Contents

TEST

Adam Hawley

April 8, 2019

1 Lecture 10: Introduction to Memory Management

1.1 Background

1.1.1 Introduction

Programs must be brought (from storage) into memory and placed within a process for it to be run. The main memory and registers are the only storage entities that a CPU can access directly. The CPU fetches instructions from main memory according to the value of the program counter. A typical instruction execution cycle looks like this:

1. Fetch instruction from memory
2. Decode the instruction
3. Operand fetch
4. Possible storage of result in memory

Memory units only see a stream of one of the following:

- Read request + address
- Write request + data + address

Memory unit does not know how these addresses are generated. Register access can be done in one CPU clock while completing a memory access may take many cycles of the CPU clock. In this case the processor needs to **stall** since it does not have the data required to complete the instruction it is execution. (In reality not 100% true because modern CPUs use techniques such as *out-of-order execution*.

The **Cache** sits between the main memory and CPU registers to mitigate the *stall issue*. Protection of memory is required to ensure correct operation. User processes should not be able to access OS memory and one user should not be able to access the memory of another user process.

1.1.2 Address Spaces

A logical address space is a range of addresses that an operating system makes available to a process. It is up to the OS to enforce **memory protection**. Address space endpoints are a **base** register (holding the smallest legal physical address of the process in the memory) and a **limit** register (specifies the size of the address space). The CPU must check that every memory access generated in user mode is between the `base` and `base + limit` for that process.

A program residing on the disk needs to be brought into memory in order to execute. In general, we do not know a priori where the program is going to reside in memory.

- Addresses in the source program are generally symbolic
  - e.g `count`
- A compiler typically binds these symbolic addresses to relocatable addresses
  - e.g "14 bytes from the beggining of this module"
- **Linker** or **loader** will bind relocatable addresses to absolute addresses

Addresses are represented in different ways at different stages of a programs life:

Compile Time If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes.

Load Time If memory location is not known at compile time and no hardware support is available, **relocatable code** must be generated.

Execution Time (Most common in general computing) Binding delayed until run time if the process can be moved during its exectuion from one memory segment to another. This needs hardware support for address maps (e.g base and limit registers).

1.1.3 Logical vs. Physical Address Space

The concept of a **logical address space** that is bound to a separate **physical address space** is central to proper memory management.

Logical Address Issued by the CPU, within processs address space

Physical Address Address seen by the memory unit.

Logical and physical addresses are:

- The same in compile-time and load-time address-binding schemes
- Different in execution-time address-binding schemes.

In the latter case, the logical address can be referred to as the virtual address.

Logical Address Space The set of all logical addresses generated by a program.

Physical Address Space The set of all physical corresponding to a given logical address space.

1.1.4 Memory Management

A **Memory Management Unit (MMU)** is a hardware device that at runtime maps logical addresses to physical addresses. The user program deals with logical addresses and never sees the real physical addresses. Execution-time binding occurs when reference is made to location in memory. Logical addresses bound to physical addresses.

1.2 Contiguous Memory

1.2.1 Single-User Contiguous Memory

First computers: all memory assigned to a single job. Key points: contiguous + entirely assigned. Advantages:

- Very simple
- Address resolution: trivial (physical address = issued address)

Disadvantages:

- Only one job can run at a time so this cannot support multi-programming.
- Processor unused during I/O operations.

1.2.2 Fixed Contiguous Partitions

OS assigns one partition per process, size of partitions defined at boot time and never changes. Key point is that it has protection against memory intrusion. The OS must be assigned its own partition. Upon starting a new process, the OS has to:

1. Determine the relevent partition
2. Determine the start address within the active partition
3. Resolve addresses: `physicalAddress = issuedAddress + fixedBaseRegister`

The problem with this approach is that it is often difficult to choose the right partition sizes which can cause the following. **Internal fragmentation** is when a process may require less space than the available partition. Or process creation may fail even though there may be enough free memory due to wasted memory by small jobs.

1.2.3 Dynamic Contiguous Partitions

Partition size is selected when the job is loaded. Address resolution becomes:

```
physicalAddress = issuedAddress + variableBaseRegister
```

This approach alleviated the problems of fixed contiguous partitioning but does not solve it completely. It is still possible to get **External Fragmentation** where the OS has to keep track of free partitions.

Partition Allocation Problem How to satisfy a request of size n from a list of free partitions?

First-fit Allocate the first partition that is big enough

Best-fit Allocate the smallest partition that is big enough

- Must search entire list, unless the list is ordered by size
- Produces the smallest leftover partition

Worst-fit Allocate the largest partition

- Must also search entire list, unless the list is ordered by size
- Produces the largest leftover partition

Random As it sounds, take a random partition.

There is no clear winner as performance of each depends on the request patterns

Mitigation of External Fragmentation External fragmentation can be mitigated by a **compaction** (or defragmentation) procedure. This requires **relocatable partitions** where the base register needs to be changed. The compaction algorithm needs spare memory space to operate efficiently (i.e to move small partitions out of the way before large partitions can be relocated). Compaction cannot be performed while I/O is in progress involving memory that is being compacted. Alternatively, the CPU can latch the process in memory while it is involved in I/O, or do I/O only into OS buffers (i.e double buffering).

1.2.4 Swapping

A process can be **swapped** temporarily out of memory to a **backing store**, and then brought back into memory for continued exectution. This means that the total physical memory space of the processes can exceed physical memory.

The **Backing Store** is a fast disk large enough to accomodate binaries of all processes. A major part of swap time is transfer time and the total time is directly proportional to the amount of memory swapped.

2 Lecture 11: Segmented & Paged Memory

2.1 Segmentation

2.1.1 Introduction

Memory-management scheme that supports user view of memory. A program is a collection of segments where a segment is a logical unit search as: main program, procedure, function, method, object, local variables, global variables, stack, array etc. Segments have variable sizes. This approach means that when you are in the main program, only the main program needs to be stored in memory.

2.1.2 Addressing with Segmentation

The logical address consists of a tuple: `<segment-number, offset>`. The addresses are stored inside a **segment table** which maps a two-dimensional logical address to a one-dimensional physical address.

`base` contains the starting physical addrss where the segments reside in memory

`limit` specifies the length of the segment

Segment table is kept in memory:

`segment-table base register` **(STBR)** points to the segment table's location in memory

`segment-table length register` **(STLR)** indicates number of segments used by a program

- segment number `s` is legal if `s < STLR`

2.2 Paging

2.2.1 Introduction

Physical address space of a process can be noncontiguous: process is alllocated physical memory whenever the latter is available:

- avoids external fragmentation
- avoids problem of varying sized memory chunks

Divide physical memory into fixed-sized blocks called **frames** (size is a power of 2, between 512 bytes and 16 Mbytes). Then divide the logical memory into blocks of the same size called **pages** (internal fragmentation is still a minor issue).

2.2.2 Costs & Implementation

The OS must:

- Keep track of all free frames in memory
- (to run a program of size N pages), need to find N free frames and load program
- set up a **page table** to translate logical to physical addresses — kept in memory

  Page-table base register (PTBR) Points to the page table

  Page-table length register (PTLR) indicates size of the page table

2.2.3 Address Resolution

Assume the logical address space is 2^m and that the page size is 2^n then the address generated by the CPU is divided into:

page number (p) used as an index into a **page table** which contains base address of each page in physical memory

- size of **p** is `m - n` bits

page offset (d) combined with base address to define the physical memory address that is sent to the memory unit

- size of **d** is `n` bits

2.2.4 Internal Fragmentation in Paging

Internal fragmentation happens when the process requires memory which is not a multiple of the page size, when this happens, the last page will cause internal fragmentation as it will not fill the frame. The worst case fragmentation would be equal to `1 frame - 1 byte` but average fragmentation is around half a frame size.

Page Size Trade-Off

- Reducing the page size $\rightarrow$ minimises internal fragmentation
- Increaseing the page size $\rightarrow$ less pages needed, reduces page table size (faster, simpler implementation of MM)

2.2.5 Performance Issues

If the page table is kept in main memory every data/instruction access requires two memory accesses (one for the page table and one for the data/instruction). The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**. The TLB is typically small (64 to 1,024 entries). Frequently accessed pages will have their frames stored in a TLB. On a TLB miss, the value of the (missed page-table and frame-number), is loaded into the TLB for faster access next time that address is used (if there is no free TLB entry, replacement policies must be considered). Some entries can be **wired down** for permanent fast access.

Address translation `(p,d)`:

1. If `p` is in associative register, get frame # out
2. Otherwise get frame # from page table in memory

2.2.6 Shared Pages

Shared Code Processes that are read-only can be shared because there is no danger of modification. This means that only one copy is needed. It is similar to the idea of multiple threads sharing the same process space and is also useful for interprocess communication if sharing of read-write pages is allowed.

Private Code & Data Each process keeps a separate copy of the code and data. The pages for the private code and data can appear anywhere in the logical space.

2.2.7 Page Table Structure

Memory structures for paging can get huge using straight-forward methods:

- Consider a 32-bit logical address space
- Page size of 1KB (2^{10})
- Page table would have 4 million entries ($2^{32}/2^{10}$)
- If each entry is 4 bytes, page table is of size 16 MB
  - Can be costly
  - Do not want to allocate that contiguously in main memory

There are several approaches to this problem:

- Exploit heirarchy
- 64-bit address spaces require even more sophisticated solutions

2.2.8 Hierarchical Page Tables

Break up the logical address space into multiple page tables A simple technique is a two-level page table. We then page the page-table.

A logical address (on a 32-bit machine with 4K page size) is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Since the page table is paged, the page number is further divided into a 10-bit index p_1 into the outer page table and a 10-bit displacement p_2 within the page of the inner page table.

3 Lecture 12: Virtual Memory

3.1 Demand Paging

3.1.1 Introduction

Program is permanently stored in a backing store and is swapped in as needed. The backing store is also split into storage units (called **blocks**), which are the same size as the frame and pages. The prgram ends up with a small set of pages loaded in main memory — a *working set*. Programs are executed (more or less) sequentially and coverage of the program is generally small (many functions are seldom used e.g. error handling routines, mutually exclusive modules, maintenance etc.).

3.1.2 Advantages of Demand Paging

Demand paging allows to run programs which require much more memory than physically available (limited by secondary storage or addressing space).

3.1.3 Requirements of Demand Paging

- Fast secondary storage device: DMA
- More decisions to be taken by the OS:
  - What to do when all memory is full and a new page is needed? - Which page is replaced?

3.2 Virtual Memory

3.2.1 Introduction

Separation of user logical memory from physical memory. Logical address space can be much larger than physical address space since only part of the program eeds to be in memory for execution. More performance and resource efficiency since less I/O needed to load or swap processes and allows for more efficient process creation.

