BSc. Computer Science Year 1

A comprehensive summary on

Operating Systems

M. Diallo, R.A.J. Wacanno, D. Kroeb,
F. Van Verseveld, J. Stalenburg and B. Groskamp

May 2018

Contents

1 Processes

M.Diallo

A process is an **abstraction** for the execution of a task. Properties of a process abstraction include a state and a **process identifier (PID)**. A process is also the set of concrete resources used to execute the task including:

- Processor registers: e.g. program counter
- Data: Variables and constants
- Code: Instructions for the processor
- OS structures: Memory layout, buffers, file descriptors

The life cycle of a process contains five stages. New processes are put into the **new state**. Eventually these processes will advance into the **ready state**. Once they have been told to execute, they'll enter the **running state**. From that point on, there can be multiple things that occur. The process can enter a **blocked state**, after which it returns to the **ready state** and the cycle continues. Or it can enter the **finished** state.
In each off these state/queues, various things are happening.

New: The process is loaded and required resources are reserved.
Ready: Process is ready to use CPU time. The scheduler decides the process *dispatch order*.
Blocked: Process is waiting for an event (I/O, message, timer, . . .).
Running: A CPU core is executing the process.
Defunct: The finished process is cleaned and resources are released.

A **context switch** happens when there is a transition of one process to the next one on the same processor/core.
In the process control block are stored by the OS:

- Process identifier (pid), just an integer.
- Process relation: Owner/permissions/parent process/child process.
- Content of CPU registers.
- Process state: new, ready, blocked, running or finished.
- **MMU(Memory management unit)** information.
- Allocated resources (files, channels, . . .).
- Used CPU time and statistics.
- Pointers to other PCBs (for obtaining and traversing a process list).

1.1 Process creation

Unix uses **fork** and Windows uses **spawn**. A fork duplicates an existing process where the duplication becomes the child and the original is the parent.

1.1.1 Userprogram startup arguments

During the first assignment, one can remember that the return code of the fork call was **zero for the child** and the **PID of the child process for the**

parent. It is common to have a fork call be followed by a call to a variant of **exec** (execv, execvp, . . .) This command accepts a three-tuple containing:

- Path to binary file of the program (e.g.: `/bin/bash`)
- Arguments for the new program's main (e.g.: `-Wall -std=gnu99`)
- Environment variables (e.g.: shell prompt line, language and locale settings)

1.1.2 Fork performance

A common question is whether fork + exec are expensive operations. After all, all the data needs to be copied. On systems with virtual memory, **copy on write** greatly reduces this cost because pages are copied *on demand* when written.

1.2 Communication between processes

There's various mechanisms of communication being made possible between various processes:

- Arguments + environment during exec (Main)
- file descriptors (Main)
- pipes (Main)
- signals (Main)
- Additionally also available are:
- Shared memory
- Message queues
- Semaphores

1.2.1 File descriptors

These are a uniform interface for all sorts of I/O to files, devices, network and pipes. The file descriptors are essentially integers representing an index of a table. This table is managed by the Operating System with which it can determine where the file descriptor points to. There are various API for file descriptors:

- Open/Connect: open, socket, accept, pipe ..
- Read/Write: read, write, recv, send ..
- Configure/Position: Lseek, fcntl, ioctl, ...
- close/terminate: close

See the appendices for more information about libc calls.

1.2.2 Pipes

Pipes make it possible to connect the input/output of two processes to each other. Usually this is between a parent process and a child process. The process for creating a pipe that connects a parent and child is as follows:

1. Initialize a variable (integer array?) int fd_id[2];
2. The parent process calls pipe(fd_id)
3. After this is done, the parent gets 2 file descriptor ids back. **The communication is uni-directional** meaning that the communication goes in one way. Reading is done from fd_id[0] Writing is done to f_id[1].
4. Parent then calls fork
5. Parent will be writing to fd_id[1]
6. Child will be reading from fd_id[0]
7. It is important to remember that 0 is for reading, 1 is for writing. [0_____1] information flows from right to left.

Another example when executing the command `bc | grep a`:

1. The shell/parent creates a pipe with FD0 and FD1
2. The shell forks, this fork will be child 1.
3. Child1 connects it's output (STDOUT) with FD1 and it **closes FD0 as it will not use it**
4. Child1 Executes BC
5. Shell **closes FD1 since Child1 is done executing/writing** .
6. Shell forks to child2
7. Child2 connects its STDIN with FD0 (inherited from parent).
8. Child2 executes grep
9. Shell closes FD0 as child2 has finished executing/reading.

