Operating Systems Summary

Thijs van Loenhout

23 October, 2019

Contents

1 Introduction

This is a summary for the course Operating Systems (NWI-IBC019). This summary will mainly summarize the lecture slides. I'll try to cover the most important parts of the book that were used for the exercises (but as this book is 829 pages long, I can't promise to cover everything O_o)

2 Operating Systems

An operating system is a program that manages a computer's hardware, provides a basis for application programs and acts as an intermediary between these two sides. Operating Systems were developed to include these tasks:

- Offer a common interface from hardware to applications
- Manage the interaction with hardware
- Manage the execution of (multiple) active applications: processes
- Facilitate the communication between processes (between themselves and between the OS and the processes)
- Memory management

There are some conditions that can measure the quality of an operating system:

- Reliability (crashes)
- Performance
- Security
- Energy efficiency

2.1 The kernel

An operating system consists of a program, the **kernel**, that manages the execution of applications and connects them with the hardware. The kernel uses a scheduler to switch between processes.

The kernel runs with elevated permissions (kernel mode), which provides full access to resources and memory. Additionally, all hardware instructions are allowed. By differentiating between a restricted user mode and kernel mode, it is ensured that the system does not crash when a program does, thus increasing its reliability.

2.1.1 System calls

The only entry point into the kernel system are **system calls**: a way for programs to interact with the operating system, an interface between a process and an OS to allow user-level processes to request services of the operating system. Typically, a number is associated with each call. The system-call interface maintains a table indexed according to these numbers. The system call interface invokes the intended system call in the kernel and returns the status of the system call and any return values. It is important to note that the caller does not need to know anything about how the system call is implemented. The handling of the system call `open()` is shown in figure 1.

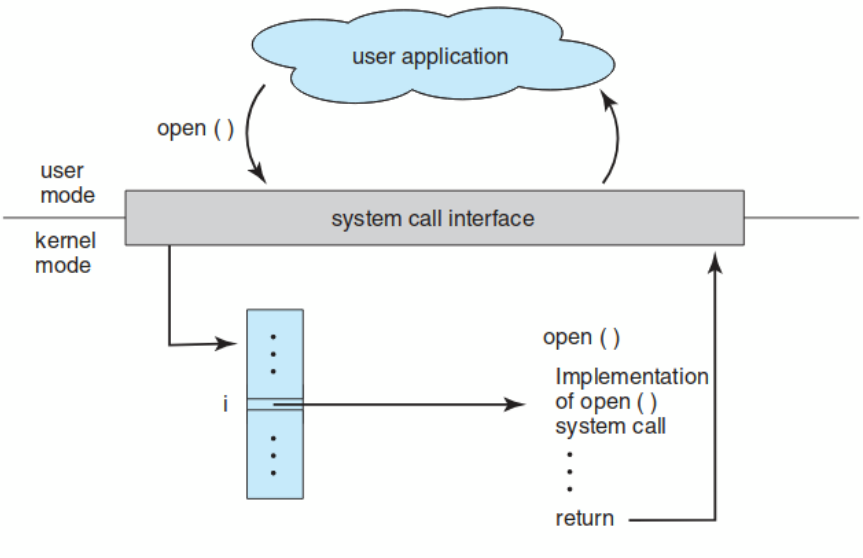


Figure 1: The handling of a user application invoking the `open() system call`

2.2 processes

While a computer program is a passive collection of instructions, a process is the actual execution of tho instructions. Each CPU (core) executes a single task at a time. A process is a task that can be assigned and executed on a specific processor. It is a unit of activities with the following properties:

- A chunk of (hardware) instructions
- Current program state
- Assigned system resources

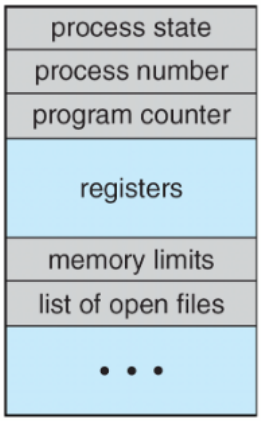
The concept of a process is essential for an OS. It is an important condition that processes are independent of each other: they should not influence each other directly. This has the consequence that each process has their own, isolated piece of memory.

The OS keeps an overview of all processes that currently exist in the system by storing a process's state in the **process control block** (**PCB**). A context includes information like the process's state (see 2.2.2), a program counter, CPU registers, memory management information, etc. (see 2.2.1)

2.2.1 Process Control Block

The process control block stores information associated with each process:

- Process state: running, waiting, etc.
- Program counter: location of the instruction to execute next
- CPU registers: contents of all process-centric registers
- CPU scheduling: priorities, scheduling queue pointers
- Memory-management information: memory allocated to the process
- Accounting information: CPU used, clock time elapsed since it started running, time limits
- I/O status information: I/O devices allocated to the process and a list of open files



2.2.2 Process states

A process has a state. During its execution, its state changes (see figure 2. This figure can expanded upon by elaborating the connection between 'ready' and 'waiting'). States include:

- New: the process is being created (for example by the system call `fork()`
- Running: instructions are being executed
- Waiting: the process is waiting for some event to occur
- Ready: the process is waiting to be assigned to a processor
- Terminated: the process has finished execution

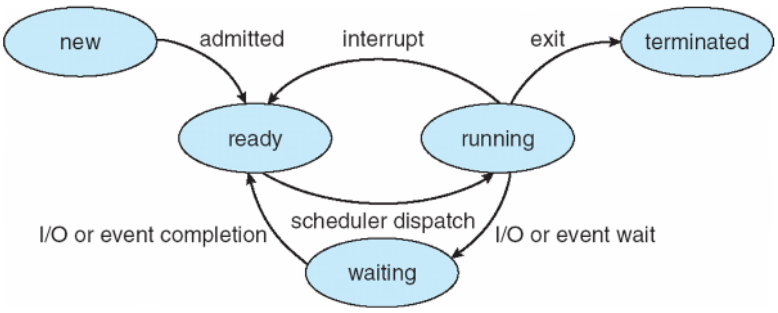


Figure 2: A diagram of process states

Switching the running process with a waiting one is called a **context switch**. This takes a lot of time because the PCB has to store the whole context of the old process and restore that of the new one. Switching will probably have bad consequences for performance and CPU caching (see section 5).