3.2.2 Virtual Address Space

Usually design logical address space for stack to start at Max logical address and grow *down* while heap grows *up*. This maximises space use by leaving unused address space between the two, this means that no physical memory is needed until the heap or stack grows to a given new page. It also enables sparse address spaces with holes left for growth, dynamically linked libraries etc. System libraries shared via mapping into virtual address space. Shared memory by mapping pages read-write into virtual address space. Pages can be shared during `fork()`, speeding process creation.

3.3 Demand Paging Mechanism

Similar to paging system with swapping. Pager is a swapper that deals with pages that only loads the needed pages. A page is said to be needed when there is a reference to it (read/write to its address range).

- Invalid reference → Abort
- Not-in-memory → Bring to memory

When a process is to be swapped in, the pager predicts which pages will be used before the process is swapped out again. So instead of swapping in a whole process, the pager brings into memory only its estimate of the working set. The OS must distinguish between the pages that are in memory and the pages that are on the disk (to do this, usually a valid-invalid scheme is used). If pages needed are already **memory resident** then there is no difference from regular paging. If the page needed is not **memory resident** then there is a **page fault** and the OS needs to detect it and load the page into memory from storage without changing the program behaviour and without the programmer needing to change code.

3.3.1 Handling Page Faults

A page faut is an interrupt so a context switch ensues. This means that the process state is saved and the OS is enabled to restart the instruction that caused the page fault, as the CPU will be in exactly the same state as prior to the memory reference. The OS will do the following upon page fault:

1. Find a free frame
2. Swap page into frame via scheduled I/O operation
3. Reset tables to indicate page now in memory (set validation bit equal to valid)
4. Restart the instruction that caused the page fault

In the extreme case, a process may be started with none of its pages in memory. The solution is *pure demand paging*.

1. OS sets instruction-pointer to the first instruction of the process, non-memory-resident.
2. Page fault signalled and recovered
3. Same for every other process pages on first access.

A single instruction can access multiple pages and cause multiple page faults. For example, fetch and decode of an instruction which adds 2 numbers from memory and stores the result back to memory. The two numbers may reside in two different pages.

3.3.2 Demand Paging Mechanism Performance

There are three major activities:

1. Service the interrupt — few hundred instructions
2. Read in the page — lots of time
3. Restart the process — few hundred instructions

Page Fault Rate The liklihood of a memory access to be a page fault:

- If $p = 0 \rightarrow$ no page faults
- If $p = 1 \rightarrow$ every reference is a fault

Effective Access Time (EAT) `EAT = (1 - p) * memory access time + p(page fault overhead + swap page out + swap page in )`

3.3.3 Demand Paging Mechanism Optimisation

- Swap space I/O faster than file system I/O even if on the same device. (The swap is allocated in larger chunks so there is less management needed than file system)
- Copy entire process image to swap space at process load time and then page in and out of swap space. (Need to remember the big initial copy cost)
- Demand page from program binary on disk, but discard rather than paging out when freeing frame.
  - Used in Solaris and current BSD
  - Still need to write to swap space
- Copy-On-Write (COW) allows both parent and child processes to initially share the same pages in memory (if either process modifies a shared page, only then will a page be copied). This allows for more efficient process creation.

- Variation on `fork()` system call has parent suspended and child using address space of parent (`vfork()` on Linux and designed to have the child make an `exec()` call). This is very efficient.

4 Lecture 13: Page Replacement

4.1 Introduction

4.1.1 When does Page Replacement Occur?

When a page fault occurs, we need to bring the desired page into memory. However, sometimes there are no free frames for the page to fill. This is when we must replace one of the pre-existing pages to with the new one.

4.1.2 Outline of Page Replacement

Page replacement is done by finding some page in memory, but not really in use and paging it out. We use an algorithm to decide which frame to free. The performance of this algorithm is based on trying to achieve the minimum number of page faults possible. When replacing pages it should be noted that the same page can be brought into memory several times.

4.1.3 Optimisation of Page Replacement

Use a **modify*(*dirty**)*bit* to reduce overhead of page transfers (then only modified pages are written back to the disk and we are not rewriting unchanged pages). Page replacement completes separation between logical memory and physical memory (large virtual memory can be provided on a smaller physical memory).

4.2 Replacement Mechanism

1. Find the location of the desired page on disk
2. Find a free frame
   - if there is a free frame, use it
   - if there is no free frame, use a page replacement algorithm
     - select a **victim frame**
     - write victim frame to disk if it is *dirty*
3. Bring the desired page into the (newly) free frame (i.e. update the page and frame tables).
4. Continue the process by restarting the instruction that caused the trap.

4.3 Page Replacement Algorithms

Remember: page-replacement algorithms aim for the lowest page-fault rate on both first access and re-access. Algorithms are evaluated by running it on a particular string of memory references and computing the number of page faults on that string. The string is just page numbers, not full addresses. Repeated access to the same page does not cause a page fault. Results depend on the number of frames available.

4.3.1 FIFO

Just use a FIFO queue to keep track ages of pages. One would expect that the more frames are allocated to a process, the fewer page faults. In FIFO this is not necessarily true (see Belady's anomaly).

4.3.2 Optimal

Best possible replacement policy: replace page that will not be used for longest period of time. If we can see into the future (like when we have a reference string) then we can see which page will not be used for the longest period of time. Obviously this is really only theoretical so it is often used mainly as an upper-bound in comparative evaluation.

4.3.3 Least Recently Used (LRU)

Use past knowledge rather than future: replace page that has not been used for the longest period of time. Obviously, this involves keeping track of time of last use for each page. Using the lecture reference string, it had 12 faults which was better than FIFO but worse than OPT.

It is generally a good approach and widely used but there are implementation issues.

4.3.4 LRU Approximation Algorithms

These are obviously similar to LRU but without complex timestamping.

- **Reference Bit** :
  - With each page associate a hardware-provided bit; initially 0
  - When a page is referenced the associated bit is set to 1
  - Replace any page with reference bit = 0 (if one exists). However, we do not know the order
- **Second-Chance Algorithm**:
  - FIFO scheme, plus hardware-provided reference bit
  - If page to be replaced has
    - Reference bit = 0 $\rightarrow$ Replace it

* Reference bit = 1 then:
    - Set reference bit 0, leave page in memory
    - Replace next page, subject to same rules

4.4 Page and Frame Locking

The OS may wish some pages to remain in physical memory frames (e.g parts of the OS code itself or I/O buffers).

4.4.1 Frame Locking

If a frame is locked, it may not be replaced. It requires a lock bit with each frame (usually supported by the OS rather than hardware).

4.5 Frame Allocation

Frame allocation algorithms determine how many frames to give each process and which frames to replace in case of scarcity. Each process needs a **minimum** number of frames. The **maximum** is the total frames in the system. There are many allocation schemes such as *equal allocation*, *size-proportional allocation*, *priority allocation*, etc. It is important to keep a free frame buffer pool.

4.5.1 Global Replacement

Global Replacement Process selects a replacement frame from the set of all frames.

- One process can take a frame from another
- In priority based allocation of frames, this may enable a high priority process to increase its allocated frames by taking a frame froom a low priority process
- Page fault behaviour of a process becomes dependent on the behaviour of other processes
- Greater overall throughput, so more common (e.g. Linux)

4.5.2 Local Replacement

Local Replacement Each process selects from only its own set of allocated frames.

- More consistent per-process performance
- If a process does not have sufficient number of frames allocated to it, the process will suffer many page faults (thrashing)
- Possibly underutilised memory

4.6 Thrashing

Thrashing When a process is busier swapping pages in and out then executing itself.

If a process does not have *enough* pages, the page-fault rate is very high. Insufficient frames lead to page faults. Pages are swapped out and then needed again, so page fault so another page is swapped out which is needed again...

This can lead to:

- Low CPU utilisation
- Operating system thinking that it needs to increase the degree of multiprogramming
- Another process added to the system

4.6.1 Solving Thrashing

Locality Model Demand paging works well using a **locality model**. This is when processes migrate from one locality to another (localities may overlap). Thrashing occurs when:

$$\sum \text{size of locality} > \text{total memory size} \tag{1}$$

The effects of thrashing can be limited by:

- Local page replacement
- Priority page replacement — replace a page from a process with the lowest priority.

Working-Set Model Define Δ to be a working-set window. Then analyse the most recent Δ page references. If a page is in use, it is in the working set but if it is no longer used then it will drop from the working set Δ time units after its last reference.

WSS_i(working set of Process P_i) is defined to be the total number of pages referenced in the window Δ. WSS_i tries to approximate the size of the locality of process P_i:

- If Δ is too small will not encompass entire locality
- If Δ is too large will encompass several localities
- If $\Delta = \infty \rightarrow$ Will encompass the whole program

$$D = \sum_{i=0}^{n} WSS_i \tag{2}$$

If $D > m \rightarrow$ Thrashing (where m is the total number of frames).

Policy: if $D > m$, then suspend or swap out one of the processes.

Page-Fault Frequency This is a more direct approach than WSS. It is done by establishing an *acceptable* **page-fault frequency** (PFF) rate and use local replacement policy. If the actual rate is too low, process loses a frame and if the actual rate is too high, the process gains a frame.

5 Lecture 14: Input/Output & Storage Management

5.1 I/O Management

5.1.1 Introduction

I/O subsystem is responsible for controlling devices connected to a computer. It must provide processes with a sufficiently simple interface and also take device characteristics into account to maximise performance and efficiency.

There is a large variety of I/O devices:

- Storage (e.g disk drives, non-volatile memory)
- Communications (e.g. Ethernet, WiFi, Bluetooth, USB)
- User Interface (e.g. mouse, touch, keyboard, display, sound)

5.1.2 Device Drivers

Device drivers are low-level sofware that interacts directly with device hardware, they hide the hardware details to the higher levels of the OS and user applications and are often developed by the hardware vendor. They track the status of the device and enforce access/allocation policies. Types of drivers:

Dedicated Each device is allocated to a single process

Shared Each device is shared between multiple processes

Virtual Hides sharing from processes

5.1.3 Devices

Devices usually have registers where the device driver places commands, addresses and data to write or read data from registers after command execution. A minimum setup usually consists of the following:

- Data-In Register
- Data-Out Register
- Status Register
- Control Register

Where each register is typically 1-4 bytes and they may be contained in a FIFO buffer.

Devices themselves also have addresses used by direct I/O instructions or memory-mapped I/O.

5.1.4 I/O Management

The I/O subsystem provides interfaces to access devices via device drivers (or access to specific devices in a family of devices hidden by the device driver). There are three main device communication mechanisms:

- Polling & Interrupts
- Direct Memory Access (DMA)
- Buffering

5.1.5 Polling

Polling is about checking if a device is ready for communication so for each byte of I/O:

1. Read busy bit from status register until 0
2. Host sets read or write but and if write copies data into data-out register.
3. Host sets command-ready bit
4. Controller sets busy bit, executes transfer
5. Controller clears busy bit, error bit, command-ready bit when transfer done.