It is of the essence that unused file descriptors are closed. It is also essential that only the parent is creating pipes and children. Nested forks and pipes are bad.

1.2.3 Signals

Signals are a form of inter-process communication. It is sent to a process or thread to notify it of an occuring event. Upon a signal being sent, the operating system interrupts the target's flow of execution to deliver the signal. It is important to note that this execution can be interrupted during any non-atomic interruption. Once the signal is delivered, the receiving process will call its signal handler for the specific type of signal. If none is defined, the default will be used. The following are a list of signals and their default values:

- SIGINT (default: stop process) Request to terminate the program (Ctrl+C)
- SIGSTOP (default: suspend process) Request to suspend(Job control) the program (Ctrl+Z)
- SIGHUP (default: stop process) Program is trying to use a closed file descriptor

- SIGFPE (default: stop process) Program performed an invalid arithmetic operation (Division by zero)
- SIGBUS, SIGSEGV (default: stop process) Program performed an invalid memory operation (segfault)
- SIGCHLD (default: ignored) Something happened to a child process (child done executing)

2 Scheduling

M.Diallo

As earlier discussed in **processes**. There are various states a process can reside in. Eventually these processes will need to be **scheduled** once they're **ready**. **Scheduling is the distribution of a limited resource.** In operating systems, this this usually pertains to distributing the CPU time to process, IO scheduling and memory scheduling. When talking about scheduling it's important the definition of the following terms is understood:

2.0.1 Context Switch

A context switch is what happens when one switches from one task to another.

2.0.2 Turnaround Time

Turnaround time is the total amount of time it takes to fulfill a request.

2.0.3 Release time

Absolute time it takes for a process to become ready to execute

> Release time is the earliest time when a job/task can start execution.

2.0.4 Finishing Time

Absolute time it takes for a task to finish its execution

2.0.5 Response Time

1. Time between a start and finish of a task (real time domain).
2. Time until the system responds (interactive tasks).

2.0.6 Goal of scheduling

When scheduling tasks, the following objectives should be kept in mind

1. Fairness - each task should be given an equal shore of the CPU
2. Non-Starvation - No task should not finish
3. Prioritization - Some tasks may be more important
4. Short response time for interactive tasks
5. Turnaround - Minimal time until output
6. Throughput - Maximize number of jobs processed

It is quite evident that these requirements contradict each other. See for example objective three and one. One determines that each task should be given an equal share of the CPU, while the other says that certain tasks may be more important. **From that one can conclude that there is no single scheduling policy that is optimal in all scenarios.**

2.0.7 Preemptive vs Non-preemptive

When a scheduler is said to be preemptive, it means that the scheduler allows for running processes to be interrupted and for another process to start running even if the older process has not finished. A scheduler is said to be non-preemptive if it does not allow that to happen. While this is a minor difference, it has a significant effect. Non-preemptive scheduling is an unfair scheduling policy. Yet it is easier and cheaper to implement and has fewer issues with race conditions than its counterpart.

2.1 Various Scheduling Algorithms

Generic algorithms not suitable for Real-Time scheduling:

- First Come First Served (FCFS)
- Shortest-Job-First scheduling (SJF)
- Priority scheduling
- Round Robin (RR)
- Multilevel Feedback Queue (MLFbQ)
- Completely Fair Scheduling (CFS)

Real-Time specific algorithms:

- Priority based
- Rate Monotonic (RM)
- Earliest Deadline First (EDF)
- Proportional Share Scheduling (PSS)

2.1.1 First Come First Served

Processes that request the CPU first are scheduled first.

2.1.2 Shortest-Job-First scheduling

The process with the shortest next CPU burst is scheduled first.

2.1.3 Priority Scheduling

A special case of SJF where processes with higher priority are scheduled first. It can either be preemptive or non-preemptive. A major problem with this algorithm is **starvation**. A solution to this problem is **aging**, which increases a process' priority after a certain amount of time.

2.1.4 Round Robin

Designed for time-sharing systems, similar to FCFS scheduling, but preemptive. The **time quantum** or **time slice** is a process' maximum execution time.

2.1.5 Multilevel Feedback Queue

Processes are divided into **interactive foreground** and **batch background** which run in their own multilevel queue.

