered. The instruction `call N` would push the return address to the stack and begin executing the subroutine at location *N*. The instruction `ret` would pop the return address from the stack and load it into the PC.

This method allows multiple return addresses to be stored and, therefore, makes recursion possible.

5.1.5 Saving Registers

A subroutine may need to use CPU registers in its calculations. However, the previous register values will need to be restored when execution returns to the caller.

A common solution to this problem is to push all register values to the stack at the beginning of a subroutine (after the return address has been pushed) and to pop the register values from the stack at the end of the subroutine (before the return address is popped). Register values are popped in reverse order.

The return address and saved registers are known as a 'stack frame'.

5.2 Stacks for Calculations

5.2.1 Reverse Polish Notation (RPN)

Reverse Polish notation (RPN) or postfix notation is a method of writing calculations in which operators follow their operands and the order of operations is as written (no brackets are needed). This differs from standard infix notation, in which operators are written between or around their operands and the order of operations is determined by precedence.

RPN is a useful way of implementing calculations with stacks. If the next item in the input stream is a value or variable, it is pushed to the operand stack. If the next item is an operation, the relevant number of operands is popped from the operand stack, the operation is performed, and the result is pushed to the operand stack.

The following infix calculation and postfix calculation are equivalent.

$$(5 + 2) \times \sqrt{x \times x + y \times y} + 8$$

$$5\ 2 + x\ x \times y\ y \times + \sqrt{\ } \times 8 +$$

One notation that shows the effect of an operation, such as subtraction, on a stack is as follows.

$$\ldots, \text{val1}, \text{val2} \quad \rightarrow \quad \ldots, \text{val1} - \text{val2}$$

Java bytecode uses operand stacks for calculations. In Java, each method call has its own operand stack.

5.2.2 Stack Machines

Stack machines use a return stack for subroutine return, and an operand stack for RPN calculations. Registers are not needed for calculations. This allows more space for calculations, and machine instructions do not need to specify registers. However, it is difficult to know where things are on the stack.

The term 'operand' could refer either to the byte(s) after the instruction opcode in memory, or the entries in the operand stack.

5.3 Bitwise Boolean Operations

The XOR operation is an exclusive OR. It returns true only if exactly one operand is true.

Truth tables.

A	B	A AND B	A OR B	A XOR B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

5.4 Conditional and Unconditional Jumps

No stacks are used when an unconditional jump is executed. Conditional jumps are only executed if a condition is true; if the condition is false, execution continues to the next instruction.

Java bytecode contains six comparisons that are used in conditional jump instructions. There are two types of conditional jump instruction. Jump instructions of the form `ifXX` pop one value from the stack and compare it to zero. Jump instructions of the form `if_cmpXX` pop two values from the stack and compare them to each other.

For example, `ifeq N` pops a value from the stack and jumps to location *N* if the value is zero. The instruction `if_cmpeq N` pops two values from the stack and jumps to location *N* if they are equal.

Conditional jump instructions.

Comparison		Instruction	
Symbol	Mnemonic	`ifXX`	`if_cmpXX`
$=$	`eq`	`ifeq`	`if_cmpeq`
$<$	`lt`	`iflt`	`if_cmplt`
$\leq$	`le`	`ifle`	`if_cmple`
$\neq$	`ne`	`ifne`	`if_cmpne`
$>$	`gt`	`ifgt`	`if_cmpgt`
$\geq$	`ge`	`ifge`	`if_cmpge`

6 The Java Virtual Machine and Java Bytecode

6.1 Execution on the Java Virtual Machine

The `javac` compiler compiles Java source code in `.java` files to intermediate-level Java bytecode in `.class` files. As long as there is a JVM on another platform, the bytecode can be run on that platform without recompilation. This feature is known as 'portability'.

The JRE loads Java bytecode files, verifies their internal consistency and executes their methods using the JVM. Different CPUs or operating systems can run the same Java bytecode but use different JREs.

6.2 Stack Frames

Every time a method is called in Java, a stack frame is constructed for it. Each frame has

- a reference to the frame of its calling method,
- space for local variables,
- space for an operand stack
- a PC and SP (for the JVM, not the CPU), and
- other items.

6.2.1 Variable and Operand Storage

All local variables and operand stack entries are stored as 4 bytes. The `long` and `double` data types, which need 8 bytes, are stored as two consecutive entries.

Storage of data types.

Data type	Size		Slots
	/ bit	/ B	
`bool`	1		1
`byte`		1	1
`char`		2	1
`short`		2	1
`int`		4	1
`float`		4	1
reference		4	1
`long`		8	2
`double`		8	2

6.2.2 Local Variables

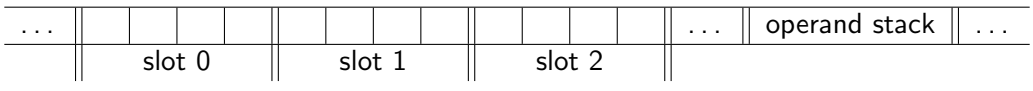
In the context of Java bytecode, local variables include:

1. `this` reference (for non-static methods only)
   - a reference to the object on which the method was called
   - stored as local variable in slot 0
2. Parameters of the method
   - stored from slot 0 onwards for static methods
   - stored from slot 1 onwards for non-static methods
3. Variables declared inside the method
   - stored in slots immediately after parameters

Instance variables and class (static) variables are not stored as local variables since they are defined outside of methods.

6.2.3 Slot Numbers

Java bytecode does not refer to local variables by their identifiers in the source code. Instead, it refers to them by their slot numbers. A local variable of the `long` or `double` data types takes the number of the first of its pair of slots.



Local variable slots in the stack frame.

6.2.4 Stack Overflow

The operand stack can never overflow. The Java compiler calculates exactly how much space is required and the loader checks each method to verify that no operand stack underflow or overflow is possible. Operand stack overflow is different from stack overflow, which occurs when a new frame is needed, but there is not sufficient memory.

6.3 Java Bytecode Instructions

Each Java bytecode instruction takes up at least one byte (for its opcode). It may also have one or more operand bytes. For each opcode, there is a corresponding mnemonic.

A single operand may consist of multiple bytes. In this case, the operand is stored with its most significant byte first (in the smaller address). This order is known as 'big endian'.

6.3.1 Arithmetic

Each arithmetic instruction has a prefix letter that denotes the data type on which it operates. For example, the `iadd` instruction pops two entries from the operand stack, adds them as if they are of the type `int`, and pushes them back onto the stack.

Mnemonic prefixes.

Data type	Prefix	Addition
`byte`	`b`	
`short`	`s`	
`int`	`i`	`iadd`
`long`	`l`	`ladd`
`float`	`f`	`fadd`
`double`	`d`	`dadd`
reference	`a`	

Errors are likely to occur if an instruction is used on the wrong data type. For example, calling `fadd` on `int` entries will result in the wrong answer since the instruction expects entries in floating point representation. If there is only one operand on the stack and the instruction expects two, operand stack underflow will occur. This can occur when `ladd` is called on `int` entries.

No checks are performed during execution. The JRE verifies the bytecode when the class is loaded. This includes checking that types are used consistently.

6.3.2 Pushing

The `bipush <1_byte_operand>` instruction has two prefixes: `b` for `byte` and `i` for `int`. This instruction sign extends its `byte` operand (1 B) to an `int` value (4 B) and pushes the result to the operand stack. Similarly, there exists an `sipush <2_byte_operand>` instruction to push a `short` to the operand stack as an `int`.

There exist seven instructions for pushing common integer constants to the operand stack that do not require an operand. Instructions that take no operand help to save space in the bytecode.

Load instructions of the form `iload <slot_number>` push local variables (indicated by their slot numbers) onto the operand stack. There are special load instructions that take no operand for pushing values stored in common slots to the operand stack (`iload_0`, `iload_1`, `iload_2`, `iload_3`).

Instructions for pushing common integer constants.

Instruction	Value pushed
`iconst_m1`	-1
`iconst_0`	0
`iconst_1`	1
`iconst_2`	2
...	...
`iconst_5`	5

6.3.3 Popping

Store instructions pop entries from the operand stack into local variable slots. For example, `istore <slot_number>` or `istore_2`.