Step 1 is a busy-wait cycle to wait for I/O from device. This is reasonable if the device is fast but inefficient if it is slow. The CPU could switch to other tasks, but if miss a cycle data could be overwritten/lost.

5.1.6 Interrrupts

Polling can happen in 3 instruction cycles:

1. Read status
2. Extract status bit
3. Branch if not zero

How to be more efficient if devices are seldom ready?

CPU Interrupt-Request line triggered by I/O device (checked by processor after each instruction). The interrupt handler receives interrupts (maskable to ignore or delay some interrupts). Interrupt vector to dispatch interrupt to correct handler. This has a context switch at the start and at the end. We get **interrrupt chaining** if more than one device at same interrrupt number.

5.1.7 Direct Memory Access (DMA)

This is used to avoid programmed I/O (one byte at a time) for large data movement. This requires a DMA controller and bypasses CPU to transfer data directly between I/O device and memory.

The OS writes DMA command block into memory.

- Source and destination addresses
- Reade or write mode
- Count of bytes
- Writes location of command block to DMA controller.

5.2 Storage Devices

5.2.1 Introduction

The hierarchy of storage devices is driven by performance and volatility of data. **Data access time** includes:

Ready time Time to prepare set up storage media to read/write data at the appropriate location (e.g. wind/rewind tape, rotate disk, charge memory row)

Transfer time Time to read/write data from media

Different devices may impose access latencies at different orders of magnituse and hence, the OS should manage each of them appropriatly and mediate transfers (e.g. buffering).

5.2.2 Tertiary Storage

Tertiary storage is usually used for backups, storage of infrequently used data and transfer between systems. The two main forms of tertiary storage are:

- Magnetic tapes:
  - GB to TB capacity
  - Very slow access time (must wind and rewind to position tape under read-write head but once in place, reasonable transfer rates >140 MB/s)
- Optical discs:
  - MB to GB capacity
  - Read-only or read-write using high intensity laser beams

5.2.3 Secondary Storage

Secondary storage is mainly used for non-volatile storage, high-capacity storage supporting swapping/paging.

Magnetic Disks (HDDs)

- Made of n disks (2/n/ sides), each side is divided into tracks (circular), and each track into sectors.
- **Cyclinder**: Set of tracks at the same position on all sides
- **Access Time**: Seek time (disk head movement) + Search time (rotational delay) + Transfer time
- Typical Avg Values:
  - Seek = 25ms
  - Search = 4ms
  - Transfer = 0.00094ms/MB
  - Rotation speed = 7200rpm (120rps)

Non-volatile memory (NVMs, SSDs)

- Made of no mechanical components
- More reliable than HDDs (no moving parts), can be faster (no seek time or latency), consumes less power.
- More expensive per MB, lower capacity, may have shorter lifespan (writes wear it out).

Redundant Arrays of Independent (Inexpensive) Disks (RAIDs)

- Set of physical disks viewed as a single logic unit by the OS.
- Simultaneous access to multiple drives
  - Increased I/O performance
  - Improved data recovery in case of failure
- Data is divided into segments called stripes, which are distributed across the disks in the array.
- RAIDs can be classified as level 0,1, . . . , 6 (different levels denote different approaches to data redundancy and error correction methods).

1. RAID 0

- Block-level striping: data is divided into segments that are stored across the disks in the array.
- Minimum disks: 2
- Read speed-up is roughly proportional to the number of disks in the array, because distinct data can be read from different disks simultaneously.
- Write speed-up is roughly proportional to the number of disks in the array, because distinct data can be written to different disks simultaneously.
- No redundancy, therefore no fault tolerance.
- As reliability is inversely proportional to the number of disks, a RAID 0 will be more vulnerable to faults than a single hard disk.

Where $MTTF$ is Mean Time To Failure:

$$MTTF_{group} \approx \frac{MTTF_{disk}}{number} \tag{3}$$

It is possible to use disks of different capacities, but the storage space added to the array by each disk is limited to the size of the smallest disk.

- For example, if a 120GB disk is striped together with a 100GB disk, the capacity of the array will be 200GB.

2. RAID 1
   - Full redundancy: mirroring (data is copied in all disks of the array)
   - Minimum disks: 2
   - Tolerates faults on up to $N - 1$ disks
   - Read speed-up is roughly proportional to the number of disks in the array because distinct daya can be read from different disks simultaneously.
   - No write speed-up, as writes have to be done on all disks.
   - Very low space efficiency: $1/N$
3. RAID 2,3,4 **SKIPPED IN LECTURE AND NOT ASSESSED**: SEE LECTURES/SILBERSCHATZ FOR DETAILS
4. RAID 5
   - Block-level striping, distributed parity. (Think of parity bits from ICAR)
   - Minimum 3 disks, tolerates fault in one disk
   - Writes are costly operations
   - Widely used

5. RAID 6

    - Block-level striping, double distributed parity. (Think of parity bits from ICAR)
    - Minimum disks: 4, tolerates faults in two disks.
    - Writes are costly
    - Widely used

5.3 Storage Management

Multiple requests have to be handled concurrently, several programs may have requested storage operations. There are a number of policies for servicing disk requests.

The I/O scheduler is similar to the process scheduler. It choses which request should be chosen next and often depends on some criteria which usually aim to reduce average response times. There may also be prioritised requests e.g. from OS components. Specific devices may require dedicated scheduling policies (e.g. to minimise seek time in magnetic disks and avoid redundant writes in non-volatile memory).

5.4 Disk Management

5.4.1 Disk Formatting

- Low-level (physical) disk formatting is when a disk is divided into sectors (built-in error correction codes, also called checksums).
- Logical formatting is to do with creating a file-system - directory trees, maps of free and allocated space.

5.4.2 Partitions

Partitions are when there are multiple logical disks on a single physical disk.

5.4.3 Blocks

Boot Block Initial bootable code at a known sector (useful as it helps reduce power-up complexity in hardware).

Bad Blocks These are blocks which have been permanently corrupted and file systems have to be able to mark them.

5.4.4 Defragmentation

Defragmentation is when sectors which are used by files are rearranged to be contiguous.

6 Lecture 15: File Management

6.1 File Management Outline

- Permanent storage of data and programs (files and directories)
- File systems
  - Information and mechanisms used to handle files and directories on disk.
  - File protection
  - Examples: FAT (win9x), NTFS (winNT), ext family, etc.

6.2 File Concept

A file is a *logical storage unit* where the data is non-volatile. Files tend to have attributes such as name, type, location, size, protection, time of creation and modification etc. File information is kept in the directory.

6.3 File operations

A file is an abstract data type that can be manipulated by the applications which can be thought of as an array of bytes. The OS provides an API to operate on files, including functions for:

- File creation
- Opening and closing files
- Read and writing files
- Repositioning within a file
- Deleting or resetting
- Appending
- Renaming
- Copying
- Changing privileges
- Memory-mapped files

6.4 File Types

File types determine which applications can understand the contents of the file. In Unix and Windows, file types correspond to the file name extensions but in old MacOS it was a separate attribute.

6.5 Internal File Structure

Files are stored on devices such as HDD or SDD. Files are stored in physical blocks HDs without a file structure is just an array of blocks. No more than one file can use an individual block so this results in internal fragmentation. It is important to note that **BLOCK** $\neq$ **SECTOR**.

6.6 Directories

A directory is a special file which holds information of other files which is owned and managed by the OS. It provides a name space for the file names — all names in name space need to be unique (effectively maps names to the files themeselves). File operations may also update directories.

6.6.1 Tree-Based Directories

Tree-based directories contain root directories and subdirectories. Files can be accessed using an **absolute path** by prefixing the filename with the sequence of all directories drom the *root* e.g. `\cs\usr\me\lecturing\osi.html`.

They can also be accessed using a **relative path** which is similar to using the absolute path but it makes an assumption that there is a **current directory** e.g. when in `\cs\usr\me` (current) we can just refer to `lecturing\osi.html`. The advantage of this is that there is improved simplicity when dealing with deep trees (normal case).

6.6.2 Non Tree-Based Directories

Non Tree-Based directories allow files to be reached from multiple directories. To do this we have to use aliasing where multiple names point to the same file. This creates extra challenge where extra care should be taken with the aliases to avoid broken links.

6.7 Access Methods

Once a file is opened it can be accessed. During the opening of a file, a type of access method is selected: sequential or direct.

6.7.1 Sequential Access

The file is produced in order, one record after the other. Sequential access is used for applications needing to read files from the start to the end. The API for sequential access typically includes `read_next()`, `write_next()` and `reset()` calls. Writes are usually performed only at the end; data is not always deleted and is instead flagged as invalid.

6.7.2 Direct Access

File is made up of fixed-length logical records. Records can be read and written in any order (non-sequentially). Therefore the API typically includes `read(n)` and `write(n)` calls. Before a read/write operation is performed, the current file pointer has to be moved to the desired position, **SEEK**.

6.8 Access Control

Access control is the control over whether and how users/processes can access a file. Types of access: read, write, execute, travers, list, rename and delete. Permissions on each file are typically given at three levels of privileges:

- Owner
- Group (Groups of users, e.g. people in one department)
- Universe (Any other user)

6.9 Access Granularity

Read-write operations are done in units of **blocks** which are usually one or more sectors of a storage device. Can be from 32 bytes to 64kb but a typical value is 4kb. The OS does the buffering so writes are not done immediately to the disk, only when the block is full, the file is closed or when there is an explicit **flush/synch**.

6.10 File System Mounting

Before a storage device can be used, it should be **mounted** as part of a file system. It verifies that the information in the disk is a valid file system. Before a storage device can be physically removed it should be **unmounted** which involves checking that all files are closed and all information is written to the disk (no information is cached).

6.11 Allocation of Files

Files are stored in storage devices as blocks. There are three main mechanisms to allocate blocks to files:

- Continuous Allocation
- Linked Allocation
- Indexed Allocation

6.11.1 Continuous Allocation

Each file occupies a set of contiguous blocks of disk. This means that the file handler needs to know the first block and the number of blocks.

- Number of disk seeks is minimised since the disk head does not need to move very far.
- Access to file is very easy

However

- Finding space is more difficult (similar problems as external fragmentation in dynamic partitions such as best-fit vs. first-fit policies and relocation of files).
- Allocating file size is also very difficult if files are allowed to grow.

Continuous allocation is used by the IBM VM/CMS operating system for its efficiency.

6.11.2 Linked Allocation

Each file is a linked list of blocks (each block contains the block number of the next block in the list). This allows blocks scattered throughout the disk. The directory holds a pointer to the first and last block of the disk (this makes reading and accessing the end of the file easier).

- Very easy to accommodate growing files.

However

- A major problem is that access to the i^{th} block requires scanning all blocks from the first one this can lead to poor worst-case access times $\Longrightarrow$ especially dependent of disk scheduling.

A possible solution to the problem is to group the blocks continuously in clusters.