3 Synchronization

M Diallo

Processes can execute concurrently. Due to this concurrent access to shared data can and likely will occur. When this happens it is crucial that data consistency is maintained. A good problem to illustrate this is the consumer-producer problem. The consumer consumes unless the buffer (stock) is 0. The producer consumes until the buffer is full. Now consider the following code for the producer and consumer respectively.

```
int counter = 0;
while (true) {
    while (counter == BUFFER_SIZE) {} /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer

```
while (true) {
/* consume item */
    while (counter == 0) {} /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

This looks correct, but it is subject to race conditions. Suppose the producer gets pre-empted after the addition but before storing counter. Then the consumer may read an old counter or even worse: update counter with a bogus value. For example:

```
p | reg1 = load(cnt)   cnt = 5 // loaded from memory
p | reg1 = reg1 + 1    cnt = 5
c | reg2 = load(cnt)   cnt = 5 // c loads old value
p | store(cnt) = reg2  cnt = 6 // p updates value
c | reg2 = reg2 - 1    cnt = 6
c | store(cnt) = reg2  cnt = 4 // oops, overwritten by c
```

Observe that we needed only 6 assembly instructions. The problem is that our implicit critical section has been reordered messing up data consistency. We make it explicit in the next sections.

3.1 Solving concurrency problems

There are several conditions for a solution.

3.1.1 Mutual Exclusion

Mutual exclusion states that when a process is in its critical section, no other process can be executing their critical sections.

3.1.2 Progress

If no process is executing in its critical section and there exist some process that wish to enter their critical section, then the selection of the next process to go into its critical section may not be postponed indefinitely. Meaning a selection must be made.

3.1.3 Bounded waiting

Bounded waiting means that once a process A has indicated it would want to enter its critical section. There must be a bound on the amount of times other processes may enter their critical sections before the process A is granted permission to do so. This essentially means that the waiting time is limited.

3.1.4 Peterson's algorithm

```
bool flag[n] = {false, false};
int turn;
```

This algorithm uses two variables, turn and flag. The flag is set for the process that wishes to enter its critical section. So if for example there are two processes 0 and 1 and 1 wants to enter its critical section flag[1] would be set to true. Additionally if both 0 and 1 want to enter their critical section, flag[0] and flag[1] would be set to true. **The flag variable is simply an indication of which process has indicated it would like to enter their critical selection, it does not decide if a process is allowed to do so.** The turn variable indicates the number of the process which essentially has control over whom enters the critical section. Say that both process 0 and 1 want to enter their critical section and the turn flag is set to 0, then only process 0 will be allowed to enter its critical section. Essentially process 0 can then continue doing as it wishes in its critical section, eventually once it is done, it may set the turn variable to 1 to give process 1 permission to enter its critical section. Along with that, it will also set its own flag back to false.

3.1.5 Hardware Solution: Atomic Instructions

A hardware solution uses an atomic lock. Atomic means that it is a non-interruptible instruction. The following instructions can be considered **release and acquire.** Most of these instructions are quite straightforward. By using these instructions, one can adhere to the three objectives: **mutual exclusion** by not allowing two processes to be in their critical section. **Progress** as a process done with their critical section will release its lock and **bounded waiting**.

3.2 OS support for synchronization

The previous hardware solution is not very accessible. As such, most operating systems provide higher level means for synchronization. Namely Mutex/Lock and semaphores. A couple of problems arise from these uses though, deadlocks, starvation and priority inversion.

3.2.1 Mutex Locks

Operating system designers have built software tools to solve the critical section problem. Of all those solution the mutex/lock is the simplest. The mutex/lock protects critical regions with it by first calling acquire() and afterwards once done calling release(). To indicate whether a lock is available, a simple boolean variable is utilized. The important points to note are that calls to acquire/release **must be atomic.**

3.2.2 Semaphores (yay Dijkstra)

In the essence a semaphore is simply a variable. As with the previous solutions, they're used to control access to a common resource. As opposed to the previous solution, the implementation is also slightly easier: They have two standard operations, wait and signal. Semaphores can be separated into two types **binary** and **counting** semaphores. A good analogy of the counting semaphores is as follows:

