**Learning Outcomes**

|  |
| --- |
| * Demonstrate understanding of operating systems models, hierarchies, types, and standards * Show appreciation of how programs run, scheduling, and how their characteristics can be measured * Show awareness of resource utilisation, including virtual memory, memory management, communication, and virtualisation, * Understand and recognise some key developments in processors and systems in recent years |

**Summary**

|  |
| --- |
| The start-up of a system follows a series of distinct steps. This involves non-volatile memory called BIOS and sequentially moves through a hierarchy until the user is presented with the operating system.  Tasks make up the workload of the system. These are processes and threads, small subparts of the process running concurrently to achieve a shared outcome. Task scheduler run as part of the OS to ensure each task is given CPU time according to a set policy.  Performance has been increased from uniprocessor systems by adding multiple processors onto chips and utilising hyperthreading. |

**Lesson 1: Operating System Concepts**

|  |  |  |
| --- | --- | --- |
| **System Start Up**   * **Bare Metal Systems** * **BIOS** * **System Hierarchy** * **OS Kernel** |  | Operating systems can be simply defined as a collection of software operating at different levels to hide the true hardware structure from programs and users.  A **bare metal system**, without any code or software present, will crash almost immediately. At start up:   * The CPU will reset its program counter to zero and read instructions from memory * If the system only contains volatile memory, then these will be random values * The CPU will try to execute these instructions and crash * Thus, non-volatile memory content containing instructions needs to be present to function   Start up code requirements can be fulfilled through use of ROM, PROM, EPROM, or EEPROM.  This memory will contain code with start-up instructions, called the **Basic Input Output System** (BIOS):   * Initialise any important registers or components in the system * Check for existence of DRAM or SRAM * Power-on self-tests on key components to ensure expected functionality * Look for storage devices to find further instructions * Typically, this will allow a user to halt start-up to change settings   The system needs to communicate with the disk unit and understand how to access data held.   * The BIOS reads the first block of data from the disk, known as the **boot sector** * Once decoded, the BIOS can understand what the file system being used is * This is typically achieved by a small program which is executed by the BIOS   If this is successful, the involvement of the BIOS ends, and the boot sector reads code from the file system.  The structure of a system is several parts working together in a hierarchy, as visualised below:     * **Hardware** – the base system including electronic and physical components * **BIOS** – initial CPU start up code * **Boot Code** – instructions stored in the boot sector of the first device the BIOS locates * **Kernel** – overseer code that controls memory use, program starts, and low-level access * **Third-Party Drivers** – selectively installed auxiliary code to communicate with devices * **OS Utilities** – programs written by the OS vendor to provide useful capabilities * **User Applications** – limited by the capabilities of the OS, kernel, drivers, and hardware   Specifically, the kernel is responsible for:   * Controlling access to low level hardware to provide a standardised reliable route and ensure security of the system by controlling program access * Manage allocation of memory for programs, which ensures memory is allocated and released correctly and programs only access memory they have permission for * Managing allocation of CPU time to allow for fair distribution and applications to run in parallel * Managing fault conditions, meaning that it will attempt to maintain stability of the system when errors occur and ensuring failing programs don’t affect functioning ones * Controlling access to functions on a user-by-user basis, allowing different rights and permissions per user * Managing how the computer deals with IO devices needing system responses, this could include plug and play devices that may appear and disappear during system use   Together with the device drivers, the kernel achieves a concept called **abstraction**: that resources which vary from system to system appear to behave in an identical fashion despite being different models etc:     * For example, a HDD, SDD, or USB are different but may look the same to the system * This allows programs to be written without knowing the exact behaviour of low-level devices * The kernel hides the differences and provides a known point of interface |
| **Operating Systems**     * **Processes** |  | Various types of OS exist each with their own advantages. This is due to market pressures, open-source development, historical divergences in systems models, and the need for application specific systems.  A consequence is that compatibility of software and file systems across platforms remains an issue.  Two distinct categories of OS can be created, based on user interactivity:   * **Command line operating systems** – use text-based commands and interfaces, often the command line capability is referred to as the shell, eg MS-DOS and Unix * **Graphical operating systems** – use graphical user experience and are hugely popular due to their accessibility, eg Windows, MacOS, and most Linux distributions   Due to the historical evolution of operating systems, it is common for a graphical OS to be built on command line OS kernels, allowing access to the command line within the system. This is useful for:   * Administration and maintenance of software issues * Creation of shell scripts to execute a set sequence of commands   A major responsibility of the OS is to manage the running of programs on the system but as it sits on lower-level entities, such as BIOS and Boot Code, it has no responsibility for them.   * Controlling the running of programs allows them to be coordinated and run concurrently * These can be referred to as **processes** or **tasks**, a more general term * Some systems may task switch at high speeds, which gives the illusion of concurrency * This is less common with multicore processors but still occurs as the cores are limited   A further level of hierarchy exists, as processes that are running also contain **threads**: tiny programs that run concurrently as part of the same program. Each would require CPU time and memory resources.   * Threads will not be equal; some will have more work than others * For example, autosave features may do nothing most of the time but must still be present   Among the processes being run will be low consumption system tasks. These may include special purpose services, such as providing network capabilities, and are referred to as **daemons**. |
| **General Versus Specific** |  | Operating systems can be general purpose, specific, or a blend:   * General purpose examples include Windows, Linux, and MacOS. They provide reasonable capacity for general home, office, and server-based computing * Mobile examples include Apple IOS and Android. They are heavily optimised for mobile devices to provide smooth use experiences and long battery life * Real-Time Operating Systems (RTOS) are highly specialised, such as medical or automotive systems. They may lack some features and provide deterministic outcomes |