There is another major problem with linked allocation, its reliability. If one block gets damaged then the rest of the file is lost!

For example, consider FAT (File Allocation Table): Same idea as a linked list, but links are stored at the beginning of each partition for the whole disk rather than the next block number being stored in the current block. What happes if the FAT becomes corrupted?

6.11.3 Indexed Allocation

Previous methods do not nicely support direct access. In indexed allocation, the first block of a file (the index block) contains a list of the blocks used by the file (similar to the paging memory allocation). This makes direct access easy. The index block is usually cached. Obviously this leads to some wasted space (an additional full block is used for each file). Index blocks can be linked for large files.

6.12 Free Block Management

When a file needs additional blocks it needs to use the ones which are free. A good allocation policy results in improved performance. Methods could be:

- Bit Vector (i.e. 0001000100100111000, where 1 = free and 0 = busy)
- Linked list containing the free blocks
- Grouping to prepare for files which will never have such a small size.

Allocate a free block which is close to the last block on the file to minimise seek times.

6.13 Efficiency & Performance

- Efficiency: How Well Available Space is Used
  - Lost space due to pointer sized (12/16/32 bits)
  - Size of index blocks and allocations tables.
  - Internal fragmentation
- Performance
  - Access time
  - Update time: storing *last update time* requires update of directory entry.
  - Adequate usage of cache is essential
  - Read-ahead improves performance in sequential access.

It is difficult to compare because it depends on how the files will be used. If accesses will be mostly sequential then continuous is best. While if accesses will be mostly direct access then indexed is best. Generally it is a combination of approaches that leads to the best performance.

6.14 Recovery

Data on disk is essential and its loss can cause great harm. In case of a computer failure the data on the disk may be inconsistent e.g. broken links, bad index info. **Consistency checking** is performed when the file system is not properly unmounted. Backup and restore:

- Complete backup or incremental (modified files only, *diff* outputs)
- Using sets of backup devices (tapes, HDs)
- RAID Configs

7 Lecture 16: Introduction to Networking

7.1 Computer Networks

- An interconnection of autonomous computers

They should be ableto exchange information (connected via some medium such as copper wire, microwaves etc.). They also lack a centralised control.

Motivation for developing networks:

- Resource sharing
- Remote access and communication
- Reliability and redundancy
- Economic advantages

7.2 Examples of Networks

- Personal Area Network (PAN)
- Local Area Network (LAN)
- Wide Area Network (WAN)
- Internet

7.3 Network Topologies

Examples:

- Bus
- Ring
- Star
- Mesh

7.4 Connectivity

At the lowest level **nodes** are connected by some physical medium called a **link**. In the course, if it is not otherwise stated then it should be assumed that links are in a **duplex mode** (rather than simplex) where data can be transported in both directions. Physical (direct) connections can be **point-to-point** or **multiple-access**.

7.5 Indirect Connectivity

- Physical connections are often not direct.

When using indirect connectivity there is usually a **switch node** involved which is a node attached to multiple point-to-point links. There are two main classes of indirect connectivity: circuit-switched networks and packet-switched networks.

7.5.1 Circuit-Switched Networks

- A circuit must be established before data is sent.
- All blocks of a large message are always sent via the same route e.g. telephone network.
- Advantage: Time for data to be delivered is predictable once the circuit is set.

SEE SLIDES FOR DEMO

7.5.2 Packet-Switched Networks

- Handle blocks of data (*packets*)
- Different ways to handle packets along the way e.g. store-and-forward, wormhole etc.

SEE SLIDES FOR DEMO

7.6 Interconnection of Networks

An internet is an interconnection of independent networks. The currently operational TCP/IP internet is called *The Internet*.

7.7 Addressing, Routing & Casting

Addresses of nodes are needed to uniquely identify recipient nodes of messages. Switches use addresses to describe how to forward (to *route*) a message towards its destination. Depending on whether a source node (sender) sense a message to a single destination node (receiver), or to multiple or even all nodes, one speaks of **unicasting, multicasting** or **broadcasting**, respectively.

7.8 Reliability

7.8.1 Errors At The Bit Level

Corruption of single bits is very unlikely in copper cable (1:1,000,000) and optical fibre (1:10,000,000,000,000). The corruption often occurs in clusters (burst errors) and are caused by power surges, lightning or microwaves. This kind of corruption can be detected and sometimes corrected.

7.8.2 Errors At The Packet Level

This is when complete packets are lost and can be because of non-correctable bit errors. These errors also appear at congested switches because it can be difficult to distinguish between a packet that is indeed lost and one that is merely late in arriving at its destination.

7.8.3 Errors At The Node/Link Level:

- Due to a cut physical link
- Caused by hardware or software crashes
- Are eventually (after a *long* time) corrected
- It is difficult to distinguish between a cut and an error-prone link, or between a failed and slow node.

Note: Packet-switched networks have the abililty to possibly route around a failed node or link.

7.9 Common Units in Networking

- Units in computing are commonly shown as powers of 2 e.g. GB = 2^{30} bytes = 2^{33} bits.
- Units of engineering are commonly shwon as powers of 10. e.g. Gbps: giga bit per second = 10^9 or 2^{30} bits per second

7.10 Performance

The efficiency of applications running over the network often depends on the efficiency of the network itself. There are two key performance metrics:

- Bandwidth (*transmission speeds*) How many bytes per second can I move
- Latency (*transmission delays*) What is the time taken from beginning to end of a transfer

These can both be measured at the levels of individual links as well as end-to-end channels.

7.10.1 Bandwidth

Number of bits that can be transmitted over the link/channel in a certain period of time. Bandwidth is usually measured in bps (bits per second). Factors which influence bandwidth:

- The physical/electrical characteristics of links

- The software overhead needed for handling/transforming each bit of data (in the case of channels)
- The load of networks (in the case of channels)

7.10.2 Latency (Delay)

Latency in networks consists of three components:

1. Speed-of-light **propogation** delay (the speed of light is $2.3\times\ 10^8$ m/s in cable and $2.0\ \times\ 10^8$ m/s in fibre.
2. Time needed to **transmit** a unit of data (usually a packet).
3. **Queuing** delays inside the network (at switches).

Formula:

$$Latency = Propogation + Transmission + Queuing \tag{4}$$

$$Propogation = \frac{Distance}{SpeedOfLightInMedium} \tag{5}$$

$$Transmission = \frac{Size}{Bandwidth} \tag{6}$$

7.10.3 Latency Vs. Bandwidth

To which degree latency and bandwidth dominate performance depends on the application of the program. For example, a digital library program that is asked to fetch a 25MB image, with a 10Mbps channel needing about 20s for transmission means that other latency which is usually in the range of ms, does not matter. While a transatlantic network with a bandwidth of 45Mbps will need around 0.366ms for transmission of a 2KB message but the other latency which is in the range of 100ms dominates.

7.10.4 Delay × Bandwidth Product

How many bits can be sent before the first one arrives at the receiver?

- Answer: #bits = dela × bandwidth

The product also indicated how many bits can be *stored* in a link.

7.10.5 Application Performance Needs

Often applications want as much bandwidth as possible but sometimes other factors are more important. For example, the human eye can only process about 30 fps so bandwidth more than this is not useful. What matters is controlling jitter, i.e. the variation in latency (buffers storing incoming packets should not underflow or overflow). Lower and upper bounds on delays are needed.

8 Lecture 17: Network Stack

8.1 Network Requirements

A computer network must:

- Provide general, cost-effectivem, fair, robust and high-performance, connectivity among a large set of computers.
- Accommodate changes in the underlying technologies and in the demands of application programs. (e.g. developments in the ability to produce fibre-optic cable)

To cope with complexity, the principle of **abstraction** is usually applied in the sciences. In network design, abstraction is reflected in the concept of a **network architecture**.

8.2 The Principle of Abstraction

Identify a **model** capturing some important system aspect. Then encapsulate this model in an object such that is is **accessible** to other system components via its interface and its implementation details are hidden.

8.3 End-to-End Communication

Providing cost-effective connectivity among hosts is a necessary but too weak requirement to networks. What is needed is a means for application processes running on different hosts to communicate over the network.

Services for setting up, maintaining and using end-to-end connections are common for many application programs (implemented as part of the OS). Services should support commonly used communication patters such as:

- **Request-reply**: (For reliable exchange of messages between a client and a server e.g. a download).
- **Message Stream**: Potentially unreliable ordered sequence of messages (e.g. video on demand).

8.4 Network API

Interface that the OS (running on some node) provides to its networking subsystem. It is a piece of software that provides abstract routines for invoking common services of the network in a particular OS. Its implementation maps the offered abstract routines to the **services** provided by the network. For example, take UNIX sockets.

8.5 Network Architecture: Layering

Several steps of abstraction leads to **layering**. The services provided at the high-level (the more abstract) layers are implemented in terms of the services of the lower-level (the less abstract and more concrete) layers. There might be **multiple abstractions at a given layer** which is the case when considering different types of chennels.

8.5.1 Advantages & Disadvantages of Layering

Advantages:

- Obtaining manageable tasks by splitting a complex problem into multiple, simpler sub-problems.
- Achieving a modular design such that adding or altering a service affects only one layer, thus reusing the functionality of all other layers.

Disadvantages:

- Implementing a layered design might induce overheads and, thereby, reduce efficiency.

However the advantages by far outweigh this disadvantage which can usually be dealt with.

- **Note**: In network design, the objects making up the layers are the *protocols*.

8.6 Protocols: The Networking Software

The role of protocols in a layered architecture:

- Each layer n of some host may carry a conversation with layer n on some other host.
- The layer n protocol defines the rules used in this conversation.

Several layers of protocols build a **protocol stack**.

8.7 Services and Protocols

Service Interface on the other objects on the same host that want to use its communication services (such as send and receive primitives).

Protocol Interface to the counterparts (peers) on other machines, which defines a set of rules governing the messages that the protocol exchanges with its peers to implement the services.

8.8 Protocol Graphs

Protocol graphs visualise the dependencies between protocols at different layers (see slides for example).

8.9 OSI Reference Model (OSI Architecture)

The 7-layer **Open Systems Interconnection (OSI)** architecture is standardised by the International Standards Organisation (ISO). There are two sets of layers; one for end-hosts and another for switch nodes. The end host stack contains the following:

- Application
- Presentation
- Session
- Transport
- Network
- Data Link
- Physical

While the switch node stack only contains:

- Network
- Data Link
- Physical

8.10 Basic Purpose of ISO Layer

Physical Layer Handles transmission of raw bits over a physical link.

Data Link Layer Collects a stream of bits into a **frame** i.e. frames, not raw bits, are delivered to hosts. (It is usually implemented in network adaptors and in device drivers running on the node's OS.

Network Layer Deals with routing in packet-switched networks and employs the term **packet** rather than frame.

Transport Layer Implements process-to-process channels and employs the term **message** rather than packet or frame.