6.3.4 Jumps

In Java bytecode, the source and destination of a jump must be within the same method. It is not possible to jump to another method.

Unconditional jumps are achieved with the `goto <2_byte_offset>` instruction. The operand is a 2 B offset that is added to the address of the instruction to calculate the address of the next instruction to execute. There exists a version of this instruction that takes a 4 B operand to allow for larger jumps: `goto_w <4_byte_offset>`.

Conditional jumps are achieved with instructions of the form `ifeq <offset>` and `if_icmpeq <offset>`.

6.4 Call and Return

6.4.1 The Stack

If method `A` calls a static method `B(p0, p1)` that returns a result:

1. method `A` calculates the arguments on its operand stack,
2. a stack frame is constructed for method `B`,
   - the arguments are popped from the operand stack of method `A`,
   - the values are used to initialise the local variables `p0` and `p1` of method `B`,
3. method `B` calculates the result on its stack frame
4. the result is pushed to the operand stack of method `A`, and
5. execution returns to method `A`, discarding the stack frame of method `B`.

On the operand stack of method `A`, this has the effect of

$$\ldots, \mathsf{p0}, \mathsf{p1} \quad \rightarrow \quad \ldots, \mathsf{result}$$

Each frame has its own PC. While method `B` is under execution, its PC is used. When execution returns to method `A`, it resumes with its old PC value. The local variables of method `A` are unchanged by method `B`.

Linked frames have the effect of a return stack. A chain of linked frames in the JVM is officially known as a stack.

6.4.2 Runtime Constant Pool

The class variables, instance variables and methods of a class may be used by other classes. Since classes are compiled separately, the compiler does not know the address of these items in other classes. Hence, the class, source file and bytecode file must share the same name to create a 'symbolic reference'.

In addition to class variables, instance variables and method definitions, each class in Java has a runtime constant pool that contains read-only constants used in the class. This includes literal constants, symbolic references to methods and their classes, types and more.

Each pool entry has an index. Each stack frame includes a pointer to the constant pool of its class in its space for 'other items'.

6.4.3 Static Method Calls

Static methods are called using the `invokestatic <2_byte_index>` instruction. The operand is an index in the runtime constant pool of the class of the current method. The indexed constant pool entry is a symbolic reference to the requested class and method.

The JVM uses this information to find the address of the class, the size needed for the new frame, and the address of the bytecode for the method.

6.4.4 Method Returns

The return instruction for void methods is `return`. This has no effect on the operand stack of the caller.

For methods that return a result, the instruction is of the form `ireturn`. This pops the result from the operand stack of the current frame and pushes it to the operand stack of the caller.

After either of these return instructions, the current frame is discarded and execution continues on the frame of the caller, resuming from its PC.

6.5 Java Disassembler

The Java 'disassembler' (`javap`) converts Java bytecode to mnemonics with the `-c` option. For example, `MyClass.class` can be viewed with `javap -c MyClass`. Jump instruction operands in

disassembler output show the destination instruction address rather than the offset used in the bytecode.

7 Complexity of Algorithms

7.1 Algorithm Design and Analysis

An algorithm is an effective method for solving a problem expressed as a finite sequence of instructions. Often, multiple algorithms exist for the same task. There are many criteria for choosing a suitable algorithm for a particular task, including efficiency, simplicity, clarity, elegance and proof. It is also necessary to consider whether the algorithm is always correct, always terminates or whether the problem can even be solved with an algorithm.

In general, efficiency is achieved by finding a balance between many conflicting parameters, such as run time, response time, memory usage, bandwidth usage and power consumption. One very important issue is understanding how problems and algorithms grow in complexity. Although a solution may work very well for small problems, it is important to know how its performance may be affected if the problem becomes larger.

It may be possible to improve code so that it runs quickly or uses less memory, but it may be the case that an algorithm will inherently perform worse as the amount of data increases. It may even be the case that the problem itself is inherently difficult and becomes more difficult as the size of the problem grows. Sometimes the problem can be modified so that it still satisfies the overall requirements, but behaves in a more manageable way.

Algorithm efficiency can be considered in terms of time and space.

Space (or bandwidth) complexity is not considered in the same way as computational complexity, since memory and bandwidth are limited by practical constraints. The bandwidth of a system can often be increased if necessary.

Instead, computational complexity is usually considered in terms of 'time complexity' — a measure of how much computation is involved in solving a problem. It is not necessary to know quantitatively exactly how many times harder a problem will grow. Often a qualitative indication of complexity will suffice.

7.2 Time Complexity

There are two ways to determine the run time of an algorithm.

The first method is to measure empirically. This involves benchmarking the algorithm using a representative set of inputs. This does not measure the complexity of the algorithm, rather the complexity of an implementation of the algorithm on a particular piece of hardware.

The second method is to analyse theoretically the worst case scenario, by identifying the operations of the algorithm — its time complexity.

Time complexity is a measure of the number of operations that an algorithm must execute in terms of the size of the input or problem. Since it is the algorithm that is analysed, and not the implementation, time complexity is analysed using pseudocode. The time complexity $\mathrm{T}(n)$ of an algorithm is a function of the size n of its input.

The following algorithm has time complexity $\mathrm{T}(n) = 2n^2$. It is quadratic because it has two nested loops that go through every element. It has a coefficient of 2 because there are two operations in the nested statement.

Matrix-vector multiplication of size n.

```
for i=0...n-1:
    for j=0...n-1:
        x[i] = x[i] + A[i][j] * b[j]
```

7.3 Complexity Class and Big-O Notation

Usually, it is not necessary to know the exact time complexity. It is sufficient to know the complexity class, which ignores constant factors or overheads, and expressions of lower orders. This focusses on performance for large n.

Class complexity is expressed using 'big-O notation'. For example, $\mathrm{O}\left(n^2\right)$ is "of the order of n^2".

Corresponding time complexities and complexity classes.

Time complexity	Complexity class
$\mathrm{T}(n) = n$	$\mathrm{O}(n)$
$\mathrm{T}(n) = n + 2$	$\mathrm{O}(n)$
$\mathrm{T}(n) = 2n^2$	$\mathrm{O}\left(n^2\right)$
$\mathrm{T}(n) = 10n^3$	$\mathrm{O}\left(n^3\right)$
$\mathrm{T}(n) = 5\,(n + 2)$	$\mathrm{O}(n)$
$\mathrm{T}(n) = 1000$	$\mathrm{O}(1)$
$\mathrm{T}(n) = n^2 + n + 1$	$\mathrm{O}\left(n^2\right)$

To determine complexity class, it is often sufficient to count the number of loops and the number of times they are executed.

Polynomial classes are described as 'tractable'. Exponential classes are described as 'intractable' — they can execute with small amounts of data, but cannot with large amounts.

The binary search algorithm (on a sorted array) has class $\mathrm{O}(\log_2 n)$ because the size of the loop is halved on each iteration.

Common complexity classes.

Complexity class	Name
$O(1)$	Constant
$O(\log_2 n)$	Logarithmic
$O(n)$	Linear
$O(n\log_2 n)$	Log Linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential

Binary search algorithm.

```
left = 0; right = n-1
while left < right:
    mid = (left + right) / 2
    if x[mid] < v:
        left = mid + 1
    else:
        right = mid
if x[left] == v:
    return left
else:
    return -1
```

The total algorithm complexity is determined by the complexities of its components. Sequential algorithm phases are summed, and the maximum term is considered. Function or method calls are multiplied.

For example, this following altered matrix-vector multiplication algorithm has two sequential phases and a total complexity of $O(n) + O(n^2) \Rightarrow O(n^2)$.

Initialisation and matrix-vector multiplication of size n.

```
for i=0...n-1:
    x[i] = 0
for i=0...n-1:
    for j=0...n-1:
        x[i] = x[i] + A[i][j] * b[j]
```

The following iterative binary search algorithm uses a call to a method of logarithmic complexity within a fixed loop. It has complexity $O(n) \times O(\log_2 n) \Rightarrow O(n\log_2 n)$.

7.4 Algorithm and Problem Complexity

Algorithm complexity is the worst-case run time of an algorithm. This an upper bound and is the most informative complexity.