**Online**:Section 4.3-4.7, Computer Architecture and Operating Systems, University of York

**Print**:Chapter 9.1-9.7, Computer Architecture and Operating Systems, Crispin-Bailey

Chapter 8.0-8.1, Computer Systems: A Programmer’s Perspective, Bryant et al

**Lesson 2: Workload Management**

|  |  |  |
| --- | --- | --- |
| **Processes Threads Tasks**     * **Process ID** * **Thread Management** |  | Understanding of tasks can be refined by categorising and establishing what they can and cannot do:     * **Processes** – programs controlled, allocated private memory space, and given CPU time by the OS. This compartmentalised the process, improving security by preventing other processes from reading data in the private memory space * **Threads** – a process may create many threads: submodules within the process each with their own function which run concurrently to achieve a collective outcome. Each thread is given its own CPU time at frequent intervals, facilitating concurrent progress   As threads can be considered siblings within the same process, they are able to access the same memory space allocated to the process. This allows threads to share key data and work collaboratively.  To track processes, the OS uses two important concepts:   * **Process identifier** (PID) – a unique number assigned to a process when it starts * **Process control block** (PCB) – contains detailed information about the process, including state, parent processes, priority, privileges, IO related information   Together, these allow a task management function known as the **scheduler**, to manage each process according to the OS configuration.  Although threads belong to processes, they can be created and managed under certain models:     * **User-level threads** – threads within a process are scheduled with the process itself * Therefore, the kernel does not see them as separate entities * This allows custom scheduling within the boundaries of that process * **Kernel-level threads** – processes request the kernel creates threads on its behalf * Consequently, a process does not need its own coded scheduler * However, it must accept the constraints of the kernel scheduler   While both schemes can coexist, generally where kernel level threading is available it should be used. |
| **Task Scheduling**     * **Scheduling Algorithms** * **Pre-Emptive** * **Achieving Priority** |  | Allocating time slices to each task allows them all to progress with a **task-switching overhead** incurred.   * In the worst case, time taken will be longer than if executed serially * However, overall throughput may still be achieved by task-switching * Additionally, serial execution is simply not feasible with the demands of modern systems   The kernel’s **task scheduler** manages switching in line with a policy defined by the **scheduling algorithm**.  Ultimately, all these algorithms aim to ensure all processes receive an appropriate share of resources.   * The task scheduler holds a list of currently running tasks, known as the **task list** * Programs are added and removed as they start and end, making the list dynamic * The **round-robin** is a simple model, which picks one task at a time and loops around * A **context switch** occurs, **suspending** the current task and **resuming** the next * This involves copying the paused CPU state into the PCB and restoring the next task’s   This simple round-robin example is fair: each process gets an equal amount of CPU time. However, important tasks will be made to wait to complete. Thus, this example has no sense of **prioritisation** Additionally, one tasks may be intensively working while another may be idle for most of its clock cycle.  To correct for this, a task scheduler may use a **queue** instead of a list. New and suspended tasks are placed at the back of the queue, terminating tasks are removed. This allows for pre-emptive scheduling.  Without pre-emptive scheduling, tasks yield control back to the scheduler to allow the scheduler to switch task.  With **pre-emptive scheduling**, switching is triggered according to the **Repetitive Interrupt Timer** (RIT).   * In the simplest case, the timer enacts a uniform time for all processes * However, if a task has a **priority level**, the scheduler may give it a longer period   Additionally, a policy may allow processes to voluntarily give control back to the scheduler prematurely.  When the scheduler allocates fixed times, even if different for each task, it becomes **deterministic**. However, this is not the case if tasks are able to yield control voluntarily due to the inherent unpredictability.   * Serious variety in execution times is known as **jitter** * This is a problem for real time and safety critical systems   Multiple queues can be introduced to further improve the scheduling model.   * A process could be placed into a queue that best represents their current state * This could avoid unnecessary overheads by not switching to tasks waiting * An example could be a **ready queue** and an **IO wait queue** * The **IO-Polling** algorithm reads the status of devices, but is a wasteful use of time slices * **Interrupts** are signals used to indicate events have occurred in the system * This allows for a task to be suspended in the IO queue until an interrupt is received   The idea of priority is that some tasks are more important than others and need more CPU time.  The OS manages priority by default, but this can be influenced by applications or the user. Priority is often numerical, for example Linux tasks have priorities from high (-20) to low (19), with 0 being the default.   * One option is to change the length of the time slice given to a level of task * This is often difficult to implement in practice * Or time slices can be uniform but less important tasks in the queue are skipped * Alternatively, separate high, medium, and low priority queues could be used   Specialised operating systems may use **Earliest Deadline First** (EDF) priorities, where urgent tasks are given priority. Such explicit deadlines are not usually a concern in general purpose computing. |