Session Layer Ties togethr different ttransport streams e.g. audio and video for video conferencing.

Presentation Layer Coodrinates the format of data exchanged between peers e.g. the length of integer represenations.

Application Layer Includes application protocols e.g. HTTP.

8.11 Internet Architecture (TCP/IP Architecture)

The Internet architecture only considers 4 layers of the OSI model; specifically, it leaves out the presentation and session layer and says that these should be dealt with in the application layer. The data link and physical layer are also replaced with a **host-to-network** layer which is what made it so popular since it expects so little of these layers.

8.12 Overview of the Internet Protocols

The Internet Protocol (IP) supports the interconnection of multiple network technologies into a single, logical network. There are two main rtansport protocols (end-to-end protocols):

- The **Transmission Control Protocol TCP** provides a reliably byte-stream channel.
- The **User Datagram Protocol (UDP)** provides an unreliable message delivery channel.

The Internet comes with many application protocols such as:

- File Transfer Protocol (FTP)
- HyperText Transfer Protocol (HTTP)
- Directory Name Service (DNS)
- Real-time Transfer Protocol (RTP)

8.13 OSI vs. Internet Architecture

The Internet protocols were invented before the OSI model, and the Internet architecture was just a result of an existing implementation. The Internet protocols became the de-facto standard since they were shipped with the popular Berkeley distribution of UNIX, which also helped their further development. The conceptual OSI reference model, however, is a great use for conceptually discussing and teaching about computer networks.

8.14 Network Standardisation

- International Telecommunication Union (ITU): Telecom sector includes data communications systems.
- International Standards Organisation (ISO): Member of the ITU.
- Institute of Electrical and Electronics Engineers (IEEE): Standards are occasionally adapted by ISO.
- Internet Society:

- Internet Engineering Task Force (IETF)
- Internet Research Task Force (IRTF)

Generally protocols are submitted to different organisations depending on their level in the stack. They are submitted by engineers (often funded by their employer) who try to create vendor-independent and non-ambiguous rules.

9 Lecture 18: Physical & Data Link Layers

9.1 Physical Layer

Deals with the transmission of bits over physical medium (twisted pair/coaxial cables, fibre optics, wireless etc.). The layers usually have a standardised interface to transmission media (e.g. how 1/0 signals are carried over a link).

Requirements of the physical layer include both **synchronisation** and **flow control** as well as multiplexing (FDM (frequency division multiplexing), TDM (time division multiplexing) and CDM (code division multiplexing)). Multiplexing is where you use the same connection for multiple transmissions.

9.2 Data Link Layer

Deals with the framing of raw data, as well as flow control and error correction (i.e. if they cannot be solved at the physical level). The data linke layer is often divided into two sublayers:

- **Logical Link Control (LLC)**: Multiplex protocols running over the data link layer, error and flow control.
- **Media Access Control (MAC)**: Control channel access, append and check FCS (frame chack sequence), discard malformed frames, addressing.

9.2.1 Example: Ethernet

- Standard: IEEE 802.3

Uses CSMA/CD local area network technology:

- Multiple-Access (MA): Several nodes are connected to the same cable (cf. data bus).
- Carrier Sense (CS): A node can distinguish between a busy and an idle link.
- Collision Detect (CD): A node listens when it transmits a frame in order to detect whether the frame interferes (collides) with a frame transmitted by another node.

Multiple Ethernet segments are joined by **repeaters** that forward the signals. In an Ethernet, every signal is propagated in all directions over the entire network, even crossing repeater boundaries.

- Problem: All hosts compete for access to the same link; they are said to be in the same **collision detection**.

The problem is solved by intelligently partitioning the **collision domain** using the following:

Hub Multiway repeater supporting several point-to-point segments - still only one collision domain.

Bridge Each port is connected to a different collision domain. Transmissions within separate domains are allowed to happen in parallel.

Switch Frames are sent only to their destinations — no collisions.

How can we know that a frame is going between certain nodes?

Each Ethernet adaptor has a unique Ethernet (MAC) address, which is 6 bytes long and read as a sequence of six numbers, each given as a pair of hexadecimal digits (e.g. 08:00:2B:E4:B1:02). Each connected adaptor receives all frames transmitted over an Ethernet, but passes to the upper protocol layers only those matching:

- The adaptor's own address (unicase)
- The Ethernet broadcast address (FF:FF:FF:FF:FF:FF).
- Multicase addresses (not used in practice just in the standard) (least significant bit of the first byte set).

See lecture for frame format example.

Other Data Link Layer protocols include:

- IEEE 802.11 - Wireless Lan
- IEEE 802.16 - Wireless Broadband (e.g. WiMax)
- IEEE 802.15.4 - Low-rate Personal Area Networks (e.g. ZigBee, WirelessHart)

9.3 Learning Bridges

How does a bridge know to which port to forward a packet, i.e. on which port a destination host resides?

A bridge inspects the source address of each received frame and notes the number of the incoming port. The pair </port number/, /source address/> is added to the bridge's forwarding table. If there exists no entry for a destination address (yet), the packet is forwarded to all ports, except the packet's incoming port.

Learning algorithm:

1. The forwarding table is empty when the bridge boots.
2. The forwarding table it updated according to the above scheme.
3. Table entries are discarded after some amount of time to protect against the possibility of LAN addresses changing segments.

The table is updated when messages are sent as even though they cannot know which domain the receiver is in, they can know which the sender is.

9.4 Problems with Loops in Extended LANs

Loops in extended LANs:

- Provide **redundancy** in case of failures of links/bridges.
- Potentially enable frames to **loop forever**.

To stop them looping forever, a **distributed spanning tree algorithm** is run, which selects, for each bridge, the ports over which it should forward frames such that loops are avoided.

9.4.1 Spanning Tree

Think of an extended LAN (with loops) as a graph (with cycles). The **spanning tree** of a graph is a **subgraph** that emerges from the original graph by leaving out edges such that no cycles/loops remain. There is a spanning tree algorithm included in the IEEE 802.1 specification for LAN bridges. Each bridge decides the ports over which it is (not) willing to forward frames.

9.4.2 Distributed Spanning Tree Algorithm

Basic Idea

1. The algorithm first elects the bridge with the smallest id (address) as the **root** of the spanning tree.
2. Each bridge computes the **shortest path** to the root and notes its ports that lie on this path (*preferred port towards root*)/
3. Each LAN elects a single designated bridge (one of the closest to the root — smallest id wins in tie) which is made responsible for forwarding frames towards the root bridge.

Afterwards each bridge just forwards frames over those ports (i.e. to those LANs) for which it is the designated bridge.

Implemented by bridges exchanging configuration messages with each other; these messages include:

id For the bridge that the sending bridge believes to be the **root**.

distance (measured in hops) from the sending to the root bridge.

id for the bridge sending the message.

Each bridge:

- Initially thinks that it is the root and sends over all of its ports the message (id,0,id), where id is the bridge's identifier.
- May receive a message over one of its ports and checks whether:
  - It identifies a root with a smaller id.
  - It identifies a root with an equal id but with shorter distance.
  - Root id and distance are equal but the sending bridge has a smaller id.
- If so, it adds 1 to the distance, saves this info & discards old info.

See slides ror example.

10 Lecture 19: Network Layer & Internet Protocol

10.1 Network Layer Outline

The network layer is responsible for packet forwarding and routing through intermediate routers.

10.2 Packet Forwarding

Send an incoming or previously buffered packet to the appropriate output port. Consult an identifier in the packet header and interpret it with respect to the:

- Connectionless (datagram) Approach
- Connection-Oriented Approach
- Source-Routed Approach

We will make a couple of assumptions:

- We identify nodes via globally unique addresses (such as Ethernet addresses).
- Each input and output port of a switch is given a unique number (relative to the considered switch).

10.2.1 Connectionless (Datagram) Approach

Every packet contains the full destination address. The decision of how to forward packets is made by a **forwarding table** (**routing table**).

Advantages:

- A host can send a packet anywhere anytime, i.e. all packets can immediately be forwarded by consulting the forwarding table.
- A switch or link failure must not have serious consequences as long as one may route around the failure (by modifying forwarding tables accordingly).

Disadvantages:

- A sending host does not know whether the network is capable of delivering a packet or whether the destination host is even up.
- Each packet is forwarded independently of other packets means that packets may be sent via different routs and, thus may reach the destination out of order.

10.3 Connection-Oriented Approach

Uses the concept of a *virtual circuit(VC), which requires that first a virtual connection from the source to the host is set up before any data can be transferred. Virtual circuits can be set up in different ways:

- Permanently/statically by the system administrator, leading to **permanent virtual circuits (PVC)**.
- Temporarily/dynamically by the sending host via sending appropriate messages into the network (signalling); this leads to **switched virtual circuits (SVC)**.

The second option is by far the most widely used out of the two.

10.3.1 Establishing VCs

Each switch needs to keep the following information **for each VC** (connection) in a **virtual circuit table**:

- An **incoming interface** (port) on which packets for this VC arrive.
- A **virtual circuit identifier (VCI)** that will be carried in the header of arriving packets.
- An **outgoing interface** on which packets for this VC leave.
- A VCI that will be used for outgoing packets.

Important remarks:

- The VCI is **not global**; it has significance only on a given link.
- When setting up a VC, the network administrator picks VCI that are currently unused.

Disadvantages:

- There is at least 1 RTT (round-trip delay) of delay before any data can be sent.
- A switch/link failure leads to a broken connection.
- Released/broken virtual circuits need to be torn down.

Advantages:

- Each data packet does not need to include the full address of the receiver, but just a small identifier (a VCI) that is unique relative to each link (the overhead caused by headers is reduced).
- When a virtual circuit is established, the sender knows that there is a route to the receiver, the receiver is willing and able to receive and there are enough resources along the route.

See slide 15 for very useful comparison table.

10.3.2 Source Routing

All information about network topology that is required to switch a packet across the network is provided by the source host.

1. Include an **ordered list of port numbers** in a packet's header.
2. Each switch forwards the packet to the port determined by the number at the front of the list.
3. Before forwarding, a switch must:
   - **Strip** the front number from the packet, or
   - **Rotate** the ordered list such that the next port number comes to the front. (At the last switch, the received list is then identical to the list originally sent by the sending host).

Disadvantages:

- Every host needs to know many details of the network's topology in order to be able to construct a packet header.
  - Similar to the problem of building forwarding tables in a datagram network, or determining how to route a setup messsage in a VC network.
  - Suffers from a scaling problem since getting complete path information is very hard in reasonably large networks.
- Headers have a variable size, probably with no upper bound.

Source routing is used in:

- Virtual circuit networks as a means for getting the initial request from the sending host to the destination host.
- Embedded systems and PANs (Personal Area Networks) where the topology is simple and unlikely to change.
- In the Internet protocol as an option (a *datagram* protocol).