Iterative binary search.

```
for i=0...n-1:
    binary_search(x, v[i])
```

A problem has a complexity class of O(f(n)) if there exists an algorithm of complexity class O(f(n)) to solve it. Sometimes lower bounds are considered.

Problems solvable in polynomial time (PTIME or P) are assumed to be tractable. If a solution can be guessed and then checked in polynomial time, the problem is solvable in non-deterministic polynomial time (NP).

9 Introduction to Operating Systems

9.1 Early Computers

Early computers used magnetic tape decks for storage and used paper tape for input and output. Each program was keyed in by hand to control the required system function. There was a distinct lack of monitors. The 'operators' controlled the loading of magnetic tapes, fed in cards and paper tape, and maintained the system. This 'operation' was tedious, expensive and error-prone.

As computers became faster and more powerful, the role of the operators became a limiting factor. An automated system was required in order to free the operators from mundane tasks. Such a system is known as an 'operating system'.

9.2 Operating System (OS)

An operating system (OS) is a layer of system software that acts as an intermediary between a computer user and the underlying hardware. The main function of an OS is to dynamically allocate the shared system resources to executing programs.

From the perspective of a user, the OS is designed for ease of use. It provides a convenient interface for executing user programs, and hides the complexity of the hardware implementation.

From the perspective of the system, the OS is intimately involved with the hardware and acts as a resource manager for CPU time, memory, file storage, I/O devices etc. Requests for these resources may be numerous and conflicting. It also acts as a control system between I/O devices and user programs. It manages execution to prevent errors and improper use, such as one program attempting to view the memory of another.

9.3 Components of a Computer System

The main components of a computer system are the

- hardware (the basic computing resources),

- operating system (controls and coordinates use of hardware),
- application programs (use the system resources to solve computing problems), and
- users (people, machines and other computers).

Services provided by the OS include

- program development,
- program execution,
- access to I/O devices,
- access to files,
- system access,
- error detection and response, and
- accounting.

The most frequently used functions of the OS are stored in its 'kernel'.

9.4 Evolution of Operating Systems

9.4.1 Serial Processing

Serial processing was used in the earliest computers. This relied on human operators rather than an OS. Users had access to the computer in 'series'.

One problem with serial processing concerned scheduling. Most installations used a sign-up sheet for the reservation of computer time. Mistakes in the estimations of time allocations would result in wasted computer time. A considerable amount of time was also spent simply setting up the program to run.

9.4.2 Simple Batch Systems

Since early computers were very expensive, it was important to maximise processor utilisation. In simple batch systems, the user no longer had direct access to the processor. Computer operators would batch jobs together and place them on an input device. The loading of programs and management of jobs was handled by software known as a 'monitor'.

The resident monitor always resides in memory. After it reads in a job, the processor executes instructions from the memory containing the monitor until it reaches an error or ending condition. Control then passes back to the monitor.

In this system, processor execution alternates between user programs and the monitor. As a result, main memory and processor time is given to the monitor. Despite this overhead, simple batch systems improved the utilisation of the processor.

9.4.3 Multiprogrammed Batch Systems

In uniprogrammed systems, the processor executes a program until it reaches an I/O instruction. It must then wait for the I/O instruction to complete before it continues.

In multiprogrammed systems, when one job must wait for I/O to complete, the processor can switch to another job.

9.4.4 Time Sharing Systems

Time sharing systems are used to handle multiple interactive jobs. Multiple users access the system simultaneously through terminals. The OS interleaves the execution of each user program in short bursts or quanta of computation.

Comparison of multiprogrammed batch and time sharing systems.

Operating system	Principal objective	Source of directives
Multiprogrammed batch	Maximise processor use	Commands provided with job
Time sharing	Minimise response time	Commands entered at terminal

9.5 Elements of an Operating System

9.5.1 Firmware

A small 'bootstrap' program is loaded when a computer is started. This program is stored in read-only memory (ROM) or electrically erasable programmable read-only memory (EEPROM) so that it cannot easily be altered, either accidentally or maliciously. The program is, therefore, known as 'firmware'.

The bootstrap program initialises all CPU registers, device controllers and memory contents, and loads the OS kernel into memory. Once the firmware is loaded, some services are provided outside the kernel. These are loaded at boot time to become system processes or system 'daemons', and run the entire time the kernel is running. On Unix systems, the first system process is 'init'. Once all these processes are loaded, the OS waits for events to occur.

Events such as a user clicking a mouse button or a program attempting to access a file are known as 'interrupts'. Hardware may trigger an interrupt by sending a signal to the CPU. Software triggers an interrupt by executing a special operation known as a 'system call'.

The CPU reacts to an interrupt by suspending its current execution, immediately transferring execution to a fixed address (usually the starting address of the service routine for the interrupt), executes the interrupt service routine, and finally resumes execution where it was originally interrupted.

9.5.2 Memory

Memory is an array of bytes, in which each byte is addressable. ROM cannot be modified. EEPROM can be modified, but only infrequently. The CPU requires memory that can be read and written. It uses dynamic RAM (DRAM) and registers.

Main memory is erased when a machine powers off. Secondary storage is used as a more permanent solution. This includes magnetic and optical disks.

Data that is used frequently is stored in cache memory for faster access. If the CPU requires data, the cache is checked first.

9.5.3 Time Sharing

The CPU can execute multiple jobs simultaneously by switching between the jobs stored in memory. Switches occur so frequently that users can interact with each program as it is running. CPU scheduling is used to decide which job is brought to memory to be executed when space is an issue. Reasonable response time is of the utmost importance.

Processes can be swapped between main memory and the disk. 'Virtual memory' allows for the execution of a process that is not entirely in memory. The process can use both virtual and physical memory as 'logical' memory. This allows users to run programs that require more than the available physical memory and frees programmers from concern over memory limitations.

9.6 Dual Mode

Operating system execution is split into user mode (mode bit of value 1) and kernel mode (mode bit of value 0). In user mode, certain areas of memory are protected from user access, and certain instructions cannot be executed. In kernel mode, protected areas of memory may be accessed, and privileged instructions may be executed.

The OS allows user programs to execute system calls by resetting the mode bit to 0. Once the kernel has executed the requested system call, the mode bit is set to 1. This prevents crashes in user mode from affecting the kernel.

The MS-DOS architecture has no mode bit. Hence, user programs were able to wipe the entire OS, write to devices without permission, execute illegal instructions and access the memory of other users. With dual mode, hardware can detect errors that violate modes and handle them with the help of the OS.

When a mode violation is detected, the OS must terminate the program, give an error message and produce memory dumps by writing to a file.

9.7 System Calls

System calls provide OS services to the user via an application programming interface (API). The system call APIs are typically written in C or C++, though some are written in assembly language.

The system call for reading data from one file and writing it to another is `cp <file1> <file2>`. This uses the following services.

- Open `file1`
- Handle possible error
- Create `file2`
- Start read and write
- Handle possible errors
- Close `file1` and `file2`

The services are not accessed directly. They are accessed via an API.

Examples of Windows and Unix system call APIs.

Type of call	Windows	Unix
Process Control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File Manipulation	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device manipulation	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	`write()`
Information maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communication	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shmget()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

System call APIs specify a set of functions that are available to an application programmer. They describe the parameters to pass to each function, and the values that are returned. The programmer accesses the API via a library provided by the OS.

A Java API exists for programs that run on the JVM. The JVM itself uses the system calls of its particular platform.

System call APIs are used for program portability. A program using the API can be compiled and run on any system that supports the API. Invoking system calls directly is usually more difficult. Using a system call API is similar to implementing an interface.

10 Computer Architecture and Operating System Structures

10.1 Storage Structure and Memory Hierarchy

A CPU can load instructions only from memory. Main memory is rewritable RAM — commonly implemented with DRAM semiconductor technology.

Memory is an array of bytes. Each byte has its own address. Bytes are moved between memory and CPU registers with load and store instructions. The CPU automatically loads instructions from main memory for execution.

The fetch instruction loads an instruction from memory. When the instruction is decoded, operands may also be loaded from memory. After the instruction is executed, the result is stored to memory. The memory unit only sees a stream of addresses.