2.3 Threads

Instead of creating new processes for all computations, you can also use threads: a basic unit of CPU utilization comprised of an ID, program counter, register set and a stack. They have a few benefits:

- Responsiveness: threads may allow continued execution if part of the process is blocked (especially important for user interfaces).
- Resource sharing: threads share resources of a process, see figure 3. This is easier than shared memory or message passing between processes.
- Economy: thread creating is more lightweight than process creation (allocating memory and resources is costly). Additionally, thread switching has a lower overhead (excess or indirect computation time required for a task) than context switching.
- Scalability: processes can take advantage of multiprocessos architectures where multi-threading can result in parallel computing.

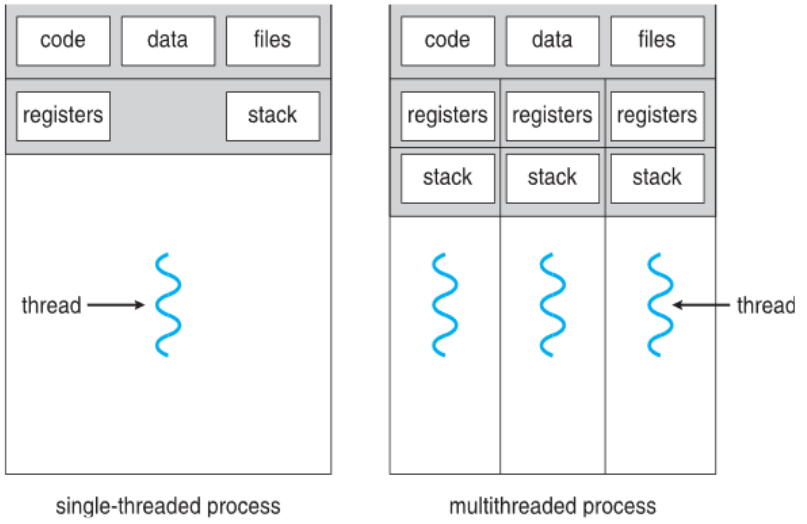


Figure 3: Single-threaded versus multithreaded processes

There is a difference to be made between user and kernel threads. The latter are supported by the kernel and managed directly by the operating system. User threads are mapped to kernel threads. This can be done in multiple ways:

- Many-to-one: Map many user-level threads to a single kernel thread. If one thread blocks, it causes all to block. Also, multiple threads may not run in parallel because only one may be in kernel at a time.
- One-to-one: Each user-level thread maps to one kernel thread. Creation of a user thread causes the creation of a kernel thread.
- Many-to-many: allows many user threads to be mapped onto many kernel threads.

2.3.1 Concurrency and parallelism

Parallelism implies a system can perform more than one task simultaneously, while **concurrency** supports more than one task making progress by constantly switching the task it works on. See figure 4. For parallelism, at least 2 cores are needed.

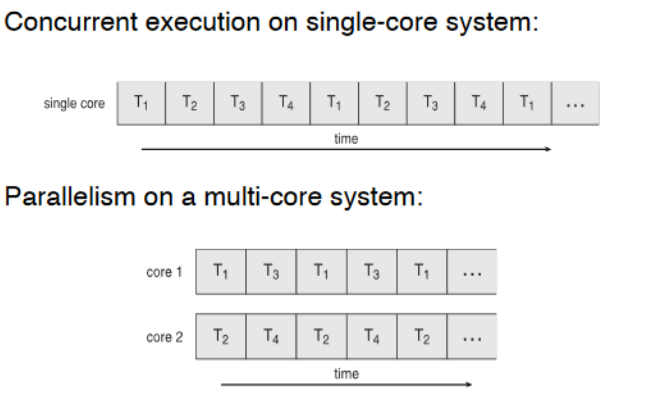


Figure 4: Tasks T_1, T_2, T_3 and T_4 performed concurrently and in parallel

You can use **Amdahl's Law** to estimate performance gains from adding additional cores to an application that has both serial (concurrent) and parallel components. For serial portion S and N processing cores:

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

As an example: say that an application that is 75% parallel and 25% serial is moving from 1 to 2 cores:

$$1 \text{ core} = \frac{1}{.25 + \frac{(.75)}{1}} = 1$$

$$2 \text{ cores} = \frac{1}{.25 + \frac{(.75)}{2}} = 1.6$$

So, the speed gain is 1.6, or 60%.

A system can split its workload using parallelism. There are two types:

- **Data parallelism**: distribute subsets of the same data to multiple cores. Each core then performs the same operation to its part of the data.
- **Task parallelism**: distributing threads across cores, each thread performing a unique operation.

2.3.2 `fork()` and `exec()`

- `exec()` is an operation that runs an executable file in the context of an already existing process, effectively replacing the previous one.

- `fork()` is an operation where a process creates a copy of itself (including local variables, the place in the code that's running, etc.)

`fork()` could cause some issues: does is duplicate the calling thread or all threads? Some UNIX systems have two implementation of `fork()`, one for each option.

3 Scheduling

If there are multiple processes to run, the way they are arranged can have great impact. In a single-processor system, only one process can run at a time. Other processes must wait. In multiprogramming, this is more complicated.
A process is executed until it must wait, typically for the completion of some I/O request (Input/Output). In this time, the CPU sits idle: time is wasted. We want to use this time productively. To achieve this, several processes are kept in memory at the same time. When a process has to wait, the OS takes the CPU away from the waiting process and gives it to another one.
Whenever the CPU becomes idle, it's the job of a ('short-term') scheduler to select a process from memory that is ready to execute and to allocate the CPU to this process. A difference must be made between **preemptive** and **nonpreemptive** scheduling: if the scheduling scheme is nonpreemptive, the process that is allocated a CPU keeps this CPU until it releases it either by terminating or by switching to a waiting state. If the scheduling is preemptive, a process can be interrupted forcefully.
The **dispatcher** is the module that gives the CPU to the process selected by the scheduler. the time it takes for the dispatcher to stop one process and start another is called the **dispatch latency**.