10.4 Internetworking

Internetwork An arbitrary **collection of networks** using different technologies, which are interconnected via **routers** (**gateways**) to provide a host-to-host packet delivery service, i.e. an internetwork is a **logical network**.

The underlying networks, each based on a single technology, are often called **physical networks**, which might contain collections of Ethernets connects by bridges or switches.

*Simply put, an **internetwork** is a network of networks.*

The Internet Protocol glues the single network together, yielding a large, logical and heterogeneous network.

10.5 IP Service Model

The IP **service model** defines the host-to-host services which an internetwork should provide. Philosphy:

- The model is **undemanding** enough such that any existing and hopefully any future network technology is able to provide the services.
- The protocol assumes a **best-effort, connectionless service** of the underlying physical networks.
- Therefore it runs on virtually any network.

10.6 IP Addressing

To identify all hosts in an internetwork, a global addressing scheme is needed, giving each node a unique address. Problem:

- Ethernet addresses are flat, i.e. they have no structure that provides forwarding information to routing protocols.

Solution:

- IP addresses are hierarchical, reflecting the hierarchy of an internetwork.
  - `IP Address = <network part, host part>`

Each host is assigned an IP address; similary, every interface/port of a router is assigned an IP address.

11 Lecture 20: Network Layer & Internet Protocol Continued

11.1 Intra-Domain Routing

Routing is needed for forwarding packets in a datagram (connectionless) network, or for establishing virtual circuits in a VC (connection oriented) network. Routing algorithms or protocols create routing tables from which one may derive the neccesary forwarding tables. These in turn, define the output port through which a packet will be forwarded.

Most routing protocols work only for 10s or 100s of nodes and hence they are referred to as **interior gateway protocols (IGPs)** or (**intra-domain routing protocols**). To make them scale, internetworks employ a hierarchical routing structure based on **domains**.

- A **domain** is an internetwork where all routers are under a single administritative entity (e.g. university campus).
- Each domain uses IGPs to route packages within its boundaries and uses gateway routes to forward packets to other domains (inter-domain routing).

11.1.1 Graph Representation of Routing

Routing is a graph-theoretic problem and requires one to calculate the lowest-cost path between two nodes.

- Nodes are hosts, switches, routers or networks
- Edges are network links, each associated with a cost.
- Cost of a path is the sum of the costs of all traversed edges.

There are two main types of algorithms for solving the problem:

- Global routing: all routers have complete topology and link cost info — "link state" algorithms.
- Decentralised routing: router knows link costs to neighbours — "distance vector" algorithms.

11.1.2 Link State Routing Algorithm

Dijkstra's Shortest Path Algorithm is based on a link state broadcast, aiming to provide all nodes with the same information. After k iterations, the algorithm knows the least cost path to k destinations. Notation is as follows:

c(x,y) Link cost from node x to y, c = ∞ if not direct neighbours

D(V) Current value of cost of path from source to destination v.

p(v) Predecessor node along path from source to v.

N' Set of nodes whose least cost path is definitively known.

11.1.3 Distance-Vector Routing Algorithm

One classic example is the **Bellman-Ford routing algorithm** (1957) and **Ford-Fulkerson algorithm** (1962). It is a **dynamic routing algorithm** and was used in the Internet Protocol under the name RIP (Routing Information Protocol).

Each node stores an array (a **vector**) containing the currently believed distances to all other nodes. Initially this distance is ∞ for all nodes except the considered node's immediate neighbours. Vectors are distributed to a node's immediate neigbours.

11.1.4 Dimension-Order Routing

Some network configurations can be vulnerable to deadlock. Deadlock arises when a cyclic resources dependency occurs and the messages become blocked forever. Dimension-order routing can avoid deadlock by removing one of the conditions: circular dependency.

- XY Routing:
  - Forbid any Y to X turn
  - It is deterministic
- West-first Routing:
  - Forbid the west turns only
  - Partially adaptive

11.2 Inter-Domain Routing

11.2.1 Border Gateway Protocol (BGP)

BGP is the de facto inter-domain routing protocol for the Internet. It supports route coordination across domains, referred as **"autonomous systems" (AS)**. BGP provides routers a means to:

e(xternal)BGP Obtain subnet reachability information from neighbouring ASs.

i(nternal)BGP Propogate reachability information to all AS-internal routers.

Determine *good* routes to other networks based on reachability information and policy.

See lecture slides for example.

12 Lecture 21: Transport Layer

12.1 Introduction

The transport layer provides **end-to-end connectivity** in terms of a **transport protocol** (end-to-end protocol). The underlying network layer usually only provides a **best-effort host-to-host service** (e.g. IP):

- messages are dropped (due to congestion)
- messages are re-ordered
- messages are delivered several times (problem of duplicates)
- messages are limited to some finite size
- messages are delivered after some long delay

Different transport protocols address (some of) these limitations by offering different services:

- Simple (application) demultiplexing service (**User Datagram Protocol**)
- Reliable Byte-Stream Service (**Transmission Control Protocol**)

13 Lecture 3: Processes

13.1 Processes and Programs

- A **program** is an ordered set of specific operations for a computer to perform.
- A **process** is an abstraction of a running program.

A program is a passive entity stored on the disk including an executable file while a process is active and loaded into memory.

13.2 Process Management

Process management is one of the key functions of the OS, it includes:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Deciding which process should run
- Providing mechanisms for process synchronisation
- Providing mechanisms for process communication
- Provising mechanisms for deadlock handling

13.3 Process Structure

A process is more than the program code, which is sometimes known as the text section. It also includes the current activity such as the calue of the program counter and contents of the processor's registers. It also includes the process stack, which contains temporary data (such as function parameters, return addresses and local variables) and a data section which contains global variables. It may also include a heap, which is memory that is dynamically allocated during process run time.

See lecture slides for the process layout in memory. In particular, slide 9 on the **Process Control Block**

13.4 Process States

One thing stored in the PCB is the process state. Most OSs contain states which are similar or equivilent to the following:

- **Ready (or Runnable)** — scheduled to run but not currently using the processor
- **Running** — currently using the processor
- **Waiting (or Blocked)** — suspended waiting for some event to occur (such as a periodic clock tick or printer device to complete printing)
- **Terminated** — the process has finished execution (and will be removed from the system ASAP

See slide 11 for a useful flow diagram of states.

13.5 Process Lifecycle

- Processes are created
  - at system initialisation
  - by another process
  - by a user (e.g double clicking an icon)
  - batch job
- On creation, OS will:
  - allocate memory
  - create PCB
  - load program from disk to memory
  - copy arguments to program's stack and itialise registers

There are resource sharing options such as:

- Parent and children process share all resources
- Children share subset of parent's resources
- Parent and child share no resources

And execution options:

- Parent and children execute concurrently
- Parent waits until children terminate

13.5.1 Ready and Running

When a process is created it is places into a ready queue. The scheduler decides which process to put into running state. The **policy** makes the decision itself and the **mechanism** uses the information in the PCB to return to the selected process.

13.5.2 Waiting/Blocked

Processes issuing I/O requests are placed in a waiting state. When the I/O completes, the process returns to the ready state. If all processes are waiting, then the processor is idle until one of the processes becomes unblocked and ready. This should be avoided at all costs.

13.5.3 Terminated

There are several different reasons a process can be moved into a terminated state:

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary)
- Killed by another process (involuntary)

13.5.4 Context Switch

When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch. Context of a process is represented in the PCB.

Context-switch time is pure overhead. The system does no useful work while switching. The more complex the OS and the PCB, the longer the context switch.

Switching is also dependent on hardware support. Some hardware proved multiple sets of registers per CPU which means that there can be multiple contexts loaded at once.

See slide 18 for a context switch diagram

14 Lecture 4: Threads & Concurrency

14.1 Concurrency

14.1.1 Multiple Processes

Applications are often constructed from multiple co-operating programs.

For example, web browsers used to run as a single process, therefore if oone site causes trouble, the entire browser can hang or crash. Google Chrome is multi-process with 3 different types of processes:

- Browser process manages user interface, disk and network I/O
- Renderer process renders web pages, deals with HTML and js (with a new renderer created for each open website)
- Each type of plugin has a separate process

14.1.2 Motivation

- **Responsiveness** — program may continue execution even if a process if blocked, especially important for user interfaces.
- **Modularity** — it may be easier to create a program as a set of interacting processes.
- **Scalability** — process can take advantage of multiprocesssor architectures.

14.1.3 Multiprocessor Systems

Multi-CPU Systems — Multiple CPUs are placed in the computer to provide more computing performance.

Multicore Systems — Multiple computing cores are places on a single processing chip where each core appears as a separate CPU to the operating system.

Concurrent programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency. On a system with a sigle computing core, concurrency means that the execution of the processes will be interleaved over time. But on a system with multiplr cores, concurrency means that some processes can run in parallel, because the system can assign a separate process to each core.

14.1.4 Concurrency vs. Parallelism

There is a fine but clear distinction between concurrency and parallelism. A **concurrent** system supports more than one task by allowing all the tasks to make progress. In contrast, a system is **parallel** if it can perform more than one task simultaneously. Thus, it is possible to have concurrency without parallelism. Concurrency within a program is an opportunity for parallelism if there is parallel hardware to exploit.

Types of parallelism:

- Data parallelism — distributes subsets of the same data across multiple cores, same operation on each.
- Task parallellism — distributing threads across cores, each thread performing unique operation.

14.1.5 Amdahl's Law

Amdahl's law indentifies performance gains from adding additional cores to an application that has both serial and parallel components:

Where N is the number of processing cores and S is the serial portion:

$$speedup \leq \frac{1}{S + \frac{1-S}{N}}$$

That is, if an application is 75% parallel and 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times. As N approaches infinity, speedup approaches $\frac{1}{S}$. As S approaches 0, speedup approaches N. Serial portion of an application has disproportionate effect on performance gained by adding additional cores.

14.2 Threads

14.2.1 Intro

Multiple concurrent processes are conceptually equivalent to multiple independent programs (each has a separate address space and requires inter-process communications). **Threads** are concurrent units within a process. All threads of a process share the same address space, low overhead in thread creation. Benefits from shared-memory processors, low-inter thread communication overhead.

14.2.2 User & Kernel Threads

Support for threads may be provided at two different levels:

- **User threads** — are supported above the kernel and are managed without kernel support, primarily by user-level threads library.
- **Kernel threads** — are supported and managed directly by the operating system.

Virtually all contempory systems support kernel threads.

Many-to-One Relationship Many user-level threads mapped to single kernel thread. One thread blocking causes all to block. Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time. Few systems still use this model.

