**Learning Outcomes**

|  |
| --- |
| * Demonstrate an understanding of operating systems models and hierarchies, types, and standards in brief * Show appreciation of how programs run, scheduling, and how their characteristics can be measured |

**Summary**

|  |
| --- |
| When executed programs need to be assigned memory to store data that they may generate during their run time: this can be dynamic or static in size. When physical memory is exhausted, virtual memory can be created to allow programs to continue to function.  A storage disk can be partitioned to allow for more drives or joined with other disks to form larger drives.  Files can be compressed to fit more onto a storage disk. This comes at a CPU cost, as they must be decompressed in order to be opened, but often this remains beneficial. |

**Lesson 1: Operating Systems - Going Deeper**

|  |  |  |
| --- | --- | --- |
| **Memory Management**     * **Memory Allocation** * **Access Control** |  | Along with workload management, the OS also must manage memory resources.  Systems have a limited amount of physical memory available. If operating systems were constrained by this amount, may of the expected features of a modern system may not be possible.  **Memory allocation** is the process by which a task obtains temporary ownership over a memory space.  There are two explicit allocation types:   * **Static** – a fixed amount of memory is assigned to the program for use when it starts * Execution time is controlled, but memory wastage may occur * **Dynamic** – a kernel function allows a program to request more memory during its lifetime * Execution time increases, but memory can be freed   Additionally, standard default memory allocation is provided to all tasks. This is stored in a region called the **stack**, which must be provided with enough memory by the OS.  Memory must also be de-allocated, or **released**, when a task is finished to allow it to be used elsewhere.  Repeated allocation and freeing of memory leads to **memory fragmentation**, meaning a program may have blocks of memory scattered across the memory space. This incurs a performance cost.  Program code must also be allocated memory space like data is, as it is loaded into memory when started.  Code-space memory allocation is influenced by a process called **linking**, which is part of the process of converting source code to executable machine code. There are two types of linking:   * **Static linking** – all parts of the program are combined into a single self-contained block of program code which is then loaded in its entirety into memory * Execution speed is faster as all the modules of the program are present from the start * However, size of the program is limited by the memory size * **Dynamic linking** – the core of the program is loaded into memory, with additional parts loaded or discarded as appropriate * Reducing the **initial memory footprint** of the program allows for larger programs * However, this can cause delays when modules or features are first used   Therefore, fast, predictable, real-time behaviours may be better served by static linking.  The OS must ensure tasks are private and do not interfere with other memory areas, unless allowed:   * This provides **security** against unwanted behaviours * Malfunctioning processes are prevented from corrupting memory of other processes * Potentially sensitive data cannot be stolen from other process memory areas |
| **Virtual Memory**     * **Paging Cost** |  | Rather than return an error when physical memory is exhausted, **virtual memory** allows the system to carry on operating. Thus, a modern system has both:     * **Physical memory** – the actual chips on the motherboard. Contents of the logical memory can be swapped back and forth as per the demands of the system and user * **Logical memory** – the bulk is stored on a secondary medium, such as an HDD. Since storage devices have large capacity, logical memory can therefore be huge   The process of dumping physical memory into logical memory to make room for retrieval from logical memory to physical memory is known as **paging**:     * This is performed by an OS module called the **paging supervisor** * The blocks of swapped memory are referred to as **pages** * The storage space is known as the **swap file** * The disk used is called the **swap disk** and can be a bespoke drive if performance is needed * Swapping pages of the application rather than the entire application is more efficient * The physical address of the page will change as the supervisor looks to fill unused spaces * Therefore, a **page table** provides address look ups, possibly managed by hardware   The physical memory effectively caches the most needed and used subset of the larger virtual memory.  Paging incurs performance costs, as it is moving data around a system. Even with an SSD this is significant.  **Thrashing** is when swaps occur too frequently, as tasks compete for limited resources. The system spends a lot of time waiting for the disk unit to catch up with transfer requests than it does progressing tasks.  This could be alleviated by:   * Increasing the size of the physical memory but can be cost ineffective * Using an SSD to mask the impact of thrashing to some degree * Creating **pinned pages** that must always be in the physical memory, such as OS functions |
| **Process Communication**   * **Pipes** * **Message Queries** * **Shared Memory** |  | Threads can share data held in memory as they are siblings under a process, however processes are limited from doing so to provide security.  However, this becomes possible if the OS supports **inter-process communication**.  A **pipe** can allow two processes to communicate: a one-way conduit for sending data under OS supervision.     * Pipes are unidirectional, but a second can be added to allow for two-way data flow * Typically, they have a buffer, allowing blocks of data to be sent not just single bytes * This is efficient as it allows both tasks to execute in different time slices * Data is FIFO, preserving the order of the bytes sent   One problem with pipes is that they can only exist between processes which share the creating process. Without a shared parent, the processes have no contextual knowledge the other exists.   * This can be overcome with a **named pipe** * As it has a fixed name, any process can look for it in the OS environment   Pipes only exist between two set point and cannot be used in a generalised or broadcast fashion.  **Producer-consumer** models can facilitate this functionality:     * A **message queue** can be created, and any process can **post** information to the queue * Consumer processes can read messages in the queue and deal with the contents * Messages can be removed by the process or retained depending on the need * A message typically has a short header followed by a block of fixed data   Defined messaging standards can be written into pre-defined code modules, known as **Application Programming Interface** (APIs). These allow communication to be simplified by handling the details.  Many operating systems have messaging processing functionality integrated to allow events such as mouse clicks changing windows, receiving messages from USB devices etc.  An OS can support shared memory, accessible by multiple processes simultaneously. A block of memory is reserved by a process and then creates sub-processes which can then access that block.   * This is generally faster than using pipes or message passing * However, processes must not be allowed to access an area of memory at the same time * **Semaphore** is a method that signals when memory areas are available * Data is therefore accessed under the principle of mutual exclusion |
| **Privileges / Restrictions** |  | Restrictions and privileges help protect the integrity of the system from unauthorised changes.  Privileged access in an operating system provides security advantages. It prevents:   * Unauthorised access to blocks of memory, called **memory protection** * Programs running without OS control * Execution of specialised CPU instructions   The actual definition of privilege levels changes according to the OS being used. Linux has a kernel mode which has very low restrictions and a user mode which restricts programs from accessing prohibited areas.   * Restraints on processes are also inherited by their created threads * A user process can only access resources that need kernel mode privileges must request access through the kernel as a reliable intermediary * Windows is slightly different; however, the overall philosophy will be the same |