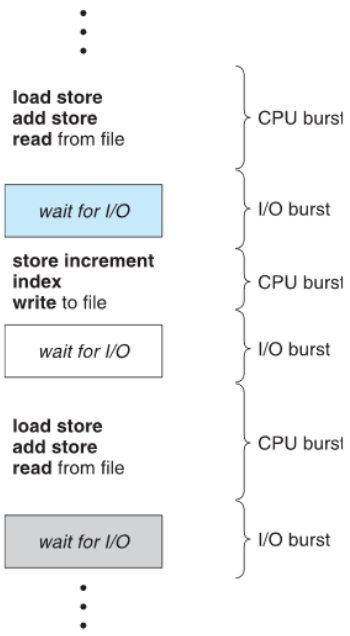


Figure 5: The CPU-I/O burst cycle

There are several criteria that can influence your choice of scheduling algorithm. These include:

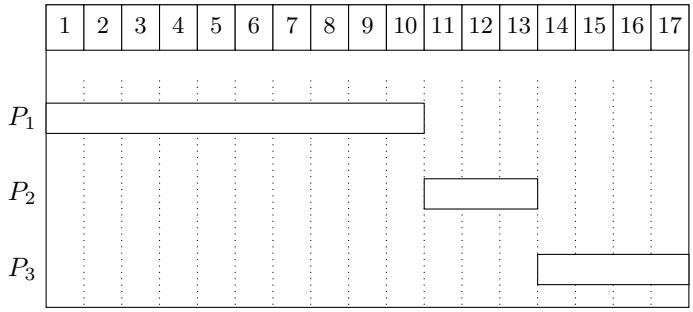
- **CPU utilization**: The amount of time the CPU is being used.
- **Throughput**: The number of processes that are completed per time unit.
- **Turnaround time**: The interval from the time of submission of a process to the time of its completion.
- **Waiting time**: The amount of time a process spends in the waiting queue.
- **Response time**: The interval from the time of submission of a process to the time of its first response. (This may be a better criterion in interactive systems than turnaround time is, as a process can probably produce some results fairly early on)

3.1 First-Come, First-Served scheduling

First-come, first-served (FCFS) scheduling is perhaps the easiest: allocate the CPU to the processes in the order they have requested the CPU. An example:

Process	Burst Time
P_1	10
P_2	3
P_3	4

The Gantt chart that is the result with an FCFS scheduler:



(Note that this is a different notation for Gantt charts than used in the lectures and the book, but the idea is the same. I think these are easier to read.)

FCFS scheduling is very much prone to the **convoy-effect**: a short process gets stuck behind a long process. In the example given above: average waiting time would be much better is P_1 would be handled after P_2 and P_3.

3.2 Shortest-Job-First scheduling

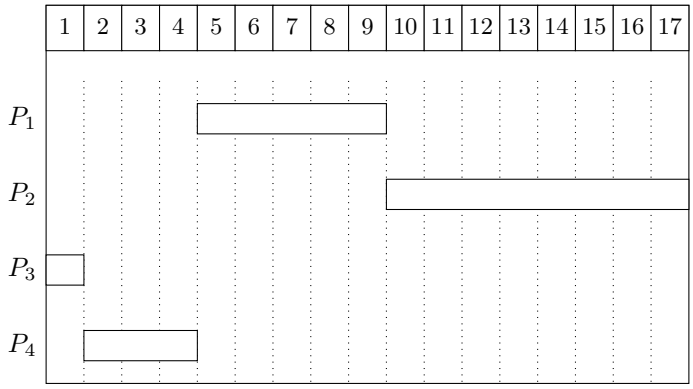
To minimize waiting time, shortest-job-first (SJF) scheduling is introduced: estimate the length of a process's next CPU burst and schedule the process with the shortest burst time. An estimate can be made with **exponential averaging**:

$$
\begin{aligned}
t_n &= \text{actual length of } n^{th} \text{ CPU burst} \\
\tau_{n+1} &= \text{predicted value of value for the next CPU burst} \\
\alpha &\in [0, 1] \\
\tau_{n+1} &= \alpha t_n + (1 - \alpha)\tau_n
\end{aligned}
$$

An example:

Process	Burst Time
P_1	5
P_2	8
P_3	1
P_4	3

The Gantt chart that is the result with an FCFS scheduler:



3.3 Priority scheduling

Priority scheduling (no fancy abbreviation for this one) is a scheduler that works with priorities: a process with the highest priority (typically the smallest number) gets the CPU allocated to it the first. A process's priority can be based on different factors. Implicitly, SJF scheduling is a variant of priority scheduling where priority is based on burst time alone!

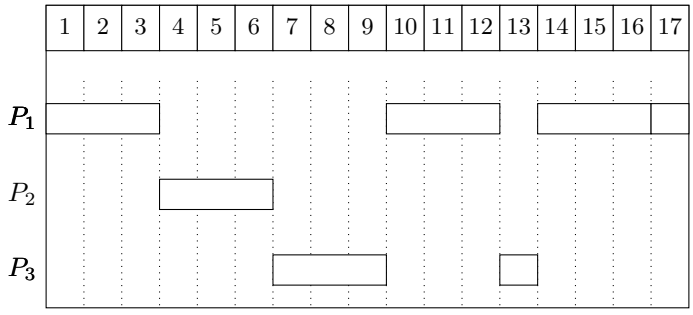
A problem may arise with priority scheduling: **starvation**. Starvation is the phenomenon that a low priority process may never execute (because there keep coming higher priority processes). A solution for this is ageing: increasing a process's priority the longer it's in the waiting queue.

For an example, look at that of SJF scheduling. The priority is equal to the burst time.

3.4 Round-Robin scheduling

In round-robin (RR) scheduling, each process gets assigned a time quantum q. After q time units have elapsed, the process is preempted and placed at the end of a FIFO (first-in, first-out) queue. The next process in the FIFO queue gets the CPU allocated to it. If q is large, RR scheduling acts the same as FCFS. q should be large with respect to the context switch time, because otherwise overhead would be too high.

An example using the same processes as in the FCFS example. $q = 3$:



4 Synchronization

Processes that are executed concurrently (see 2.3.1) may be interrupted at any time, partially completing execution. Concurrent access to shared data may result in data inconsistency, which can be a big problem.

An example is the **producer-consumer problem**, where a producer produces a resource that is consumed by a consumer. Say there is a variable `counter` that both the producer as the consumer use and they respectively increment and decrement it. A problem occurs in the situation in figure 6. This is called a **race condition**.