One-to-One Relationship Each user-level thread maps to a single kernel thread. Creating a user-level thread creates a kernel thread. More concurrency than many-to-one. Number of threads per process sometimes restricted due to overhead. Examples:

- Windows 95-XP
- Linux

Many-to-Many Relationship Allows many user level threads to be mapped to many kernel threads. Allows the operating system to create a sufficient number of kernel threads. Examples:

- Solaris prior to version 9
- Windows with the *ThreadFiber* package

14.2.3 Thread Libraries

Thread library provides programmer with an API for creating and managing threads. Two primary ways of implementing:

- Library entirely in user space
- Kernel-level library supported by the OS

Widely used thread libraries:

- POSIX Pthreads
- Windows threads
- Java threads

14.3 Threading Issues

14.3.1 Thread Cancellation

Thread cancellation means terminating a thread before it has finished working. There can be two approaches to this:

- **Asynchronous cancellation** — terminates the target thread immediately. (Immediate and responsive but more dangerous)
- **Deferred cancellation** — allows the target thread to periodically check if it should be cancelled.

14.3.2 Signal Handling

Signals are used in UNIX systms to notify a process that a particular event has occured. When a multithreaded process receives a signal, OS can be set to deliver it to all or to specific thread(s).

14.3.3 fork() System Call

If one thread of a process calls `fork()`, will the entire process be duplicated, or only the calling thread?

Some versions of UNIX have two versions of `fork()` and the choice between them often depends on whether `exec()` will be called afterwards.

`exec()` usually works as normal — replace the running process including all threads.

14.3.4 Security and Integrity Issues

There are many security and integrity isuues because of the sharing of resources between multiple threads. This will be covered in greater detail later in the module.

15 Lecture 5: CPU Scheduling

15.1 Introduction

Most processes exhibit the following behaviour:

- CPU burst followed by I/O burst.
- Process execution consists of a cycle of CPU execution and I/O wait.

Maximum CPU utilisation obtained with multiprogramming. This is where one process uses the CPU while the other waits for I/O. The CPU burst usually takes less than 8ms.

Whenever the CPU becomes idle, OS selects one of the processes in the ready queue to be executed. The selection process is carried out by the CPU scheduler. The ready queue may be ordered in various ways. CPU scheduling decisions may take place when a process:

- switches from running state to waiting state
- switches from running state to ready state
- switches from waiting state to ready state
- terminates

15.2 Dispatcher

The dispatcher switches context, switches to user mode, jumps to the proper location in the user program to restart that progmram. This brings **dispatch latency** (overhead) — the time it takes for the dispatcher to stop one process and start another running. This latency needs to be minimised as occurs frequently.

15.3 Scheduling Algorithms

Existing algorithms aim to satisfy the following criteria:

- **Max CPU utilisation** — keep the CPU as busy as possible.
- **Max throughput** — number of processesthat complete their execution per time unit.
- **Min turnaround time** — amount of time to execute a particular process.
- **Min waiting time** — total amount of time a process has been waiting in the ready queue.
- **Min response time** — amount of time it takes from when a requestwas submitted until the first response is produced (for time-sharing environment).
- **Fairness** — each process should get its fair share of the CPU.

There are three main families of scheduling algorithms:

- **Non-preemptive** — Process runs to completion, OS schedules next process upon termination.
- **Preemptive** — Process runs until suspended by OS. Preemption can be time or event triggered.
- **Cooperative** — Process explicitly relinquishes control and allows OS to make a scheduling decision.

15.4 Classic Scheduling Algorithms

The three classic sheduling algorithms are:

- FCFS (or FIFO) — first come, first served. **Non-Preemptive**
- Round-Robin (RR) — aims for fairness. **Preemptive**
- Priority Scheduling — **Preemptive or Non-Preemptive**

Or it is possible to have combinations of the above.

15.4.1 FCFS

Easy to implement and maintain but can give a wide variety of average wait times.

15.4.2 RR

Each process is assigned a *quantum* when it enters the CPU (a quantum is a time slot using the CPU). If the process is still running at the end of the quantum then preempt it:

- Start a context switch
- The process is put at the back of the "runnable queue"

Process may become no longer runnable **before** the end of the quantum e.g I/O operation. In which case, do as though it was the end of the quantum.

Overall Round-Robin ensures fairness, but does not reflect relative importance of processes.

15.4.3 Priority Scheduling

Not all processes are equally important, so priority scheduling assigns a priority for each of them.

Note: Priority values vary between books, courses etc. In this course, higher priority processes have lower priority numbers (e.g 1 or 0 = highest priority).

In the preemptive version, the runnable process with the highest priority is allowed to run immediately. Otherwise, the highest priority process may have to wait for an ongoing process to finish. This algorithm has one large problem and that is that the highest priority process may run indefinitely or to completion and hence, may lead to starvation for other processes. Therefore the OS should have a way of identifying and recovering from such scenarios (e.g aging — increase the priority of the process as time progresses).

15.4.4 Comparison of Classic Algorithms

	Pros	Cons
FCFS	Simple & Low overheads	Bad turnaround for short processes
Round-Robin	Good turnaround for short processes	High Overhead
Priorities	Predictability for high priority tasks	May delay lower priority tasks for long periods

15.4.5 Bonus Algorithm SJF

Shortest-Job-First associates the length of each process' next CPU burst with the process itself. Then we use these lengths to schedule the process with the shortest time. This is non-preemptive. SJF is optimal in that it gives the minimum average waiting time for a given set of processes (assuming that the estimation of the next CPU burst time is correct.

Why don't we always use SJF? The burst sizes can be unpredictable, they may be dependent on input sizes.

16 Lecture 6: CPU Scheduling Continued

16.1 More algorithms...

16.1.1 Shortest-Remaining-Time-First

Preemptive version of SJF.

16.1.2 Earliest Deadline First (EDF)

Each process must declare a deadline, the algorithm then runs the most urgent process first. Implemented as a variable priority scheduling algorithm.

16.2 Multilevel Queue

In most operating systems, more than one algorithm is used at one time. To enable this, **multilevel queues** are used. Often they are separated into **foreground**(interactive) and **background** (batch). For example:

- Foreground: RR
- Background: FCFS

This example means that the user will see the snappiest responses to the foreground services.

Scheduling has to be done between the queues. Fixed priority scheduling could be used (i.e serve all from the foreground then from the background). Or *time slice*, where each queue gets a certain amount of CPU time, e.g. 80% to forground and 20% to background.

16.3 Multilevel Feedback Queue

A more complicated version of the multilevel queue. In a **multilevel feedback queue** a process can move between the various queues. This can be used to control the *aging* of processes. Multilevel feedback queue schedulers are defined by the following parameters:

- Number of queues
- Scheduling algorithms for each queue
- Method used to determine when to upgrade a process
- Method used to determine when to demote a process
- Method used to determine which queue a process will enter when that process needs service

16.4 Thread Scheduling

16.4.1 Kernel Threads

OS knows about all processes and threads, so makes scheduling decision across all processes and threads.

16.4.2 User Threads

OS decides which process to run. OS doesn't know whether a process contains threads (if that process contains a user threads implementatin, decisions regarding scheduling of the user threads are taken there). Essentially hierarchical scheduling: when a process is run, the thread scheduler within the process (managing the user threads) willl decide which thread to run.

16.5 Multiprocessor Scheduling

CPU scheduling is more complex when there are multiple processors (cores) available).

16.5.1 Homogenous processors

- **Asymmetric multiprocessing**: only one processor accesses the system data structures, alleviating the need for data sharing.
- **Symmetric multiprocessing (SMP)**: each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes.

Given a set of runnable processes, now it has to be decided which process to dispatch and which CPU should run it. Some processes may have a **processor affinity** meaning they have an affinity for the processor on which it will run.

- Soft affinity $\implies$ preference
- Hard affinity $\implies$ requirement
- Variations including processor sets (e.g with affinity masks where a process will use a bitmask to show which sets of processes it will run on).

16.5.2 Non-Uniform Memory Access (NUMA)

NUMA is one of the reasons for implementing processor affinity. Sometimes certain processors will have faster access to certain memory blocks than other processors. Naturally if a process uses data within one of the processor-specific blocks, it would have a greate affinity for the respective processor.

16.5.3 Load Balancing in Multiprocessors

If SMP, need to keep all CPUs loaded for efficiency.

- **Load balancing** attempts to keep workload evenly distributed.
- **Push migration** is a periodic task which checks the load on each processor and if it finds the processor is overloaded, it pushes tasks from the overloaded CPU to other CPUs.
- **Pull migration** is when idle processors pull waiting tasks from busy processors.

17 Lecture 7: Synchronisation

17.1 Coordination

The OS must support multiple applications/processes (threads working together). This is formally known as **Inter-Process Communication** (IPC). It is achieved through two main concepts: shared memory and message passing.

Shared memory requires synchronisation to ensure data is consistent. For example, if a data structure is being updated then a reader will read a consistent view of that data (i.e. doesn't read partial updates).

Some programming lanuages make synchronisation primitives available to programmers. Languages rely on atomic operations when accessing shared data. However, atomic operations at the language level are often implemented at multiple instruction at the CPU level and interrupts can occur between instructions.

17.1.1 Concurrent Access to Shared Data

Race conditions are when many processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last. This can mean that data may become inconsistent. Therefore, the ability to execute an instruction, or a number of instructions atomically is crucial for the correct operation of synchronisation primitives provided by programming languages.

17.1.2 Critical Section Problem

Every process which could have a race condition has a **critical section**. To overcome the problem, one general approach is to only allow one process into the critical section at a time.

One general protocol for solving the problem is outlined as follows:

- Each process must ask permission to enter the critical section in what is known as **entry section** code.
- It then executes in the critical section.
- Once it finishes executing in the critical section it enters the **exit section** code.
- The process then enters the **remainder section** code.

```
do {
    entrySection
        criticalSection
    exitSection
        remainderSection
};
```

17.1.3 Solutions to the Critical Section Problem

Requirements to Fulfill Solutions to the critical section problem must satisfy the following three requirements:

Mutual Exclusion If process $\mathbf{P_i}$ is executing in its critical section, then no other processes can be executing in their critical sectinos.

Progress Processes waiting to enter a critical section cannot block processes inside; only those processes that are not executing in their remainder sections can participate in making the decision as to which process will enter its critical section next.

Bounded Waiting A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Disabling Interrupts One way to satisfy these requirements is to disable interrupts when a process is in the entry section and reenable them when it is in the exit secton. However this is very impractical if the critical section code takes a long time to execute as it becomes impossible to preempt. This means that any part of a preemptive scheduling algorithm becomes redundant.

Peterson's Algorithm

```
int turn; // indicates whose turn it is to enter the critical section if both are ready
bool flag[2]; // indicates if a process is ready to enter the critical section
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
        criticalSection
    flag[i] = false;
        remainderSection
};
```

This algorithm assumes read and store operations are atomic (in checking the conditions of the while loop). It also loops while waiting to enter the critical section (also called **spin lock**) - busy waiting, wastes CPU. This implementation works for two processes only, even though it is extendable.