> Suppose a library has 10 identical study rooms, to be used by one student at a time. Students must request a room from the front desk if they wish to use a study room. If no rooms are free, students wait at the desk until someone relinquishes a room. When a student has finished using a room, the student must return to the desk and indicate that one room has become free.
>
> In the simplest implementation, the clerk at the front desk knows only the number of free rooms available, which they only know correctly if all of the students actually use their room while they've signed up for them and return them when they're done. When a student requests a room, the clerk decreases this number. When a student releases a room, the clerk increases this number. The room can be used for as long as desired, and so it is not possible to book rooms ahead of time.
>
> In this scenario the front desk count-holder represents a counting semaphore, the rooms are the resource, and the students represent processes/threads. The value of the semaphore in this scenario is initially 10, with all rooms empty. When a student requests a room, they are granted access, and the value of the semaphore is changed to 9. After the next student comes, it drops to 8, then 7 and so on. If someone requests a room and the resulting value of the semaphore

> would be negative,[2] they are forced to wait until a room is freed (when the count is increased from 0). If one of the rooms was released, but there are several students waiting, then any method can be used to select the one who will occupy the room (like FIFO or flipping a coin). And of course, a student needs to inform the clerk about releasing their room only after really leaving it, otherwise, there can be an awkward situation when such student is in the process of leaving the room (they are packing their textbooks, etc.) and another student enters the room before they leave it.

It is important to note that by using counting semaphores, one only knows how many resources are free, it does not tell you which. A binary semaphore is simply a semaphore with two values, 0 and 1 (true or false) it acts as a plain lock.

3.2.3 Busy Waiting

Remember that a process can be in the running state. What happens when a resource it wants access to is locked? Assuming protection measures are in place it will continue waiting until said resource is available. **Busy waiting** occurs when the process is waiting while consuming CPU time. This prevents other processes, that are not in a waiting stage from using CPU time. In modern operating systems, system calls are available to prevent this from happening.

3.2.4 Dining Philosophers

3.2.5 Deadlocks

A deadlock is a state in which each member of a group is waiting for another member to take action, yet no progress can be made. There are four mandatory conditions that must be fulfilled simultaneously to exhibit a deadlock:

- Mutual Exclusion - At least one resource must be held in a non-sharable mode. Otherwise no process would be prevented from using the resource
- Hold and wait/Resource holding - A process is holding at least one resource and is requesting an additional resource held by another process.
- No Preemption - A process may only be released voluntarily be the process holding it
- Circular wait - Each process must be waiting for a resource held by another process. Which in turn is waiting for the first process to release the process.

3.2.6 Starvation

> Starvation is the name given to the indefinite postponement of a process because it requires some resource before it can run, but the

resource, though available for allocation, is never allocated to this process.

3.2.7 Priority Inversion

"Priority inversion is a scenario in which a process with a higher priority is indirectly preempted by a lower priority task" this gives the term priority **inversion.** This can occur when two tasks are running: H(priority 3) and L (priority 2). Either of those two tasks can acquire the use of a shared resource R. if H attempts to acquire the resource R after L, it means H will have to wait for L to release the resource, even though H has a higher priority than L. A good design choice would prevent this from happening by having L release the resource R as soon as a higher priority task requires it.

It is worth noting that priority inversion is what caused a bug within the mars lander.

4 Real Time

M.Diallo

A real time system is a system whose specifications require logical (correct) and temporal (time) correctness. A real-time operating system(RTOS) is an operating system for those kind of systems. When one thinks of real-time systems one can think of high-risk systems such as the ones used in airplanes, nuclear reactions and self driving cars. These systems experience high-reliability/fault-tolerance requirements. Often, they also suffer from harsh environment constraints.
Two forms of real time systems can be distinguished: **Soft** and **hard** deadline systems. A hard deadline should always be met, even under worst case scenarios. A soft deadline is more flexible, occasionally missing a deadline is not a significant problem.
Some misconceptions about hard real-time are: they are fast, they are obsolete because of multicore processors, the critical sections are written in assembly or that real-time systems are related to performance. All of these are false.
In a RTOS, tasks are implemented as threads. They communicate with other tasks, may use system resources and have **timing constraints.** Three forms of tasks that can be distinguished: **periodic, aperiodic and sporadic**. A **periodic task**, as the name indicates is a task that repeats with a given period and has a hard deadline. An example of a periodic tasks is a blood pump system. An **aperiodic** task is a task that is event driven, it may have either soft, hard or no deadline at all. Event driven means that is triggered by an event, for example the wheel control of airplanes would be an aperiodic task with a hard deadline. Finally, there is **sporadic** tasks. These tasks have random release times and a hard deadlines. These tasks are often activated by an interrupt.