Ideally, both program instructions and data would reside in the main memory. However, this is difficult as main memory is small and volatile (erased after loss of power). The solution is to store programs and data in secondary storage. When programs are run, their instructions are loaded from secondary storage to main memory.

There are many forms of storage available. They differ in speed, cost, size and volatility.

Typical properties of storage types.

Level	Name	Size	Access time / ns	Bandwidth / $\mathrm{MiB\,s^{-1}}$	Managed by
1	Registers	$< 1\,\mathrm{KiB}$	0.25 to 0.5	20000 to 100000	Compiler
2	Cache	$< 16\,\mathrm{MiB}$	0.5 to 25	5000 to 10000	Hardware
3	Main memory	$< 64\,\mathrm{GiB}$	80 to 250	1000 to 5000	OS
4	Solid-state disk	$< 1\,\mathrm{TiB}$	25000 to 50000	500	OS
5	Magnetic disk	$< 10\,\mathrm{TiB}$	5000000	20 to 150	OS

10.2 I/O Structure

A large portion of OS code is dedicated to the management of I/O. Without I/O, computers would be useless.

An I/O device interacts with the system via a 'device controller' connected through a common bus to the CPU. A small computer system interface (SCSI) controller is a piece of hardware that allows an SCSI storage device to communicate with the operating system via an SCSI bus. A device controller maintains some local buffer storage and a set of special-purpose registers. The device controller moves data between the peripheral device it controls and its local buffer storage.

An OS has a device driver for each device controller. These are typically downloaded. The device driver is able to interact with its corresponding device controller and provides the OS with a uniform interface to the device.

10.3 I/O Mechanisms

There are three types of I/O mechanism.

1. Programmed I/O (also known as 'polling')
2. Interrupt-driven I/O
3. Direct memory access (DMA)

10.3.1 Programmed I/O

Programmed I/O requires the CPU to 'poll' the status of the controller by repeatedly checking its status until it is ready to accept an I/O request. It then continues to check the status of the controller until the I/O operation is complete.

This mechanism is very fast due to the direct involvement of the CPU. However, this also prevents the CPU from working on other tasks that may be more important.

10.3.2 Interrupt-Driven I/O

Interrupt driven I/O works as follows.

1. The CPU initiates I/O through the device driver.
2. The CPU checks for interrupts between executing other instructions.
3. The device controller begins the I/O operation.
4. When input is ready, output is complete or there is an error, the device controller sends an interrupt signal to the CPU.
5. When the CPU detects the interrupt, control is transferred to the interrupt handler, which will process the data before returning from the interrupt.
6. The CPU resumes the interrupted task.

Interrupt driven I/O is suitable for transferring small amounts of data. The CPU is less involved in the I/O operation and is free to execute other instructions. However, this reduces the speed of the I/O operation.

10.3.3 Direct Memory Access (DMA)

The DMA controller transfers data directly between the I/O device and memory. This requires additional architecture for the device, including buffers, pointers and counters. The device controller transfers entire blocks of data to and from its own buffer storage and memory. There is no direct involvement of the CPU during transfer.

1. The CPU programs the DMA controller.

2. The DMA controller requests transfer to memory.
3. The disk controller transfers the data to memory.
4. The disk controller checks with the DMA controller that there is no more data to transfer.
5. The DMA controller sends an interrupt signal to the CPU when transfer is complete.

Only a single interrupt is generated. This is much better than the one interrupt per byte generated by low-speed devices. The CPU is completely free to execute other instructions. However, both the DMA controller and the CPU share the bus that connects different hardware components. Thus, the CPU will experience slowdown during DMA transfer as it must wait for access to the bus.

10.4 General Computer System Architectures

10.4.1 Multiprocessor Sytems

Multiprocessor systems have multiple processors in parallel or multiple cores. The processors share the bus, clock, memory and peripherals. This increases throughput, although this does not increase linearly with the number of processors. There is also an economy of scale, as additional processors share the same peripherals, storage and power. Reliability is also increased, as a failed processor will not stop the entire system. This is known as 'graceful degradation'.

Asymmetric multiprocessor systems follow a boss-worker relationship. Each processor is assigned a specific task.

Symmetric multiprocessor systems follow a peer-to-peer relationship. Each processor is utilised and has an equal chance of being used.

10.4.2 Non-Uniform Memory Access (NUMA)

In non-uniform memory access (NUMA) systems, multiple processors are interconnected and each processor has its own memory. Each processor can access the other memories through their corresponding processors, but this is slower than direct access to its own memory.

10.4.3 Multicore Systems

Multicore systems have multiple cores on a single chip. This is useful because on-chip communication is faster and consumes less power. This is used in blade servers to minimise physical space and power consumption.

10.4.4 Clustered Systems

Clustered systems incorporate multiple interconnected computers with a shared network storage. This provides a high availability of computers and processors, which is useful if one computer fails. It also facilitates standby or backup servers, high-performance computing and parallelisation of applications.

10.5 Operating System Services

Operating system installations typically include

- system programs (program loader, command interpreter),
- language processors (C compiler, assembler, linker),
- utilities (text editor, terminal emulator), and
- subroutine libraries (standard C library, JVM).

10.6 System Calls / OS Relationship

System calls are APIs for the services provided by the OS. They are typically written in high-level languages such as C or C++.

Typically, there is a number associated with each system call. The system call interface maintains a table indexed according to these numbers. The system call interface invokes the requested system call in the OS kernel and returns the status of the system call and any return values.

10.7 OS Design and Implementation

The internal structures of operating systems can vary widely. The goals and specifications of an OS can be affected by hardware and the type of system. User goals are that the OS should be convenient to use, easy to learn, reliable, safe and fast. System goals are that the OS should be easy to design, implement and maintain, as well as flexible, reliable, error-free and efficient.

Early operating systems were produced in assembly language, then in system programming languages such as Algol and PL/I. Nowadays they are written in C and C++.

Modern operating systems actually use a combination of different languages. The lowest levels may be written in assembly, the main body in C, and system programs in C, C++, PERL, Python and shell scripts. Operating systems written in higher-level languages are easier to port to other hardware, but they are also slower. Emulation allows an OS to run on non-native hardware.

10.8 Operating System Structure

A general-purpose OS is a very large program. There are various ways to structure one.

- Simple structure (e.g. MS-DOS)
- More complex or non-simple structure (e.g. Unix)
- Layered approach (an abstraction)
- Microkernel structure (e.g. Mach OS)
- Modular approach
- Hybrid approach

10.8.1 Simple Structure

MS-DOS is an OS with a simple structure. It was written to provide the most functionality in the least space. It is not divided into modules. Although it has some structure, its interfaces and levels of functionality are not well separated. This is dangerous, as application programs have the ability to access the lowest levels of the system.

10.8.2 More Complex or Non-Simple Structure

The Unix OS has a more complex structure. The Unix OS is limited by hardware functionality, has a limited structure, and consists of two separate parts. These are the system programs and the kernel.

The kernel consists of everything below the system call interface and above the physical hardware. It provides the file system, CPU scheduling, memory management and other OS functions. This is a large number of functions for one level, and makes the operating kernel difficult to debug, since the kernel services rely on each other. It is also dangerous since there is a lot of functionality running in kernel mode.

10.8.3 Layered Approach

In the layered approach, the OS is divided into a number of layers or levels. Each layer is built on top of a lower layer. The lowest layer (layer 0) is the hardware. The highest layer (layer N) is the user interface.

Using modularity, functions are split into layers such that the functions of one layer call upon only the functions of the layer directly beneath. While this is more structured and less dangerous, it prevents higher layers from communicating with lower levels.

10.8.4 Microkernel Structure

Mach is an example of a microkernel OS. Microkernel structure moves as much functionality from the kernel to the user space as possible. Communication between user modules takes place via messages passed through the kernel.

Microkernel structures are easier to extend and port to new architectures. They are also more reliable and secure, since less code runs in kernel mode. However, performance is lost due to added communication between the user space and the kernel space, as well as between user modules through the kernel.

10.8.5 Modular Approach

Many modern operating systems implement loadable kernel modules. This uses an object-oriented approach, in which each core component is separate, communicates with others through known interfaces, and is loaded as needed within the kernel. This is similar to the layered approach, but is more flexible. Linux and Solaris use the modular approach.