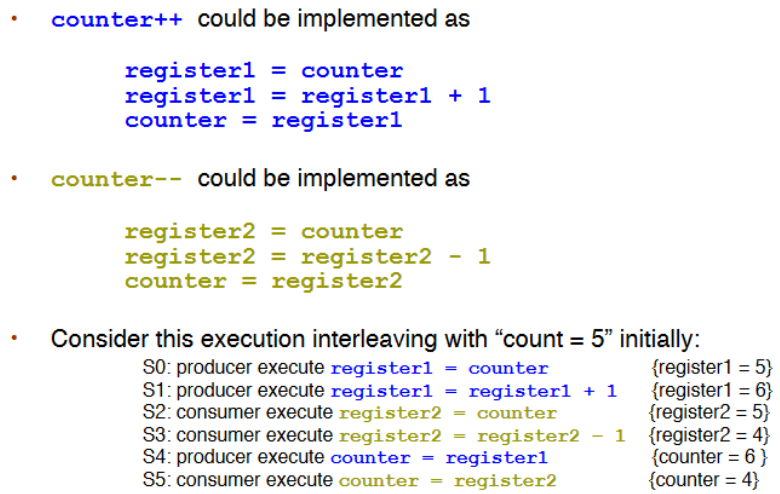


Figure 6: `counter` is incremented once and decremented once, but this does not have the expected result

This introduces a more generalized problem: **the critical section problem**. Each process has a critical section segment of code that performs operations on shared variables, that wants to edit the same file as other processes, etcetera. Then only one process should be able to execute its critical section at the same time.

The problem here is to design a protocol to prevent issues as in figure 6. A solution has several requirements:

- **Mutual exclusion**: If process P_i is executing its critical section, no other process can be executing in their critical section
- **Progress**: If there is no process executing in its critical section and there is some process that wants to enter its critical section, then the selection of this process cannot be postponed indefinitely.
- **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.

Many systems also provide hardware support for implementing critical section code.

4.1 Peterson's algorithm

There is a famous solution for the critical section problem for two processes sharing two variables. It is called Peterson's solution. The two variables:

- int `turn` indicates whose turn it is to enter the critical section

- Boolean `flag[2]` is used to indicate is a process is ready to enter the critical section. `flag[i]-true` implies that P_i is ready.

Peterson's solution for process i:

```
do {
    flag[i] = true;                 //i wants to enter its critical section
    turn = j;                       //give the turn to j
    while (flag[j] && turn == j);   //wait until j is finished
    //critical section
    flag[i] = false;
    //remainder section
} while (true)
```

The mutual exclusion, bounded waiting and progress requirements are all reserved. (It is important to make note of this in all exercises involving the critical section problem)

4.2 Mutex Locks and semaphores

Peterson's solution works for two processes. A simple, more general software solution to the critical section problem are mutex locks: a lock that protects critical sections. This is commonly a boolean variable indicating if the lock is available. A process must call `acquire()` before and `release()` after its critical section. These operations must be atomic (meaning there can't be a context switch halfway through them).

Mutex locks require **busy waiting**: while a process is in its critical section, other processes that want to enter their critical section must loop continuously. This is called a **spinlock** and wastes CPU cycles (however, no context switch is required). One thread can spin on one processor, while another thread performs critical section.

A representation in code:

```
acquire() {
    while (!available) {
        //busy waiting
    }
    available = false;
}

release() {
    available = true;
}
```

```
do {
    acquire lock
        //critical section
    release lock
        //remainder section
} while (true)
```

Another, similar solution for the critical section problem is the use of **semaphores**. Semaphores are integer values that represent the number of resources available. Two standard modifications modify the semaphore: `wait()` and `signal`. Again, these operations should be atomic.

```
wait (S) {
    while (S <= 0) {
        //busy waiting
    }
    S--;
}
```

```
signal (S) {
    S++;
}
```

Semaphores are a generalization of mutex locks: a counting semaphore can have an integer value over an unrestricted domain, a binary semaphore can only take values of 0 and 1. A binary semaphore is just a mutex lock.

Busy waiting is still an issue. One could bypass this by associating a waiting queue to each semaphore.

4.3 Deadlocks

A **deadlock** is a scenario wherein two or more processes are waiting indefinitely for an event that can only be caused by one of the waiting processes. For example:

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
...	...
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Starvation is the act of indefinite blocking: a process may never be removed from the semaphore queue in which it is suspended. Examples of deadlock situations are the bounded buffer problem, the readers and writers problem and the dining philosophers problem. (I recommend you to look at them to get a grip on the idea of deadlocks. Resources are wildly available. My edition of the book presents them in section 6.7, page 269).

So, while semaphores can solve certain synchronization issues (the critical section problem), they come with their own (deadlocks). A deadlock can occur only if several conditions are met:

- **Mutual exclusion**: only one process at a time can use a resource.
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No Preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait**: there exists a set $\{P_0, \ldots, P_{n-1}\}$ of waiting processes such that P_i is waiting for a resource held by process $P_{i+1 \mod n}$.

It's fairly easy to see that removing any of these conditions provides an easy 'solution' for the deadlock. This way, we can *prevent* deadlocks. For deadlock *avoidance*, a priori information is needed. For example, each process could declare the maximum number of resources of each type it may need. With an algorithm, you could determine whether the resource-allocation state is safe or not. A resource-allocation state is defined by the number of available and allocated resources and the maximum demands of the processes. A system is in a safe state is there is a sequence of all processes such that for each process P_i the resources it can still request can be satisfied by the currently available resources plus the resources held by processes P_j with $j < i$. With deadlock avoidance, you make sure to never leave this safe state.

Deadlocks can be visualized using resource-allocation graphs: graphs that depict, you guessed it, resource allocation. Some notation:

- A circle represents a process
- A square represents a resource. The units inside a square represents the number of resources of this type are available
- An arrow to a resource is a request of that resource
- An arrow from a resource to a process represents this process holding that resource (the arrow is drawn from the particular unit inside the resource-square)

A deadlock can only occur if there is a cycle in the graph (but a deadlock is not guaranteed if there is one!). Some examples are shown in figure 7. Note that the third situation is not a deadlock state, as instances of R_1 and R_2 can be freed by P_2 and P_4 respectively.