**Online**:Section 5.3-5.4, Computer Architecture and Operating Systems, University of York

**Print**:Chapter 11.1-11.6, Computer Architecture and Operating Systems, Crispin-Bailey

Chapter 8.3, Computer Organisation and Architecture, Stallings

**Lesson 2: File Systems**

|  |  |  |
| --- | --- | --- |
| **File Systems**     * **File System Concept** * **Boot Sector** * **Volumes / Partitions** * **Root and Hierarchy** * **Special File Cases** * **File Attributes** |  | Data has perhaps changed more than any other system aspect. This can be seen in:   * The growth in data volume being managed by systems * Increase in the diversity of data types * Methods of storage and retrieval, both remote and networked   Consequently, file management has had to evolve to handle and make use of these developments.  In a file system, each file is a separate data entity: eg, a document, a program, an image etc   * On one level every file is a sequence of binary values * However, the arrangement of these bytes dictates a particular file format   An HDD stores data in tracks and sectors. As sectors are limited in size, a file will consist of many sectors.   * It is unlikely, but a file may fill the sectors they’re assigned * However, it is more likely that there will be wasted space, empty of valid data * Every file on average wastes 0.5 sectors of data storage * Making these sectors larger to accommodate data has negative effect on file management   Consequently, as sectors are likely distributed across the disk, a **file allocation table** is needed to track the location of the sectors belonging to each file. Smaller the sectors, the more that need tracking.  The expectations of a file system can therefore be outlined as:   * Keeping track of the identify of files on a disk * Knowing which sectors relate to which file * Confirming to a standard, eg FAT or NTFS, to allow **interchangeability** * **Efficient** to a degree, able to balance competing requirements * Some **fault tolerance** of disk problems * Some degree of security assurance   Most file systems allow for the presence of a boot sector: a portion of the disk containing start up code.  The disk is usually configured to meet a file system standard and contain a boot sector when **formatted**.  Disks do not necessarily need to be bootable: they can serve solely as secondary disks.  **Partitions** allow a physical storage drive to be divided to act as separate drives, with their own formatting.    Conversely, multiple drives can be aggregated to appear as a single drive, called a **spanned volume**.    The complexities of these two configurations are hidden.  All file systems have a **root** directory: the first place an OS will look for files on the disk.  However, all the files being placed in the root directory would be messy and difficult to navigate for a user. Consequently, a **hierarchical file system** is used to provide organisational sensibility.     * Named containers called sub-directories can be created to hold groups of files * These containers can also contain containers, establishing a nested structure * The OS will typically have its own sub-directory * Some systems will allow access to certain directories to be restricted   A **virtual root** directory can be used, which will prevent access to the rest of the file system.    The file system hierarchy allows for the illusion of a particular file configuration, when in fact the file is located elsewhere. This is possible through two different approaches:   * **Symbolic links** – file or folders appear in one location, but this is a hidden link to its true location elsewhere. This provides convenience to the user * Files can be freely moved around and while the link is valid the user never needs to search for its location * **Virtual files** – files and directories can be located on a networked computer, but visible to a user on another. The file can be made to appear as if its local to the user’s system. * This may allow a file to be accessed through a computer but maintained by an administrator in another location   Most file systems allow files to have attributes. These control how the file is used and contains metadata.  These attributes are not universal, but may include:   * **Hidden** – if the file is visible to the user * **Executable** – whether the OS will run a file * **Syste**m – if the file belongs to the OS or kernel, typically restricted access * **Read-Only** – can be viewed but not modified * **Archive** – used by some operating systems to control the backup policy   Groups of users may have permissions set for certain files. |