**Online**:Section 4.8-4.9, Computer Architecture and Operating Systems, University of York

**Print**:Chapter 10.1-10.7, Computer Architecture and Operating Systems, Crispin-Bailey

Chapter 12.7-12.7.3, Computer Systems: A Programmer’s Perspective, Bryant et al

**Lesson 3: Further Workload Management**

|  |  |  |
| --- | --- | --- |
| **Advanced CPU Architecture**     * **Superscalar** * **Multicore** * **Hyperthreading** |  | Operating systems have developed to make use of the capabilities of modern processors.  **Scalar** execution is the process of executing one instruction at a time.   * It was quickly recognised as a limitation that could be overcome * This is based on the observation of code as having **instruction level parallelism** (ILP) * This means neighbouring instructions can often be executed independently   Executing multiple instructions simultaneously is known as **superscalar** execution and exploits ILP.   * There is a limit to how much ILP can be exploited in this way * Generally, a single program sequence will rarely execute 4 instructions in parallel * The average over a long sequence may be 2-3 **instructions per clock** (IPC)   Compilers will seek to maximise ILP.  Heat, power, and cost ultimately resulted in less transistors being placed on processors to maximise ILP.  Consequently, most current superscalar processors have settled into a 4-6 instruction model.  Processor manufacturers have begun to move towards using extra transistors to duplicate CPU cores to run threads in parallel, known as Thread Level Parallelism (TLP).   * Instead of one over complex processor, multiple modest and streamlined ones are used * General purpose systems will generally have 4-8 processor cores on a chip * The task scheduler can run multiple tasks on multiple processors * This provides true concurrent operation, not just the illusion of it * However, a suitable cache design is needed to avoid the bottleneck of the single bus   Scheduling and switching of threads on these individual cores are no more efficient than on a single core.   * This is partly due to the on-chip **state** of the processor core * This relates to things like contents of registers and buffers * Switching all this information on a task switch costs CPU clock cycles   **Hyperthreading** is seeks to optimise this problem. Instead of having a set of resources for each core and swapping on a task-switch, the state related circuits of the core can be duplicated for several threads.     * Multiple cores can coexist and issue instructions at the same time * They will compete over the resources available * However, this will ensure all these resources are heavily utilised * The result is each thread progresses intermixed with work of other threads * Without hyperthreading, the stalled thread would sit idle, wasting CPU time * This is also known as **simultaneous multithreading**   As previously, n number of threads do not necessarily produce n times the performance. However, in this example, the performance gain is still significant.  While a core might only support 4 hyperthreads, they can still be individually switched via task switching as well as running simultaneously with others. Thus, in practice, a hyper thread may actually be a group of alternating threads running in parallel with another group of alternating hyperthreads. |

**Online**:Section 4.10-4.11, Computer Architecture and Operating Systems, University of York

**Print**:Chapter 10.8-10.9, Computer Architecture and Operating Systems, Crispin-Bailey

Chapter 12.3-12.3.7, Computer Systems: A Programmer’s Perspective, Bryant et al