Bookkeeping is done in a **Task Control Block** (TCB) structure. In which the possible state values (ready, running,waiting,terminated) are stored. Along with that the TCB parameters are also:

- ID
- Priority
- Pointer to code, data,stack resources
- pointer to other TCB such as preceding, next and waiting queues.

4.0.1 Timing constraints and multi-threading

Given a set of tasks in a queue, one checks if all deadlines can be met. If so, construct a **feasible schedule** to meet those deadlines and construct an **optimal schedule** with minimized response times.

4.1 Real Time scheduling

Determines the order of RT task executions. There are two methods for doing so **dynamic priority scheduling** where the schedule is computed at run time.

Or **static priority scheduling** where the schedule is computed at compile time for **ALL possible** tasks.

4.1.1 Earliest Deadline First (EDF) Horn's algorithm

Given a set of N independent tasks, always execute the task T with the earliest deadlines. Tasks may arrive at any time, and this is dynamic priority scheduling. The schedule is compiled at run time. Given a tuple (C,T) where C represents the execution time of a tasks and T the time period in which the deadline needs to finish. .... There are two forms of these scheduling, **preemptive and non-preemptive**. In the preemptive method, if a task arrives with an earlier deadline than the currently running task, this is interrupted and a context switch occurs. In the non-preemptive method, tasks are running until completion. This is easy to implement as there is no context switches, however, this is not optimal as the running time of a task may exceed the deadline of another task that arrives during execution.

4.1.2 Rate Monotonic (RM) scheduling

This is a static priority scheduling method. The priority of a task is a monotonically decreasing function of its period. Tasks with a shorter period are given higher priority. As such, this scheduling method executes the jobs with the shortest period first. The chance occurs that a task will miss its deadline though.

4.1.3 Priority Inversion

"Priority inversion is a scenario in which a process with a higher priority is indirectly preempted by a lower priority task" this gives the term priority **inversion.** This can occur when two tasks are running: H(priority 3) and L (priority 2). Either of those two tasks can acquire the use of a shared resource R. if H attempts to acquire the resource R after L, it means H will have to wait for L to release the resource. However, L can be prevented from executing its critical section due to a higher priority process from existing.

A good design choice would prevent this from happening by having L release the resource R as soon as a higher priority task requires it.
It is worth noting that priority inversion is what caused a bug within the mars lander.

4.1.4 Priority inheritance

Priority inheritance is a method for eliminating unbounded priority inversion. A scheduling algorithm will increase the priority of a process P to the maximum priority of any other process waiting for any resource on which P has a lock. This means P will execute its critical section at a temporary elevated priority.

4.1.5 Deadlocks in priority inversion

Consider three processes: A B C with priorities in ascending order. Assume A locks resource P and eventually is pre-empted by process B. A is now suspended without having released the lock. Now suppose process C pre-empts B and requires access to resource P but locks resource P2. A deadlock now occurs if both tasks proceed to access each other's resources. The solution to this is to prevent nested locks/counting semaphores.

4.1.6 Schedulability Utilization

A set of tasks is **schedulable** if all tasks are guaranteed to meet their deadlines. **Utalization bound (UB)** test says that a task is scheduled if its total utilization is less than a bound called the Liu & Layland bound. This works under a number of assumptions:

- The Processor always executes the highest priority task
- Task priorities are assigned according to rate monotonic policy (see earlier)
- Tasks do not synchronize with each other
- Each task's deadline is at the end of its period
- Tasks do not suspend themselves in the middle of computations - context switches between tasks take zeoro time

The formula do determine if something is schedulable is as follows where C_i is the computation time, P_i is the release period, and n is the number of processes to be scheduled.

$$\sum_{i=1}^{n} \frac{C_1}{P_i} \leq n * (2^{1/n} - 1)$$

What this formula says that a set of tasks is schedulable if: The sum of each task's computation time divided by it's release period is $\leq$ the right hand side formula. However, this is not definitive. Cases might exist where this formula does not yield a correct answer.

4.1.7 Calculating the worst case response time

4.1.8 Some Real Time Operating Systems(RTOS)

- QNK - microkernel used on blackberry playbook tablet. Often used in satellite software and military stuff.
- MicroC- OS3
- Windows CE - used in windows mobile, cash registers, ticket machines. It's also open source. People laugh when it's mentioned as a RTOS
- RTAI - open source

5 Memory Management

B.Groskamp

To schedule multiple processes on the CPU, we also need to share the available memory between these processes without allowing the processes to access data that does not belong to them. The algorithms needed to achieve this can get quite complicated and often need hardware support, leading many systems to have closely integrated hardware and operating-system memory management.