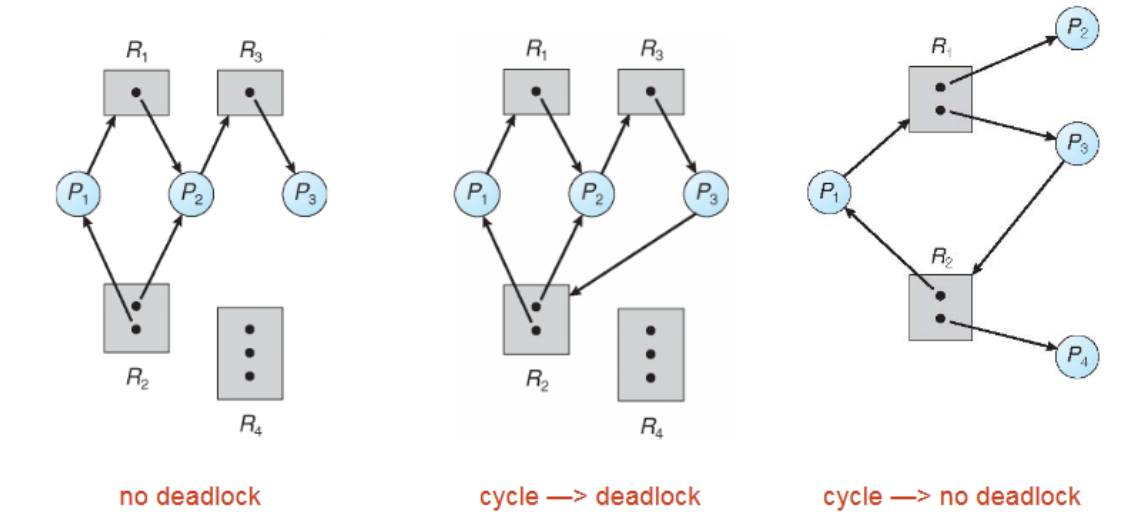


Figure 7: Some resource allocation graphs

Deadlock avoidance can be achieved via resource-allocation graphs. For this, make use of dotted lines to indicate the request of a resource. You can see when an unsafe state arises:

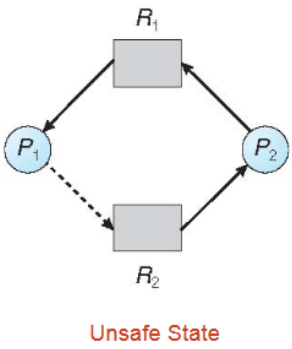


Figure 8: An unsafe state for a resource-allocation graph

This method of deadlock avoidance is only reliable if there's one instance of each resource type.

4.3.1 Banker's algorithm

If there are more instances of each resource type, one can use Banker's algorithm to determine the state of a system. This algorithm tries to find a safe sequence of all processes. It works as follows:

1. For some processes, current and maximum allocation are given, as well as the resources initially available (I'll call this *work*).
2. Calculate the *Need* for each process by subtracting the current allocation from its max: $Need_i = Max_i - Alloc_i$.
3. Try to find a process for which holds: $Need_i \leq work$.
   (a) If there is none, stop the algorithm
   (b) If there is at least one:
      i. Pick process P_i for which the inequality holds
      ii. Calculate the new *work* by $work = work + Alloc_i$
      iii. Add P_i to the safe sequence
      iv. Jump back to step 3 and ignore P_i for the remainder of the algorithm

Ultimately, you've found a sequence of processes. If this sequence contains *all* processes, than it is safe.

I will give an example:

1. Say there are 5 processes. Their allocation, maximum resource usage and the current work are given:

Process	Allocation	Max	work
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

2. Calculate each process's need by $Need_i = Max_i - Alloc_i$:

Process	Allocation	Max	work	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 2

3. Find a process P_i with $Need_i \leq work$. For P_1: 1 2 2 $\leq$ 3 2 2 $\rightarrow$ $work = work + Alloc_i$ = 3 2 2 + 2 0 0 = 5 3 2. Add P_1 to the safe sequence and update the table:

Process	Allocation	Max	work	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2	5 3 2	1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 2

   Safe sequence: $\langle P_1 \rangle$

4. Try to find a process with $Need_i \leq$ 5 3 2. P_1 should be ignored. Choose, for example, P_3. $work$ = 5 3 2 + 2 1 1 = 7 4 3. Safe sequence: $\langle P_1, P_3 \rangle$ (I will omit the new table from now on. Only the *work* column is updated)

5. Choose for example P_4: $work$ = 7 4 3 + 0 0 2 = 7 4 5 Safe sequence: $\langle P_1, P_3, P_4 \rangle$

6. Choose for example P_0: $work$ = 7 4 5 + 0 1 0 = 7 5 5 Safe sequence: $\langle P_1, P_3, P_4, P_5 \rangle$

7. Choose for example P_2: $work$ = 7 4 5 + 3 0 2 = 10 5 7 Safe sequence: $\langle P_1, P_3, P_4, P_5, P_2 \rangle$

So in this example, we've found a safe sequence $\langle P_1, P_3, P_4, P_5, P_2 \rangle$! (We would have a negative result if we could not fit all processes in a sequence: if at a certain step there were still processes left, but none of them have a $Need$ lower than the $work$ available)

There are plenty of examples online. Here is a link to a clear video explanation. I have used the same numbers as used in the video so you can watch this if something is still vague: `https://www.youtube.com/watch?v=2V2FfP_olaA`.

5 Memory

For a program to run, it must be brought from the disk into memory and placed within a process. The only memory the CPU can access directly, are the **registers** and the **main memory** (RAM). Registers can be accessed in one CPU clock, main memory access can take many cycles, causing a stall. **Cache** sits between the main memory and the CPU registers and is very fast.

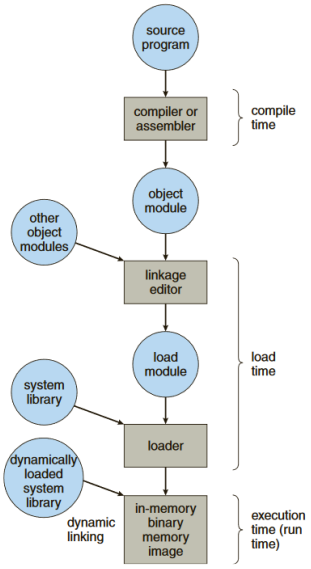


Figure 9: Multistep processing of a user program

Programs that must be placed inside a process form an input queue. The physical address space may start at address 0000, but could be inconvenient to have the first user process always at 0000. That's where **address binding** comes into play: relocating relative addresses (for example: '10 bytes from the beginning of this module'). This is done by the compiler. Aside from compiling, the program goes through several steps before executing. There are three main stages where address binding may happen:

- Compile time: If the memory location is know beforehand, absolute code can be generated. Code must be recompiled if the starting location changes.
- Load time: if the memory location is not known at compile time, relocatable code must be generated.
- Execution time; if the process can be moved from one memory segment to another during its execution, binding is delayed until run time (hardware support is needed for this).

To ensure correct operation, protection of memory is required. Each process gets its own memory space. This is defined by the **logical address space**.

A pair of base and limit registers define this space. Every memory access generated in user mode must be checked by the CPU to be sure its in the bounds of that user. There is hardware support for this.
The concept of logical address space that is bound to separate physical address space is central to proper memory management:

- Logical addresses: addresses generated by the CPU. Also referred to as virtual addresses. The logical address space is the set of all logical addresses generated by a program.
- Physical addresses: addresses seen by the memory unit. The physical address space is the set of all physical addresses generated by a program.