10.8.6 Hybrid Approach

Most modern operating systems do not use one pure model. Hybrid operating systems combine multiple approaches to address performance, security and usability. Linux and Solaris are actually monolithic (the OS works entirely in kernel space, similar to more complex or non-simple structures), but use modularity for the dynamic loading of functionality.

10.9 User Classes and Interfaces

Almost all operating systems have a user interface (UI). This interface can take several forms.

- Command-line interface (CLI)
- Batch interface (commands entered into files that are executed)
- Graphical user interface (GUI)

Any UI requires a software link to the hardware, which may be buried under other software.

Types of user:

- Programmers
  - System programmers (creators of operating systems, compilers, device drivers etc)
  - Application programmers
- Administrators (concerned with provision, operation and management of computing facilities)
- End-users (who apply software to some problem area)

Some end-users may be unaware that they are interacting with a computer, whereas others may have a substantial understanding of computers.

10.9.1 System Call Interface

All interaction with hardware must go through system calls. The OS provides a layer of subroutines known as an API.

10.9.2 Command-Line Interface

Most operating systems provide an interactive terminal into which commands may be entered. This can be used to imitate programs or perform housekeeping control routines on the system. Unix provides shell programs.

10.9.3 Job Control Language Interface

Job control language defines requirements for work submitted to a batch system. This is used in database administration.

10.9.4 Graphical User Interface (GUI)

Interaction proceeds via windows and usually a mouse-driven environment. There are desktops, icons and GUI APIs.

From the programmer perspective, this is slightly more complex than shell scripts, but can be more rewarding. Event-driven programming is used so that the interface is responsive to user actions.

From the user perspective, this can be friendlier. However, there is an increased processing load.

10.9.5 Touchscreen Interface

Sometimes computer mice are not available or not desired. Actions and selections are controlled by gestures. There may be a virtual keyboard for text entry.

11 Process Management

11.1 Processes

A process is a program in execution. A program is a passive entity stored on the disk, whereas a process is an active entity. A program becomes a process when its executable file is loaded into memory.

A program may be started through a GUI or CLI. One program may consist of several processes. For example, there may be one process for each user or instance.

Processes are represented by three main components.

- Executable program code (also known as the 'text segment')
- Data related to the program
- Execution context
  - process ID, group ID, user ID
  - stack pointer, program counter, CPU registers
  - file descriptors, locks, network sockets

The execution context is essential for process switching.

A process can be an execution environment for other code. For example, the JVM executes as a process that interprets loaded Java code and performs instructions on behalf of it.

11.2 Process Structure

A process is more than just the program code. It also includes current activity such as the value of the program counter and the contents of CPU registers. The process stack contains temporary data, such as function parameters, return addresses and local variables. The data segment contains global variables and a heap — memory allocated dynamically at runtime.

The layout of the memory of a process is known as a 'process image'. A process image is divided into segments.

- Stack segment
  - function parameters, return addresses and local variables
- Data segment
  - static variables and constants
  - memory allocated dynamically from the heap
- Text segment
  - program code — shared between processes

max	Stack ↓	Stack segment
	↑ Heap	Data segment
	Data	
	Program code	Text segment
0	Kernel	Protected

The process image.

Each invoked program results in the creation of a separate process with its own image. Each image appears to cover the entire address space. In reality, these are virtual addresses mapped to physical addresses.

11.3 Process Control Block (PCB)

The information associated with each process is stored as a process control block (PCB), which is also known as a task control block (TCB). This information includes

- process state,
- program counter (PC) — address of next instruction to be executed,
- CPU registers,
- CPU scheduling information — priorities and scheduling queue pointers,
- memory-management information,
- accounting information — CPU usage, time elapsed and time limits, and
- I/O status information — I/O devices allocated to the processes and list of open files.

A process with multiple threads of execution has instructions at multiple addresses executing at once. This requires multiple program counters. These additional thread details are stored in the PCB.

11.4 Process States

As a process is executed, it changes state. Possible states for a process are

- new — the process is being created,

- ready — the process is waiting to be assigned to a processor,
- running — the process is executing instructions,
- waiting (blocked) — the process is waiting for an event to occur, and
- terminated — the process has finished execution.

When a process is admitted to the system, its state changes from 'new' to 'ready'. When it is dispatched to a CPU, its state changes to 'running'. From there, it may time out, in which case its state reverts to 'ready', or it may need to wait for an event to occur, in which case its state changes to 'blocked'. Alternatively, it may be released from execution, in which case its state changes to 'terminated'. A process in the 'blocked' state moves to the 'ready' state once the event it is waiting for occurs.

11.5 Process Scheduling

Process scheduling is used to maximise CPU usage and minimise response time by quickly switching processes onto the CPU for time sharing. A process will 'give up' the CPU if it sends an I/O request or after a certain period of time as elapsed. When a process gives up the CPU, it is added to the ready queue or a device queue. The process scheduler selects available processes in the ready queue for the next execution.

The OS maintains various scheduling queues.

- Job queue — all processes in the entire system
- Ready queue — all processes in main memory ready and waiting to execute
- Device queues — all processes waiting for an I/O device

Processes migrate between queues. Queuing diagrams are used to represent flows of processes and resources.

11.5.1 Types of Scheduler

A short-term scheduler (or CPU scheduler) selects which process should be executed next and allocates a CPU. Sometimes this is the only scheduler in a system. It is invoked frequently and must be fast.

A long-term scheduler (or job scheduler) selects which processes should be brought into the ready queue. This is invoked infrequently and may be slow. The long-term scheduler controls the degree of multiprogramming.

11.5.2 Types of Process

An I/O-bound process spends more time performing I/O than computations. It uses many short CPU bursts.

A CPU-bound process spends more time performing computations. It uses a few long CPU bursts.

The long-term scheduler attempts to schedule a good process mix.

11.5.3 Context Switching

When the CPU switches to another process, the system saves the state of the old process and loads the saved state of the new process via a context switch. The context of a process is represented by its PCB.

The time required to perform a context switch is pure overhead; the system does no useful work while switching. More complex operating systems and process control blocks require more time for a context switch. The time required is dependent upon hardware support. Some hardware provides multiple sets of registers per CPU. This allows multiple contexts to be loaded simultaneously.

11.5.4 Dispatch Events

Context switches (dispatch events) are triggered by clock interrupts. These occur after a set time interval, typically between 3 ms and 10 ms. The execution of processes is interrupted and control returns to the OS. The current process is moved to the ready queue. The frequency of such an interrupt is an important system parameter used to balance the overhead of context switching and the responsiveness of the system.

Context switches are also triggered by I/O interrupts. When an I/O event occurs and data has been loaded into memory, the current process is interrupted and all blocked processes waiting for the I/O event are moved to the ready queue. The dispatcher must decide whether to continue execution of the interrupted process.

Memory faults or page faults also trigger context switches. These occur when an executing process refers to a virtual memory address that has not been allocated a physical memory location (the data requested is still on the disk). The current process is interrupted and an I/O request for the data is issued. The current process is moved to the blocked state and execution switches to another process from the ready queue. When the I/O has completed, the blocked process returns to the ready queue.

11.5.5 Dispatcher

A dispatcher is an OS function that allocates processes to the CPU and switches the CPU between processes. The weakness of a dispatcher with a single device queue is that all processes waiting for an event must be transferred from the device queue to the ready queue when an event occurs. A dispatcher with multiple queues for different device events moves only the processes waiting for that particular event to the ready queue.

11.5.6 Virtualisation and Swapping

A computer system may appear to allow an unlimited number of processes to occur concurrently. Not all of these processes can fit into physical memory. It is useful, therefore, to achieve medium-term scheduling and reduce the degree of multiprogramming by removing a process from memory.

Moving a process between memory and storage is known as 'swapping'. This requires two additional process states: 'ready-suspended' and 'blocked-suspended'.

A process in the 'ready-suspended' or 'blocked-suspended' state is activated to the 'ready' or 'blocked' state when it is swapped into memory. It is suspended when it is swapped out of memory. A process in the 'blocked-suspended' state is moved to the 'ready-suspended' state when the event for which it is waiting has occurred. A process in the 'running' state may be suspended.