5.1 Background

Whether the CPU wants to access data or the next instruction, data in the form of an array of bytes needs to be loaded from memory. For managing these memory requests, it doesn't matter how the stream of memory addresses is generated or to what kind of data it leads; we are only interested in the sequence of memory addresses generated by the running program.

5.1.1 Basic Hardware

Main memory (RAM) and the CPU registers are the only general-purpose storage that the CPU has direct access to. Since memory access is extremely frequent and accessing RAM is very costly compared to the registers, most hardware also has a hardware-controlled cache between the CPU and the RAM. Any data that is not in this storage (but for example stored on the hard disk) needs to be moved to the RAM or the registers first before the CPU can utilize it. To prevent user-programs from accessing memory belonging to the OS or other users, dedicated hardware is added to limit programs to their own memory space. One implementation to do so is two special registers, a **base-** and a **limit register**, both only accessible and writable by the OS. The base register holds the start of the memory space for a program and the limit register holds the size of the range of the memory space. When the hardware detects a read-/write to an address outside of this memory space, a trap will be sent to the OS, usually resulting in a fatal error. Only the operating system can access all memory since it's the only program running in kernel mode.

5.1.2 Address Binding

The processes that are waiting to be loaded from disk for execution form the **input queue**. When the program finishes execution, its memory space will be declared available. Since most systems allow a program to reside anywhere in memory space, addresses for the program usually cannot be determined at compile time. The compiler therefore uses symbolic addresses which can then be **bounded** to an absolute address. The binding of symbolic addresses to absolute memory addresses can be done at three steps:

- **Compile time.** If you know where the program will be in memory at compile time, absolute addresses can be generated from the start. If the program would move in memory, the program would need to be recompiled.

- **Load time.** If the memory location isn't known at compile time, the compiler generates **relocateable code**. The final binding will be delayed until load time and when the code would need to move in memory it would just have to be reloaded.
- **Execution time.** If the program needs to be moveable during execution, binding needs to be delayed until execution of the code. Special hardware must be available to make this possible.

5.1.3 Logical Versus Physical Address Space

An address generated by the CPU is called a **logical (or virtual)** address and the address seen by the memory unit is called a **physical address**. For compile-time and load-time address-binding, these addresses are the same. However, for execution-time binding, these addresses can differ. The run-time mapping from virtual to physical address is done by a special hardware device called the **memory-management unit (MMU)**. For a simple scheme using this MMU, we rename the base regsiter to the **relocation register**. The value in this register is now added to every address generated by a user process by the MMU. The user program never sees the actual physical addresses and only deals with virtual addresses. The concept of binding a virtual address space to a separate physical address space is central to proper memory management.

5.1.4 Dynamic Loading

To not limit the size of a program to the size of the main memory, we can use **dynamic loading**. With dynamic loading, a routine only gets loaded into memory when it is called. Code that has to be executed infrequently, like error routines, only has to be loaded when the error occurs, which can result in a significantly lower memory footprint for the program. Although the OS may help with dynamic loading by providing libraries, dynamic loading has to be implemented by the program itself.

5.1.5 Dynamic Linking and Shared Libraries

Most programs make use of system libraries. Instead of packing each program with their own copy of these libraries, **Shared libraries** can be used. At compile time, a **stub** is included in the program, containing instructions on how to include the library. At execution, the stub gets replaced by the OS with a pointer to the library. This way, all programs using a specific library point to one copy of the library. Other than saving memory space, this technique also prevents having to recompile the program when a new version of the library is released.

5.2 Swapping

A program can only be executed when in main memory. When there are a lot of processes running on a computer, it might be the case that there is not enough main memory to store all processes. To solve this, we can use **swapping**. With swapping, we temporarily move an idle process to the **backing store**, typically a HDD or SSD. This frees space in the main memory, allowing another process to be executed.

5.2.1 Standard Swapping

Since the OS needs to know the size of each process in order to swap, it is necessary that the user tells the OS when memory is released or freed (by using **malloc** or **free**, for example). When swapping, the OS also needs to be sure whether a process is completely idle or not; when a process is waiting for I/O, it cannot be swapped out. Possible solutions for this are to never swap processes that are awaiting I/O or to send the I/O data to an OS buffer and copy the data from the buffer to the process when the process gets swapped in again, also called **double buffering**. Since the backing store is usually fairly slow and swapping can add a lot of overhead, operating systems usually don't always and fully swap processes, but only swap when the available memory is below a certain threshold and/or only swap parts of processes.