Logical and physical addresses are the same in compile time and load time address-binding schemes, but differ in execution time binding-schemes.

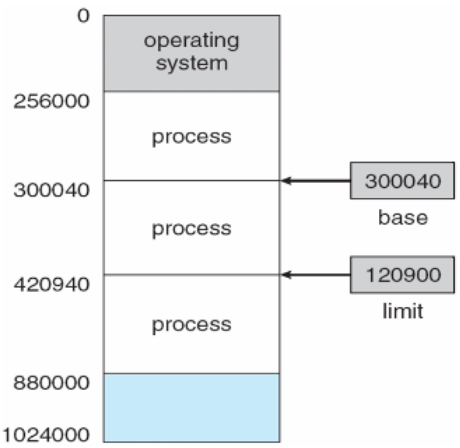


Figure 10: Logical address space

The **Memory-Management Unit** (MMU) is a hardware device that maps virtual addresses to physical addresses at run time. The base register is now called the **relocation** register and its value is simply added to every address generated by the user process at the time is is sent

to memory. This user process only deals with logical addresses: it never sees the 'real' physical addresses.

5.1 Dynamic loading

So far, it has been necessary for the entire program and all data of a process to be in physical memory for the program to execute. This way, the size of a program is limited to the size of physical memory. To obtain a better memory-space utilization, we can use **dynamic loading**. With this, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases such as error routines.

5.1.1 Swapping

A process can be **swapped** temporarily out of memory to a **backing store**, to later be brought back into memory for continued execution. This way, total memory space of processes can exceed the physical memory. The backing store is a fast disk large enough to accommodate copies of all memory images for all users. It must provide direct access to these memory images. A swapped out process does not always need to be swapped back into the same place as it was before (this depends on the address binding method).

A major part of swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped. Context switch time (see 2.2.1) can be very high.

The system maintains a ready queue of ready-to-run processes which have memory images on disk.

5.2 Contiguous memory allocation

The main memory must accommodate both the operating system and the various user processes. The OS is typically placed in low memory and processes in high memory. We want to allocate main memory in the most efficient way possible. An early method is contiguous memory allocation: each process is contained in a single section of memory that is contiguous to the section containing the next process.

With this way of allocation, the degree of multiprogramming is limited by the number of partitions (that is to say: the number of processes we can run at the same time). Variable-partition sizes are used for efficiency: the partition is sized roughly to a given process's needs.

A **hole** is a block of available memory. When a process arrives, it's allocated memory from a hole large enough to accommodate is. If a process exits, it frees its partition. The newly created hole is merged with the adjacent ones. Thus, holes can be of various sized and be scattered all throughout memory. The operating system maintains information about allocated partitions and holes.

5.2.1 Algorithms

There are several ways to determine in which hole a process ought to be placed. Here are three:

- First-fit: Allocate the first hole that is big enough. This is good in terms of speed, but is not necessarily the most efficient in terms of storage utilization.
- Best-fit: Allocate the smallest hole that is big enough. Problems are that you must search the entire list of holes. Additionally, best-fit produces the smallest leftover holes (remainder space of a hole that was bigger than the process placed in it).

- Worst-fit: Allocate the largest hole. Again, you must search the entire list of holes, but this time, the largest leftover hole is produced. A larger hole has a higher chance of being large enough for the next process.

5.2.2 Fragmentation

As mentioned before, processes will probably not fit exactly in the holes allocated to them. This causes the problem of fragmentation. There are two different (but very much related) types:

- Internal fragmentation: the memory allocated to a process may be slightly larger than the requested memory. The size difference is memory internal to the partition, but is not being used. Say we allocate a hole of 1000 bytes to a process that requests 995. The 5 remainder bytes are part of the allocated partition, but are not in use.
- External fragmentation: the total memory space exists to satisfy a request, but it's not contiguous. A solution for this is compaction: shuffling memory contents so as to place all free memory together in one large block. (This is only possible if relocation is dynamic and done at execution time)

5.3 Non-contiguous memory allocation

5.3.1 Segmentation

It could be beneficial to view memory not as a linear array of bytes, but as a collection of variable-sized segments. **Segmentation** is a memory-management scheme that supports this view. A logical address space is a collection of segments, Each segment has a name and a length. The logical addresses consist of tuples that specify both the segment name and the offset within the segment.

Because in reality, memory *is* a linear array of bytes, there must be a map between (two dimensional) logical addresses and (one dimensional) physical addresses. This is achieved by the **segment table**. Each entry in this table has a segment base and limit, not unlike the register base and limit seen in 10. Figure 11 shows an example of segmentation. Figure 12 shows how dedicated hardware works.

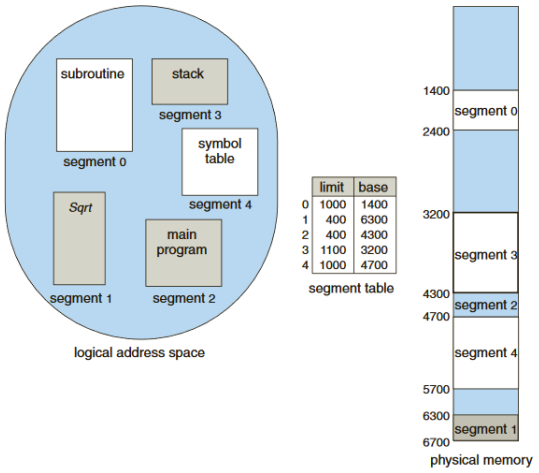


Figure 11: An example of segmentation

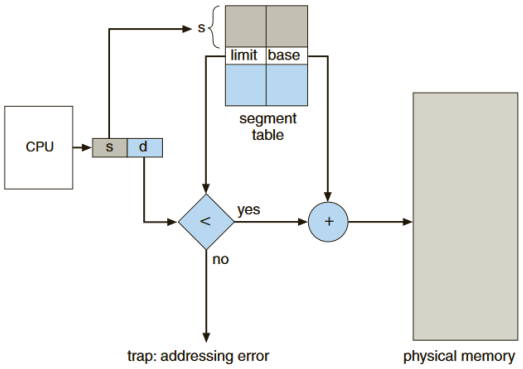


Figure 12: Segmentation hardware

5.3.2 Paging

Paging is another memory scheme that offers the advantage of memory space being non-contiguous. Paging also avoids external fragmentation and the need for compaction, whereas segmentation does not.