11.6 Process Creation

The system must provide mechanisms for process creation and process termination.

Processes are created at system boot, when an existing process spawns a child process, when the user requests the creation of a process, or when a batch systems executes its next job. When a parent process creates a child process, a tree of processes is formed. A process is identified and managed via its process identifier (PID).

When a process is created, it is assigned a unique PID, memory is allocated for its process image, its PCB is initialised, and it is added to the ready queue.

11.6.1 Parent and Child Processes

A child process may share all the resources of its parent, a subset of those resources or none of them. A parent and child process may execute concurrently, or the parent may wait for its children to terminate.

A child process is a duplicate of the address space of its parent. The child process may load a new program into its address space.

11.6.2 Process Creation in Unix

A process creates new processes with the kernel system calls `fork()` and `exec()`.

- A new slot is allocated in the process table.
- A unique PID is assigned to the child process.
- The process image of the parent is copied, apart from the shared memory areas.
- The child process owns the same open files as the parent.
- The child process is added to the ready queue.

When `fork()` is called by the parent process, execution switches to kernel mode and the OS creates a copy of the parent process. This includes the program code in the text segment of the process image. Thus, the `fork()` call is also copied. The parent process continues execution at the return from the `fork()` call. The child process begins execution at the return from the `fork()` call.

In the parent process, the `fork()` call returns the PID of the child process. In the child process, the `fork()` call returns zero. This is used by the remainder of the program to identify whether it is running as a parent or a child. The program may perform different instructions in either of these cases.

For example, the child process may execute the `exec()` system call, which takes a program name as a parameter. This system call replaces the content of the child process image with a process image of the specified program.

The `exec()` system call has a number of variations with different suffixes that describe what parameters are passed to the new process image. These variations are `execl()`, `execle()`, `execlp()`, `execv()`, `execve()` and `execvp()`.

- `e` — An array of pointers of environment variables is passed.
- `l` — Command line arguments are passed individually.
- `p` — The `PATH` environment variable is used to find the program to be executed.
- `v` — Command line arguments are passed as an array of pointers.

11.7 Process Waiting

A parent may use the system call `waitpid()` to wait for the termination of a child process.

11.8 Process Termination

There are a number of ways in which a process may be terminated.

- Normal termination (voluntary) or error termination (voluntary)
  - the process reaches regular completion, with or without an error code
- Abnormal termination (involuntary)
  - OS or user intervenes with kill signal
  - process attempts to access forbidden memory locations
  - process times out
  - process encounters an I/O error
  - stack overflow
  - parent process is terminated

Some operating systems do not allow a child process to exist if its parent has been terminated. This is known as 'cascading termination'.

A 'zombie process' is a process that has terminated but still has an entry in the process table. This occurs when a child process terminates. It allows a parent process to read the exit codes of its children.

11.9 Execution of the Operating System

An OS may be designed so that it executes

- separately from processes (non-process kernel),
- as part of the user process images (with mode switching), or
- as a set of processes (microkernel, with mode and context switching).

11.9.1 Non-Process Kernel OS

The kernel acts as a monitor for user processes. All processes are user processes. There is no concurrent execution of user processes and the kernel; the kernel may execute only during interruptions.

The OS code is placed in a reserved memory region and executes in privileged mode. It has its own system stack.

11.9.2 OS Execution within User Processes

The user address space includes kernel functions. System functions may be called from within a process by switching to kernel mode. No context switching is required as the system functions execute within the same process. The dispatcher executes context switches outside of processes.

11.9.3 Process-Based OS

Kernel functions run in kernel mode as separate processes. The dispatcher handles context switching separately. This allows concurrency within the kernel, with kernel processes scheduled together with user processes. This is useful in multiprocessor environments, as kernel functionality may be distributed amongst CPU cores.

12 Process Scheduling

12.1 Basic Concepts

12.1.1 CPU and I/O Burst Cycle

Process scheduling is the process of selecting the next process in the ready queue to be executed by the CPU. Scheduling decisions take place when events interrupt the execution of a process. Such events include clock interrupts, I/O interrupts, system calls and signals.

Process execution is a cycle of CPU execution and I/O waiting. Processes on a batch system are CPU-bound, whereas processes on an interactive system are I/O-bound with very short CPU bursts. When a process is in an I/O burst, the CPU becomes idle and the short-term scheduler must select

another process from the ready queue. The ready queue is not necessarily a FIFO queue; it may be a priority queue, tree or unordered linked list.

12.1.2 Dispatcher

The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This involves switching context, switching to user mode and jumping to the correct location stored in the PCB of the process in order to resume execution. The dispatcher must be as fast as possible since it is invoked at every process switch. The time taken by the dispatcher to stop one process and start another is known as 'dispatch latency'.

12.1.3 Levels of Scheduling

The long-term scheduler is responsible for admitting new processes to the ready queue. The short-term scheduler is responsible for selecting the next process to be given control of the CPU. The mid-term scheduler is responsible for moving processes between suspended and non-suspended states.

A non-preemptive scheduler allows an executing process to continue execution until it terminates, releases the CPU voluntarily (cooperative scheduling) or is blocked due to an event such as an I/O interrupt. A preemptive scheduler may stop an executing process due to a clock interrupt (time slice expired) or when a process with a higher priority becomes ready.

12.2 Scheduling Criteria

Scheduling algorithms are designed to maximise or minimise different parameters. OS parameters to be maximised include

- CPU utilisation, and
- throughput — the number of processes that complete execution per unit time.

User concerns to be minimised include

- turnaround time — the total time to execute a process to completion,
- waiting time — the time a process spends in the ready queue, and
- response time — the time between the submission of a new process and its first CPU burst (also known as latency).

12.3 Scheduling Algorithms

12.3.1 First Come First Serve (FCFS)

First come first serve (FCFS) is the simplest scheduling algorithm. Processes are given control of the CPU in the order that they request it. This is implemented with a FIFO ready queue. When a process enters the ready queue, its PCB is pushed to the tail. When the CPU becomes free, it is allocated to the process whose PCB is popped from the head of the queue.

Using this algorithm, the average process waiting time can vary greatly. If a long process arrives before a short process, the waiting time for the second process will be long. If the short process arrives first, the second process will have a short waiting time.

12.3.2 Shortest Job First (SJF)

Using the shortest job first (SJF) scheduling algorithm, each process is associated with an estimate of the length of its next CPU bust. When the CPU becomes available, it is assigned to the process that has the shortest estimated next CPU burst. If two processes are estimated to have next CPU bursts of the same length, FCFS is used to choose between them. A more accurate name for this algorithm would be 'shortest next CPU burst first'.

The SJF scheduling algorithm can be proven to be optimal. It gives the minimum average waiting time for a given set of processes. This works because scheduling a short process before a long process decreases the waiting time of the short process more than it increases the waiting time of the long process. Thus, the average waiting time decreases.

However, the algorithm is difficult to implement since there is no way to know the exact length of the next CPU burst. Instead, the length of the next CPU burst of a process is estimated by an exponential average of the measured lengths of its previous CPU bursts. The predicted length τ_{n+1} of the next $((n+1)\text{th})$ CPU burst is given by the following iterative formula in terms of the measured length t_n of the previous (nth) CPU burst and a weighting factor α.

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\,\tau_n$$

Commonly, the weighting factor α is set to 0.5. Thus, if the predicted and actual lengths of the nth CPU burst are 10 ms and 6 ms, respectively, the predicted length of the $(n+1)$th CPU burst would be 8 ms.

12.3.3 Shortest Remaining Time First (SRTF)

SJF can be either preemptive or non-preemptive. If the length of the next CPU burst of a newly submitted process is shorter than the remaining CPU burst of the currently executing process, the preemptive SJF would switch the two processes to minimise the average waiting time, whereas the non-preemptive SJF would allow the current process to finish its CPU burst.

The preemptive SJF scheduling algorithm is known as the 'shortest remaining time first' (SRTF) scheduling algorithm.

12.3.4 Priority Scheduling

The SJF scheduling algorithm is a special case of the general priority scheduling algorithm. A priority is associated with each process and the CPU is allocated to the process with the highest priority.

Priorities are indicated by a fixed range of numbers. Lower numbers are used for higher priorities. Processes with equal priorities are scheduled using FCFS.