Mutual Exclusion - Hardware Support Modern CPUs provide special instructions to support exclusive access to shared variables. e.g test-and-set instruction:

```
bool test_and_set (bool *target)
    {
```

```
        bool rv = *target;
        *target = TRUE;
        return rv;
    }
```

With this we can use the instruction to enforce mutual exclusion. We have a shared boolean variable `lock`, initialised to FALSE. Then each process wishing to execute critical section code does the following:

```
do {
    while (test_and_set(&lock)); //do nothing
        criticalSection;
    lock = false;
        remainderSection;
}
```

The process keeps looping unless `lock == false`. This still uses a spin lock which is still wasteful. Actual implementations put the process in a waiting queue.

17.2 Semaphores

A semaphore is a programming mechanism used to achieve synchronisation and mutual exclusion. They are based on mutual exclusion services provided by the OS. They ensure atomic execution and rely on hardware support (e.g the `test_and_set` instruction). Semaphores are accessed through system calls. They are integer variables used as a *flag* and the atomic code that increments or decrements it. There are two types of semaphore:

Binary Semaphores (mutex) Integer values can range only between 0 and 1.

Counting Semaphores:: Integer value can range over an unrestriceted domain.

17.2.1 Semaphore Operations

Semaphores have two indivisible operations: **wait** (**P**) and **signal** (**V**).

`wait`

1. Decrement semaphore
2. If semaphore value is (or became) negative, block the process and add it to a waiting queue.

`signal`

1. Increment semaphore
2. If semaphore is less than or equal to zero, process waiting at the head of the queue is awakened.
3. If semaphore is greater than zero, it means that no process is waiting, no immediate action needed.

18 Lecture 8: Deadlock

18.1 Introduction

Systems consist of resources:

- Resource types R_1, R_2, ..., R_m
- CPU cycles, memory space, I/O devices

Each resource type R_i has W_i instances. Each process utilises a resource using the following actions:

- Request
- Use
- Release

A set of processes is deadlocked if each process in the set is waiting for an action that only another process in the set can cause.

Note: A resource is considered **preemptable** if it can be taken away from the process owning it (at any time) with no ill effects.

18.2 Conditions for Deadlock

Deadlock can arise if the following four conditions hold simultaneously:

1. **Mutual Exclusion**: Only one process at a time can use a resource.
2. **Hold and Wait**: A process holding at least one resource is waiting to acquire additional resources held by other processes.
3. **No Preemption**: A resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular Wait**: Circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

If one of these conditions is absent, no deadlock is possible.

18.3 Resource Allocation Graphs

In resource allocation graphs we have nodes to represent processes as well as nodes to represent resources. See lecture slides for examples.

18.4 Methods for Handling Deadlocks

Ensuring that the system will never enter a deadlock state is usually done using one of two methods:

- **Deadlock Prevention** (18.4.1)
- **Deadlock Avoidance** (18.4.2)

Or we can allow the system to enter a deadlock state and then recover afterwards using **Deadlock Recovery**. The final method is to ignore the problem and pretend that deadlocks never occur in the system (used by most OSs, including UNIX, Linux & Windows).

18.4.1 Deadlock Prevention

Ensure that at least one of the necessary conditions for deadlocks does not hold. This can be accomplished by restraining the ways a request can be made:

Mutual Exclusion Avoid mutually-exclusive access to resources. This is impractical as most systems have inherently non-sharable resources that cannot be accessed simultaneously by various processes.

Hold and Wait Must guarantee that whenever a process requests a resource, it does not hold any other resources. This requires processes to request and be allocated all its resources only when the process has none allocated to it. This may result in low resource utilisation and possibly starvation.

Therefore, both of these are impractical to try and prevent.

No Preemption While also impractical, resource preemption may be imposed as follows:

- If a process that is holding some resources requests another resource than cannot be immediately allocated to it, then all resources currently being held are released.
- Preempted resources are added to the list of resources for which the process is waiting.
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Circular Wait Impose a total ordering of all types.

- Require that each process requests resources in an increasing order of enumeration.

18.4.2 Deadlock Avoidance

Deadlock avoidance is when you ensure that the system will never enter a deadlock state. This requires that the system has some additional **a priori** information available on possible requests.

The simplest and most useful model requires that each process declare the **maximum number** of resources of each type that it may need. Deadlock-aoidance algorithms dynamixally examie the resource-allocation state to ensure that there cam never be a circular-wait condition. Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe States Safe states are the foundation of every deadlock-avoidance algorithm. The system is said to be in a **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of all of the processes in the systems such that for each P_i the resources that P_i can still request can be satisfied by currently available resources and resources held by all the P_j, with $j<i$.

That is:

1. If a P_i resource needs are not immediately available, then P_i can wait until all processes P_j ($j<i$) have finished executing.
2. When they have finished executing they release all their resources and then P_i can obtain the needed resources, execute, return allocated resources and terminate.
3. When P_i terminates, $P_{(i+1)}$ can obtain its needed resources and so on.

- If a system is in a **safe** state $\Rightarrow$ no deadlocks
- If a system is in an **unsafe** state $\Rightarrow$ possibility of deadlocks

Deadlock Avoidance Algorithms If we have a single instance of a resource type; we can use a variant of the resource-allocation graph. (1) Or if we have multiple instances of a resource type: we can use the *banker's algorithm* (2)

1. Resource-Allocation Graph Algorithm Single instance of a resource type. Each process must a priori claim maximum resource use. Use a variant of the resource-allocation graph with claim edges. **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i **may** request resource R_j, represented by a dashed line. Claim edge converts to request edge when a process requests a resource. Request edge converted to an assignment edge when tthe resource is allocated the process. When a resource is released by a process, assignment edge reconverts to a claim edge. Resources must be a claimed a priori in the system. A cycle in the graph implies that the system is in an unsafe state.

2. Banker's Algorithm Multiple instances of a resource type. (Created by Edsger Dijkstra) Each process must a priori claim maximum use. When a process requests a resource it may have to wait. When a process gets all of its resources it must return them in a finite amount of time. The algorithm uses the analogy of an interest-free bank where:

   - A customer establishes a line of credit.
   - Borrows money in chunks that together never exceed the total line of credit.
   - Once it reaches the maximum, the customer must pay back in a finite amount of time.

   To implement the Banker's Algorithm, we use the following data structures where n = number of processes and m = number of resource types:

   Available Vector of length m

   - If `Available[j] = k`, then there are k instances of resource type R_j available.

   Max nm matrix

   - If `Max[i,j] = k`, then process P_i may request at most k instances of resource type R_j.

   Allocation nm matrix:

   - If `Allocation[i,j] = k`, then P_i is currently allocated k instances of R_j.

   Need nm matrix:

   - If `Need[i,j] = k`, then P_i may need at most k more instances of R_j to complete its task

   Therefore: `Need[i,j] = Max[i,j] - Allocation[i,j]`

   The **safety algorithm** is part of the Banker's Algorithm. It involves the following:

   (a) Let **Work** and **Finish** be vectors of length m and n, respectively. Initialise:
   - `Work = Available`
   - `Finish[i] = false for i = 0, 1, ..., n-1`

   (b) Find an **i** such that both:
   - `Finish[i] = false`
   - `Need_i <= Work`

   If no such **i** exists, go to step 4.

   (c) `Work = Work + Allocation_i Finish[i] = true` Go to step 2.

   (d) If `Finish[i] == true` for all **i**, then the system is in a safe state. Otherwise it is in an unsafe state.

19 Lecture 9: Deadlock Pt.2

19.1 Resource-Request Algorithm for Process $\mathbf{P_i}$

Let `Request_i[...]` be the request vector for process $\mathbf{P_i}$.

`Request_i[j] = k` Process $\mathbf{P_i}$ wants **k** instances of resource type $\mathbf{R_j}$.

1. If `Request_i` $\leq$ `Need_i` go to step 2, otherwise raise error condition since process has exceeded its maximum claim.
2. If `Request_i` $\leq$ `Available`, go to step 3, otherwise $\mathbf{P_i}$ must wait since resources are not available.
3. Pretend to allocate requested resources to $\mathbf{P_i}$ by modifying the state as follows:

```
Available = Available - Request;
Allocation_i = Allocation_i + Request_i;
Need_i = Need_i - Request_i;
```

- `If safe` $\Rightarrow$ the resources are allocated to $\mathbf{P_i}$
- `If unsafe` $\Rightarrow$ $\mathbf{P_i}$ must wait, and the old resource-allocation state is restored.

19.2 Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
  - Single Instance of a Resource Type
  - Multiple Instances of a Resource Type
- Recovery Scheme

19.2.1 Detection Algorithm for Single Instances of a Resource Type

Maintain a **wait-for** graph. In a **wait-for** graph, nodes are only processes, there are no resources.

- $\mathbf{P_i} \rightarrow \mathbf{P_j}$ if $\mathbf{P_i}$ is waiting for $\mathbf{P_j}$

To convert between a resource-allocation graph and a wait-for graph, just join processes where one is holding a resource that the other needs. Cycles in a wait-for diagram show deadlock.

Periodically invoke an algorithm that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of $\mathbf{n^2}$ operations, where **n** is the number of vertices in a graph, this is why this algorithm is only invoked periodically.

19.2.2 Detection Algorithm for Multiple Instances of a Resource Type

Where n = number of processes and m = number of resource types:

Available Vector of length m. If `Available[j] == k`, then there are k instances of resource type $\mathbf{R_j}$ available.

Allocation n x m matrix. If `Allocation[i,j] == k`, then $\mathbf{P_i}$ is currently allocated k instances of $\mathbf{R_j}$.

Request n x m matrix that indicates the current request of each process. If `Request[i,j] == k`, then process $\mathbf{P_i}$ is requesting k additional instances of resource type $\mathbf{R_j}$.

See lecture 8 for details of algorithm.

19.2.3 Outlining the Detection Algorithm Stage

If a deadlock is detected, we must abort (rollback) some of the processes involved in the deadlock. Need to decide when and how often to invoke the deadlock detection algorthm which depends on the following:

- How often a deadlock is likely to occur?
- How many processes will need to be rolled back?
  - (One for each disjoint cycle)

If a detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes *caused* the deadlock.

19.3 Deadlock Recovery

19.3.1 Process Termination

- Abort all deadlocked processes
- Abort one process at at time until the deadlock cycle is eliminated.

In which order should we choose to abort?

- Priority of the process
- How long process has computed, and how much longer to completion
- Resources the process has used
- Resources process needs to complete
- How many processes will need to be terminated
- Is the process interactive or batch?

19.3.2 Resource Preemption

Selecting a victim Minimise cost

Rollback Return to some safe state, restart process for that state

Starvation If the same process is always picked as a victim then it could suffer from starvation. To avoid this, we can include the number of rollbacks in cost factor.

19.4 Process Management - Conclusion

What we have covered:

- Processes, threads and their life cycle
- Scheduling
- Synchronising
- Deadlock
- Hardware Support

In the next section: memory management.