**Online**:Section 5.5-5.6, Computer Architecture and Operating Systems, University of York

**Print**:Chapter 12.1-12.9, Computer Architecture and Operating Systems, Crispin-Bailey

**Lesson 3: File Formats and Storage Options**

|  |  |  |
| --- | --- | --- |
| **File Compression**     * **Huffman Compression** * **RLE** * **Automatic Systems** * **File Encryption** |  | File compression is a technique that allows a portion of data to occupy less space on the disk.  There are two factors within the scope of compression:   * **Lossless compression** – the ability to reconstruct data in its exact original form * **Lossy compression –** returns only an approximation of the original data * When considering file systems, lossless is almost certainly the correct choice   **Huffman compression** is one of the most widely used file system compression methods.   * It is often used with LZSS compression * The ZIP file format supports many compression methods, including Huffman-LZSS   Huffman compression works by assigning lower digits to the most frequent letters, and vice versa:     * Numbers could be assigned to letters: E (05) D (04) E (05) N (14) = 05040514 * However, these are all equally sized and offer no benefit * Smaller digits can be assigned to frequent letters: E (1) D (3) E (1) N (109) = 131109 * This improves the compression from 8 digits to 6 * The compression ratio means CR = 8/6 = 1.33; anything over 1.0 is an improvement   There is a cost however, as both compression and decompression take CPU time. Furthermore, removing redundant information negatively impacts fault tolerance, possibly resulting in the file being corrupted.  **Run Length Encoding** (RLE) is frequently used with graphical data.     * Each square represents a pixel of an image * The algorithm looks for consecutive sequences, or runs, of pixel values * This results in a run, value structure which helps reduce the file size   As compression results in significant benefits, some disks and operating systems automatically apply it to all files stored. This contrasts with user directed file compression.  This results in several competing factors between the CPU and disk:   * Compressing a file requires CPU effort * CPU effort uses power and time * Compressed files are smaller so use less disk space * Less data means less effort expended moving it around the system * Smaller files have faster access time and less fragmentation   For comparison, the impact of compression can be seen in an example system:   * Average seek time 5 ms, average file size 100 sectors, 0.1 ms per sector to read   5 + (100 \* 0.1) = 15 ms average file read time   * Whereas, if the average file size is compressed by 60% and decompression takes 2 ms:   5 + 2 + (60 \* 0.1 ) = 13 ms average file read time   * Read performance is improved by 15% as well as reducing storage by 40% * Thus, the **break-even point** in this example is compression effort being below 4ms   Compression is sometimes applied along with encryption: a mathematical process to secure data.  Encryption algorithms must become more complex to keep up with increasing computing power.  For general use, file encryption is unlikely to be compromised within a short time. If longer term security is needed, stronger encryptions may be needed in future to re-encrypt the data.  Some disk systems include inbuilt encryption, notably in USB flash drives due to their portability. |
| **File System Resilience** |  | Physical integrity of data can be protected by using suitable hardware configurations, such as RAID. However, the resilience filesystem also requires protection through OS functionality and design.   * Data integrity needs to be ensured in unexpected conditions, such as power failure during a file move * Overwriting a file may be safely achieved through writing over a duplicate, deleting the original version only when the update has completed and been verified * This type of approach needs to be extended to the effects of virtual memory and the many caches in use across the system   Ultimately, there is a compromise between cost efficiency, good performance, achievable, and tolerable.   * For a general user, loss of data during a power cut leaving only an autosave is tolerable * Controlling the modification of data through file attributes is achievable * **Virtual root directories** to hide sensitive data from users and programs is cost effective |
| **Internal File Formats**     * **Structures / Encoding** |  | **ASCII** text files contain sequences of bytes, where each byte corresponds to a letter, number, or symbol. This file could be a list of items, a document, data structure, or a program source code.  **XML** organises a range of values in a way that is readable by users and computers. The repeated tags may waste space but ensure the computer can identify data if the order changes in future.  **Binary files** are continuous sequences of bytes with no human readability. Many file formats contain some binary structure as it is an efficient way to store data. This includes BMP, PNG, JPEG, WMV and MPEG.  File formats are indicated by their **file extension**: eg dog.bmp and shopping.txt.  Usually, they include an initial portion of data called the header. This contains information about the file and the version number, allowing for legacy versions to identified and thus **backward compatibility**.  It is followed by a payload, which is the actual encoded file data.  Many files, such as BMP, include algorithmic compression as part of the file encoding.  Ideally, a file encoding is a well defined and maintained standard. Any developer wishing to make use of the standard must therefore adhere to it for compatibility. |
| **File Management** |  | Most users only ever use filesystems as part of a window-based GUI.  More advanced users can use the shell or command line capability, which provides more sophisticated functionality provided the user knows how to use it:   * Different keywords and syntax may be present in different systems: eg MS-DOS, Linux * Often there are no undo commands to reverse errors * Shell scripts can be written to execute sequences of commands on request |

**Online**:Section 5.7, Computer Architecture and Operating Systems, University of York

**Print**:Chapter 12.10-12.16, Computer Architecture and Operating Systems, Crispin-Bailey