Priorities can be defined by internal criteria, such as time limits, memory requirements and number of open files, or external criteria, such as process importance, funds paid for computation or political factors.

Preemptive priority scheduling will switch processes if a newly submitted process has a higher priority than the current process. Non-preemptive priority scheduling would simply place that process at the head of the priority queue.

Priority scheduling can result in the indefinite blocking or starvation of low-priority processes if the system receives a steady stream of processes with higher priority. These processes may eventually run after a very long waiting time, or may be lost if the system crashes before that point. This can be solved by increasing priority with age.

12.3.5 Round-Robin (RR)

The round-robin (RR) scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS, but uses preemption to enable the switching of processes. A circular ready queue and small 'time quantum' or 'time slice' are used. The CPU is allocated to each process in the queue for a time interval of up to one time quantum. The ready queue is treated as a FIFO queue.

The scheduler sets a timer to interrupt the CPU after one time quantum and dispatches the first process in the ready queue. If the process has a CPU burst length of less than one time quantum, the process will release the CPU voluntarily. Otherwise, the process is interrupted after one time quantum, resulting in a context switch to the second process. The first process is pushed to the tail of the ready queue.

If there are n processes in the ready queue and the time quantum is q, each process is given $\frac{1}{n}$ of the CPU time in bursts of at most q. Each process must wait no longer than $(n-1)\,q$ until its next CPU burst. For example, in a queue of five processes with a time quantum of 20 ms, each process is given up to 20 ms of CPU time every 100 ms or less, resulting in a waiting time of no more than 80 ms between CPU bursts.

The performance of the RR scheduling algorithm depends heavily on the size of the time quantum. If the time quantum is longer than the process length, the algorithm would be no different from FCFS. If the time quantum is too short, there would be a large number of expensive context switches. Ideally, the time quantum should be much larger than the context switch time. If the context switch time is n% of the time quantum, approximately n% of CPU time would be devoted to context switching. Modern systems typically have time quanta between 10 ms and 100 ms and context switches of less than 10 µs.

14 Concurrency and Synchronisation

14.1 Introduction to Concurrency

14.1.1 Motivation

A computer system typically has multiple users or user applications working independently. The OS must protect the memory of one process from interference from another process. It also interleaves the execution of processes to maximise the resource utilisation and responsiveness of the system. To do so, it must preserve the state of each process so that its execution may resume.

Concurrent execution of processes was originally intended solely for the maximisation of utilisation due to the great cost of computer systems, but nowadays it is intended to satisfy the expectations of users who wish to execute multiple tasks concurrently.

The problem is that these processes do not always work independently. They may compete for access to resources, such as devices, files and data. They may cooperate through messages, shared memory and files. They may also be distributed such that millions of users are reading and updating information at the same time. This is the case for social networking, banking and online shopping systems.

It is also necessary that applications can interleave multiple independent but potentially conflicting tasks. An application with a graphical user interface (GUI) should not freeze when it is performing a time consuming task. Similarly, a user of a banking system may not want to wait in a queue in order to perform a transaction. Nevertheless, it is still necessary to maximise the efficiency of the system.

14.1.2 Multiple Processes

OS design is concerned with the management of processes and threads.

- Multiprogramming — multiple independent processes on a single processor
- Multiprocessing — multiple processes on multiple processors
- Distributed processing — multiple processes on multiple machines

Concurrency may be employed in the context of sharing resources amongst multiple applications, sharing resources between processes in a single application, or sharing resources between processes and threads of the OS itself.

14.1.3 Key Terms

atomic operation A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instructions is guaranteed to execute as a group, or not execute at all, having no visible effect on the system state. Atomicity guarantees isolation from concurrent processes.

critical section A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.

deadlock A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.

livelock A situation in which two or more processes continuously change their states in response to changes in other processes without doing any useful work.

mutual exclusion The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.

race condition A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.

starvation A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

A critical section must behave as an atomic operation. Mutual exclusion occurs in a critical section. A race condition must be managed by mutual exclusion.

14.1.4 Principles of Concurrency

Interleaving and overlapping can be viewed as examples on concurrent processing. They both present the same problems.

In a uniprocessor system, the relative execution of processes cannot be predicted because it depends on the activities of the processes, how the OS handles interrupts and the scheduling policies of the OS.

14.1.5 Difficulties of Concurrency

Difficulties arise in the sharing of global resources, the optimal allocation of resources by the OS, and, most importantly, in the coordination of parallel and asynchronous processes so that they behave correctly. They must produce the same results if they are executed concurrently as they would if they were executed consecutively. The debugging of concurrent programming errors is difficult because observed behaviour is non-deterministic (dependent upon timings of other processes on the machine). It is therefore difficult to identify and reproduce.

14.2 OS Concerns and Process Interaction

In order to manage the issues raised by concurrency, the OS must be able to keep track of various processes, allocate and deallocate resources for each active process, protect the data and physical resources of each process from interference by other processes, and ensure that the processes and outputs are independent of 'processing speed' (the timing of completion of different operations).

Relationships between concurrent processes.

Degree of awareness	Relationship	Influence of one process on the other	Potential control problems
Processes unaware of each other	Competition	• Results of one process independent of the other • Timing of processes may be affected	• Mutual exclusion • Deadlock (reusable resource) • Starvation
Processes indirectly aware of each other (e.g. shared object)	Cooperation by sharing	• Results of one process may depend on information obtained by other • Timing of processes may be affected	• Mutual exclusion • Deadlock (reusable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from other • Timing of processes may be affected	• Deadlock (consumable resource) • Starvation

14.3 Resource Competition

Concurrent processes come into conflict when they are competing for use of the same resource, such as I/O devices, memory, processor time and the clock. In the case of competing processes, three control problems must be faced.

- Mutual exclusion — to ensure correct behaviour
- Deadlock
- Starvation

A race condition occurs when multiple processes or threads read and write shared data items. The processes "race" to perform their read/write actions. The final result depends on the order of execution. The "loser" of the race is the process that performs the last update and determines the final value of the shared data item.

Race conditions can occur due to

- order of execution — whenever the state of a shared resource depends on the precise order of execution of the processes,
- scheduling — context switches at arbitrary times during execution, or
- outdated information — processes or threads operating with stale or dirty copies of memory values in registers or local variables.

14.4 Critical Section Problem

14.4.1 Critical Section

In order to avoid race conditions, it is necessary to control the concurrent execution of critical sections. This is achieved through strict serialisation, or mutual exclusion, causing the critical sections to behave as atomic operations.

An entry protocol is executed before entering a critical section. The process requests permission to enter the critical section. The process may have to wait for entry to be granted. It must communicate that it has entered the critical section.

A process is able to complete the execution of its critical section, even if it is preempted or interrupted. An exit protocol is executed after exiting a critical section. The process communicates to other processes that it has left the critical section.

It is important that the scope and length of critical sections is kept as small as possible. Large critical sections can be detrimental to the efficiency and throughput of a system.

14.4.2 Deadlock and Starvation

Enforcing mutual exclusion creates the problems of deadlocks (processes waiting forever for each other to free resources) and starvation (a process waiting forever to be granted entry to its critical

section). Implementations of mutual exclusion must account for these problems. Critical sections must be as small as possible and processes must spend as little time as possible in a critical section.

14.4.3 Requirements for Solutions to the Problem

1. Serialisation of access
    - Only one process at a time is allowed in the critical section for a resource
2. Bounded waiting (no starvation)
    - A process waiting to enter a critical section must be guaranteed entry within some defined limited waiting time
    - The scheduling algorithm must guarantee that the process is eventually scheduled
3. Progress (liveness, no deadlock)
    - A process that halts in its non-critical section must do so without interfering with other processes waiting to enter their critical section
    - Only processes currently waiting to enter their critical section are involved in the selection of the one process that may enter
    - A process remains inside its critical section for a finite time only

1. Software solutions
    - Shared lock variables (busy-waiting)
    - Polling and spinning or strict alternation
2. Hardware solutions
    - Disabling interrupts
    - Special instructions
3. Higher-level OS constructs
    - Semaphores, monitors or message passing

14.5 Software Solutions

14.5.1 Lock Variables

Critical sections must be protected by some form of 'lock'. A lock is a shared data item. Processes must acquire a lock before entering a critical section, and release the lock when exiting the critical section. A lock is also known as a 'mutex'.