The basic concept is to divide the physical memory into fixed-size blocks called **frames** and the logical memory into blocks of the same size called **pages**. The system needs to keep track of all free frames. Just like with segmentation, paging makes use of a table, the **page table**, to translate logical to physical addresses. The backing store is also split into pages. An example is shown in figure 13.

The page size is typically a power of 2, to make translation of a logical address into a page number easy. Define the following numbers:

$$\begin{aligned} p &= \text{page number} \\ d &= \text{page offset} \\ 2^m &= \text{size of logical address space} \\ 2^n &= \text{size page} \end{aligned}$$

In the example of figure 13, $n = 2$, $m = 4$. The logical address 13 has offset $d = 1$. Page 2 is mapped to frame 2 according to the page table. So, the physical address that logical address 13 maps onto is $n * p + d = 2 * 4 + 1 = 9$.

Let's do an example to calculate internal fragmentation. Say page size $p = 2,048$ bytes. Our process has size $72,766$ bytes. $72,766 = 35 * 2,048 + 1,086$. Thus, the internal fragmentation is $2,048 - 1,086 = 962$ bytes.

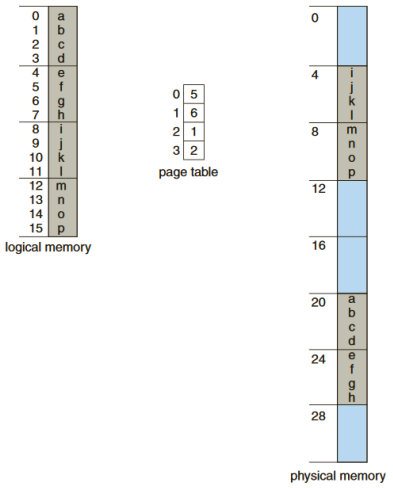


Figure 13: An example of paging

The page table is kept in main memory (rather than in registers). The **Page-table base register** (PTBR) points to the page table. The **Page-table length register** (PTLR) indicates the size of the page table. These two are fast registers. In this scheme, every data / instruction access requires two memory accesses: one for the page table and one for the data / instruction. This problem van be solved by the use of a special fast-lookup cache called **associative memory** or **translation look-aside buffers** (TLBs). Some TLBs store **address-space identifiers** (ASIDs) in each TLB entry to uniquely identiy each process to provide address-space protection.

On a TLB miss, a value is loaded into the TLB for faster access next time. (According to the lecturer, all these abbreviations are silly and you do not need to know them. I put them here for completion. This last fact is important though)

As you can probably imagine, page tables can grow to enormous lengths with the address space of modern computers. We don't want to allocate that contiguously in main memory. How can we solve this? By paging the page table! This lead to hierarchical page tables. An alternative is hashed page tables. These contain a chain of elements hashing to the same location. Each element contains the virtual page number, the value of the mapped page frame and a pointer to the next element.

5.4 Memory protection

Memory protection can be implemented by associating a protection bit with each frame to indicate if read-only or read-write access is allowed. On the logical side, you can attach an **valid-invalid bit** to each entry in the page table. "Valid" then indicated that the associated page is in the process's logical address space and is thus a legal page. "Invalid" indicates that the page is not in the process's logical address space. If there is a reference to a page, check whether the reference is valid or invalid. If it's invalid (a **page fault**): abort and swap the page into memory. Set the validation bit to valid.

It must be noted that it could be very useful for some processes to share the same pages (say if they both use a certain library). Good protection must be in place of course.

5.5 Virtual memory

Code needs to be in memory to be executed, but rarely the entire program is used at the same time. This is a waste. We introduce **virtual memory** to make a separation between logical and physical memory. Virtual memory is not mapped directly onto physical memory, only when it needs to be. This allows the logical address space to be much larger than the physical address space and via this. It has benefits like more efficient process creation and being able to run more programs concurrently. (This idea is actually already discussed in the course Hacking in C (NWI-IPC025), for which I've also put a summary online. You can check that one for more information)

5.6 Page replacement

The slides discuss a bit on the process of replacing pages. What it boils down to is finding a so called "victim" in physical memory that swaps with the page you want to swap in. To determine the victim, you'll use page and frame replacement algorithms. A frame-allocation algorithm determines how many frames to give to each process and which frames to replace. A page-replacement algorithm wants the lowest page-fault rate on both first access and re-access. Evaluation of such an algorithm can by done by running it on a particular string of memory references (**reference string**) and computing the number of page-faults.

I'll discuss a page replacement algorithm: first in, first out (FIFO). Our reference string is 7,0,1,2,0,3,0,4,2,3,0,3,0,2,1,2,0,1,7,0,1. A FIFO algorithm is very simple: the first element to come in, is the first that goes out when we need to throw one out. Say we have access to 3 frames (so 3 pages can be in memory at a time per process). The result is shown in figure 14. Every blue rectangle indicates that a swap must have taken place (a page-fault has occurred). So in total, this example has 15 page-faults.

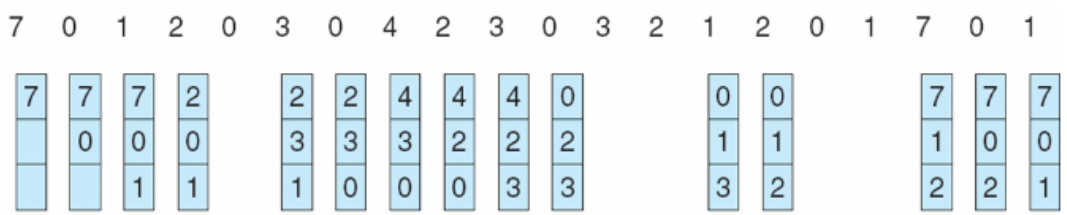


Figure 14: Our reference sting placed into page frames

Belady's anomaly states that adding more frames can cause more page-faults.

Another algorithm that can be used is the **Least Recently Used** (LRU) algorithm. Here, you swap the process that, surprise surprise, was used least recently. In the example above, this algorithm results in 12 page-faults.

6 File Systems and IO systems

There are different file types, like text files, executables, etcetera. They can have a lot of attributes attached to them, like their name, a unique identifier and their type, location, size and permissions. There's also a lot of operations to perform on most of them, such as `create`, `write`, `read`, the list goes on...