A two state lock may be implemented as a shared variable. A process will wait while the lock is true, then when it is false, set the lock to true and execute its critical actions. When its critical section is complete, it will set the lock to false. However, in this implementation, the lock itself is a shared resource and a race condition may occur on it. Atomic instructions must be used for a shared lock.

14.5.2 Busy-Waiting (Polling and Spinning)

In this solution, a process continuously evaluates whether a lock has become available. The lock is represented by a data item held in shared memory. This causes the process to consume CPU cycles

without any progress. A process busy-waiting may prevent another process holding the lock from executing and completing its critical section and from releasing the lock.

14.5.3 Busy-Waiting (Strict Alternation)

This solution imposes a strict alternation between two processes; a process waits for its turn. A token is used as a shared variable. This is usually a process ID set by the previous process that indicates which process is next to enter. A process busy-waits until the token is its own process ID. When the process exits the critical section, it sets the token to the next process ID.

This solution guarantees mutual exclusion because there is no longer a race condition on the lock. However, there is still a problem of liveness and progression since multiple processes depend on the change of the token. If one of the processes is delayed in its non-critical section, it cannot enter its critical section when it is its turn, and other processes will be blocked.

14.6 Hardware Solutions

14.6.1 Disabling Interrupts

In uniprocessor systems, concurrent processes cannot be overlapped, only interleaved. Thus, disabling interrupts guarantees mutual exclusion, but significantly reduces the efficiency of execution. This approach does not work on a multiprocessor system.

14.6.2 Special Hardware Instructions

These are applicable to any number of processes on either uniprocessor or multiprocessor systems sharing a main memory. They are simple and easy to verify and can be used to support multiple critical sections. However, they use busy-waiting and may still cause starvation and deadlocks.

14.7 Abstractions

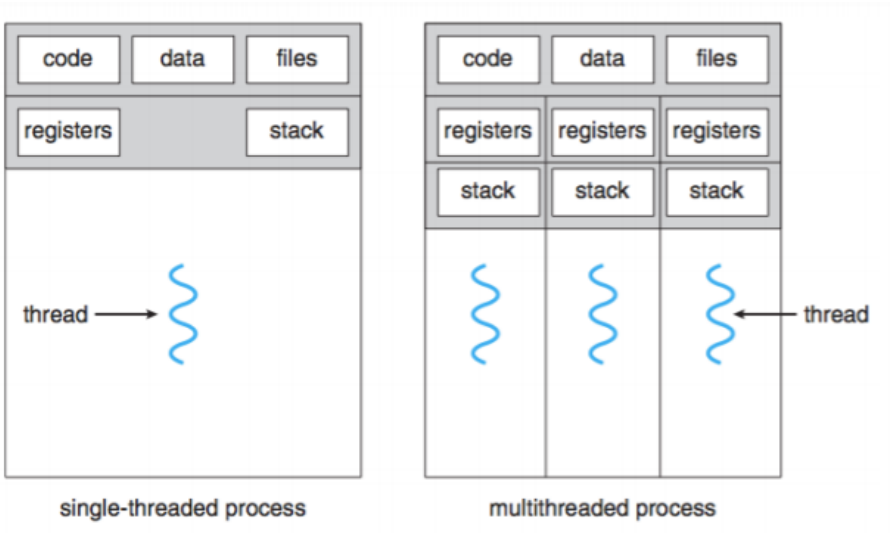
Some systems may execute critical sections regardless, then detect whether problems have occurred. If so, the changes are undone. Sometimes such problems do not occur very often, and doing a lot of work to fix the problems when they do occur is overall more efficient than doing a little work every time to avoid them. This approach is used in database systems.

In other cases, it may be acceptable to have errors caused by concurrency.

Many systems, such as databases and Java, implement their own concurrent mechanisms that work correctly and efficiently, and provide abstractions of these mechanisms to programmers. Therefore, it is often not necessary to worry about deadlocks in practice.

14.8 Parallelism and Concurrency

The process model assumes that a process is an executing program with a single thread of control. However, all modern operating systems provide features to enable a process to contain multiple



Single and multithreaded processes.

threads of control. Multithreading is the ability of an OS to support multiple concurrent paths of execution within a single process.

A thread is part of a process that contains

- a thread ID (TID),
- a program counter (PC),
- a register set, and
- a stack.

All threads within the same process share a code section and a data section (containing objects, open files and network connections). A process with multiple threads can perform more than one task at a time.

Most applications are multithreaded. Applications are suitable for threading if they need several threads of control. Examples include web browsers with separate threads for each tab, extension or native utility, as well as any GUI application, so that user events can still be handled while the main thread is performing a computation.

Multithreading is also used in systems that provide services, such as web servers and databases. When the server receives a request, a new thread is created to service the request, while the main thread continues to listen for new requests.

Threads improve responsiveness, resource sharing and economy — multithreading results in a smaller memory footprint and greater efficiency of switching between threads. Threads also improve scalability by enabling the use of parallel processing in multithreaded processors. They also reduce program complexity by breaking problems into smaller, independent tasks that can execute in parallel.

A system is parallel if it can perform more than one task simultaneously. A concurrent system supports more than one task by allowing each task to make progress. Concurrency can be achieved without parallelism.

14.9 Multicore Systems and Multithreading

Amdahl's law describes the ratio of performance of a process running on a single processor to the process running on multiple processors in terms of the proportion f of code that is parallelisable and the number N of parallel processors. This ratio is known as 'speedup'. The difference $(1 - f)$ is the proportion of the code that is inherently serial.

$$\text{speedup} = \frac{\text{time to execute on a single processor}}{\text{time to execute on } N \text{ parallel processors}} = \frac{1}{(1-f) + \frac{f}{N}}$$

Even a small amount of serial code has a noticeable impact on the overall performance. There are also overheads to parallelism that are not considered by Amdahl's law. latency is introduced due to memory access and cache load, for example.

Parallel program introduces a number of challenges. Pressure is placed on system designers to make better use of multiple cores and to write scheduling algorithms that allow parallel execution. It is difficult both to modify existing programs and to design new programs to make use of multithreading. This involves identifying tasks that can be divided in separate, concurrent and, ideally, independent tasks, balancing tasks to achieve efficiency, splitting data and identifying data dependencies. It is also difficult to test and debug parallel execution since it is non-deterministic.

14.10 Multithreading Models

Threads are used through a thread library that exists on the OS or virtual machine. Operating systems and programming languages provide APIs for creating and managing threads.

Threads may exist at either the user or kernel levels. user threads are managed above the kernel. Kernel threads are managed by the OS directly. Threads can be mapped to processes using three models.

14.10.1 Many-to-One Model

The many-to-one model maps many user-level threads to one kernel thread. Thread management is handled by the thread library in user space, so it is efficient. However, only one thread can access the kernel at a time since a system call blocks other threads. This model lacks parallelism on multicore systems and is, therefore, not widely used.

14.10.2 One-to-One Model

The one-to-one model maps each user thread to a kernel thread. This overcomes the issue of blocking, and improves concurrency and parallelism. However, for each user thread, there must be a kernel thread.

14.10.3 Many-to-Many Model

A software developer can create as many user threads as necessary in a thread pool. Corresponding kernel threads can run in parallel on a multiprocessor. If a thread blocks, the kernel can schedule another thread for execution.

14.11 Thread Interference and Memory Consistency

Threads communicate by sharing data. If multiple threads reference the same object, thread interference can occur. This happens when two functions operating on the same data interleave. This is unpredictable and difficult to debug.

Memory consistency errors occur when different threads have inconsistent views of the same data. This can be solved through synchronisation, which creates a happens-before relationship between methods and statements. This guarantees that memory writes by one statement are visible to another.

Two invocations of a synchronised method on the same object cannot be interleaved. All other threads are blocked until the synchronised thread is complete. When a synchronised method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronised method on the same object.

Synchronisation is based on intrinsic locks. These enforce exclusive access and establish happens-before relationships. Threads must acquire locks before they can do anything.