6.1 Open File Locking

Different locks can be placed on files:

- Shared lock: several processes can acquire it concurrently
- Exclusive lock: only one process can acquire it at one time

File locking can be either mandatory of advisory, meaning that access is denied depending on locks being held and requested in the first situation, or that processes can find the status of locks and decide what to do in the latter.

6.2 File access methods

There are two ways of file access:

- Sequential access: one record after the other. `read_next()` reads the next portion of the file. `write_next()` appends to the end of the file. The file-pointer (a pointer to the last read / write location) can be reset or can skip records.
- Direct access: the file has a fixed length of logical records.

A file system must be **mounted** before it can be accessed. An unmounted file system is mounted at a mount point.

6.3 Directories

Just as with files, there are plenty of operations that can be performed on a directory, like searching for a file, creating a file, deleting a file, etcetera. There are multiple ways directories can be organized. Different factors should be taken into account:

- Efficiency: locating a file quickly
- Naming: convenient to users (also: two users can have the same name for different files or a file can have different names)
- Grouping: logical grouping of files by properties

Several organizational methods are discussed, like a single-level directory (a directory contains only files, not other directories) and two-level directories (separate single-level directories for each user). There are a lot of options, none of which that is particularly scary.

6.4 Input and Output (I/O)

Little time was talked about this topic in the lectures, so I do not expect it to be the most important for the exam.

There is a lot of hardware support for I/O (microphones, keyboards, mouses, etc.) I/O devices produce and receive signals, called **concepts**. A connection point for a device is called a **port**. the **bus** is a set of wires and a communication protocol. The **controller** is a set of electronics that operate the combination of the port, the bus and the device.

How I/O devices interact with their host and the controller, is like that of a producer-consumer relationship.

Polling is the act of actively sampling the status of an external device. For each byte of input or output, the host does some operations. It is important to remember that there is a **busy-wait cycle** to wait for I/O from the device.

The slides then present a lot of hardware and miscellaneous information. I decide to skip this, as this is barely talked about in the lecture.

6.4.1 Blocking and nonblocking I/O

- **Blocking I/O**: a process is suspended until the I/O is completed. This is sometimes desirable, but sometimes inconvenient.
- **Nonblocking I/O**: the I/O call returns as much as available. This is particularly useful for user interfaces, for example.
- **Asynchronous I/O**: the process runs while the I/O executes. This is difficult to implement.

6.4.2 Kernel I/O subsystem

The kernel may need a special I/O system. This includes the scheduling of a device queue, storing data in memory while transferring between devices (buffering), caching (faster device holding copy of data) and spooling (holding the output for a device). It must also handle errors.

7 Protection and security

A computer consists of a collection of hardware or software objects. Each object has a unique name and should be accesses by processes through a well-defined set op operations. The protection problem is to ensure that each object is accessed correctly and only by those processes that are allowed to do so.

So, we should define some access rights to processes. This ought to be done by the principle of least privileges: programs, users and systems should be given just enough privileges to perform their tasks, but not more than that! This way, damage is limited it the process were to run incorrectly. Privileges can either be static (meaning they don't change) or dynamic (meaning they can change as needed) and is basically another word 'rights'.

The resources a process may access are specified in a **protection domain**. The **access rights** are defined like a tuple of the name of the object and a set of valid operations that can be performed *on* the object: $\langle object - name, rights - set \rangle$. The **domain** is the set of access rights. **Domain switching** means dynamically enabling a process to switch from one domain to another, meaning changing the access rights of a process.

7.1 Access matrix

Typically, access rights are stored in an **access matrix** wherein rows represent domains and columns represent objects. `access(i,j)` is the set of operations that a process executing D_i can invoke on $object_j$. An example is shown in figure 15. When a user creates an object, they can define the access column for that object.

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figure 15: An example of a small access matrix

The access matrix can be expanded to dynamic protection. This means entries of the matrix can be changed (e.g. by adding or deleting (special) access rights). The '*' allows a right to be copied within the same column. In 16, the `read` right for file F_2 is copied from the D_2 to the D_3 domain.

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Figure 16: An example of copying rights within a column of the access matrix

One special right is the 'owner' right. The owner rights allow to add and remove rights within a column (by now you've probably noticed that all these changes affect columns. This makes sense if you think about it, as the columns represent the files).

An implementation of an access matrix can prove problematic. With a lot of files and a lot of possible domains, it could get huge (though sparse).

7.2 Security

This part of the course is really similar to what was discussed in the course 'Security' (NWI-IPC021), for which I've also put a summary online. I recommend you to look at that if you want a more detailed explanation.

Security is the act of ensuring our protection works correctly. A system is **secure** if resources are used and accessed as intended under all circumstances. **Intruders** attempt to breach security, a **threat** is a potential security violation and an **attack** is an attempt to breach security. Several categories of security violation correspond to one goal in security:

- Breach of confidentiality: unauthorized reading of data
- Breach of integrity: unauthorized modification of data
- Breach of availability: unauthorized destruction of data
- Theft of service: unauthorized use of resources
- Denial of service (DOS): prevention of legitimate use

Here are several common methods of security violation:

- Masquerading: pretending to an an authorized used to escalate privileges (breach of authentication)
- Replay attack: using a previously received message to make a breach
- Man-in-the-middle attack: the intruder masquerading itself as the sender to the receiver and vice versa. They sit in the data flow
- Session hijacking: intercepting an already established session to bypass authentication

Measures of security levels are physical, human, OS and network. Program threads include things like a Trojan horse (code segment that misuses its environment), a trap door (specific identifier that circumvents normal security procedures), a logic bomb (a program that initiates a security incident under certain circumstances) and buffer / stack overflows (as discussed in the course Hacking in C).

A virus is a code fragment embedded in a legitimate program. It is self-replicating and designed to infect other computers.

Several other program threads of viruses were briefly mentioned. They all have descriptive names: boot-sector virus, source code virus, stealth virus (hard to detect until the damage is already done), etc.

7.2.1 Encryption

Cryptography is the broadest security tool available. An encryption algorithm consists of a set K of keys, a set M of messages and a set C of ciphertexts (encrypted texts). A function $E : K \to (M \to C)$ is a encryption function for generating ciphertexts from messages. A function $D : K \to (C \to M)$ is a decryption function for generating messages from ciphertexts. Two forms of encryption:

- Symmetric encryption: the same secret key is used to encrypt and decrypt.
- Asymmetric encryption: a public key is used to encrypt data. A secret private key, belonging to an individual user, is used to decrypt data (an example is RSA). These functions often make use of the fact that prime factorization is hard.