5.2.2 Swapping on Mobile Systems

Mobile devices usually use less spacious flash drives as persistent memory instead of HDDs. Because of the limited amount of space and the limited number of writes that flash memory can handle before getting damaged, mobile operating systems don't swap. For example, both iOS and Android *ask* processes to release memory when the available memory drops below a threshold and kill processes if necessary. When killing the process, Android does however write the process state to the flash memory to easily restore the state of the program.

5.3 Contiguous Memory Allocation

The main memory must accommodate both the operating system and user processes and allocate the memory as efficient as possible. One early way called **Contiguous Memory Allocation** was to divide the memory into two adjacent partitions; one for the OS and one for user processes. Each process, be it an OS or user process, is contained in a single section of memory in the appropriate partition, next to the other processes. Since the interrupt vector is usually stored in lower memory, the OS partition usually is too, leaving the higher, remaining part of the memory for user processes.

5.3.1 Memory Protection

Using the limit register (section 5.1.1) and the relocation register (section 5.1.3) we can safely allocate and relocate memory without the risk of processes reading each others memory. The relocation register also allows us to dynamically change the size of the operating system. For example, the OS contains a lot of drivers and libraries that may not be needed at all times. This code can usually be kept out of main memory and transferred into memory only when needed; it comes and goes as needed. This type of code is sometimes called **transient** operating system code.

5.3.2 Memory Allocation

One of the simplest ways to allocate memory is to divide memory into fixed-size **partitions** which may each contain exactly one process. In this **multi-parition method**, the amount of processes is limited to the amount of partitions. When a process finishes, its section is released and the next process from the input queue is loaded into memory. This method is, however, no longer in use due to its obvious problems.

In a **variable-partition** scheme, the OS keeps a table indicating which parts of memory are available or occupied. Initially, all available memory is seen as one large block of available memory, also called a **hole**. Eventually, memory contains a set of holes of various sizes. Memory is allocated to processes until the memory requirements of the next process cannot be met anymore, e.g. there is no hole available that can store this process. The OS can at that time either wait till a large enough block is freed, or it can skip down the input queue in search of a process that would still fit. When a process gets picked from the input queue, the system searches the set of holes for a hole that is large enough for this process. If the hole is too large, the hole gets split into two smaller holes of which one is allocated to the process. When a process finishes and its memory gets freed, the system will try to recombine the hole with adjacent, free holes to form one bigger hole.

This procedure is a particular version of the general **dynamic storage allocation problem**, which concerns how to satisfy a request of size n from a list of free holes. The most common strategies for this problem are the following:

- **First fit**. Start searching from the beginning or the end of the holes and allocate the first fitting hole.
- **Best fit**. Search the entire list of holes and allocate the smallest fitting hole.
- **Worst fit**. Search the entire list of holes and allocate the largest hole. This produces larger leftover holes than best fit, which may be more useful.

First fit and best fit both perform better than worst fit in terms of decreasing time and storage utilization. First fit and best fit both perform about the same in terms of storage utilization, but first fit is generally faster.

5.3.3 Fragmentation

Both first fit and best fit suffer from **external fragmentation**; they both result in a lot of smaller, available blocks that cannot fulfill allocations, even though there is enough total memory space available. Depending on the total amount of memory available and the average process size, one-third of memory may be lost to fragmentation. This is also known as the **50% rule** (because for every N blocks allocated, 0.5N blocks are lost). A solution to external fragmentation is **compaction**, where we shuffle around memory so the free blocks can form one bigger, contiguous block. This does however have a large overhead and only works if all processes can be relocated dynamically. Another solution is to permit the logical address space of the processes to be non-contiguous, allowing a process to be allocated wherever there's available memory. To avoid breaking up memory in too many blocks, there is usually a fixed block size, for example 256 bytes. If we were to allocate 257 bytes, we'd need two blocks of 256 bytes, meaning we waste 512-257 = 255 bytes on this allocation. This issue is called **internal fragmentation**.

5.4 Segmentation

Segmentation provides a mechanism that maps the programmer's view of how memory works to the actual physical memory, allowing the system to manage memory better and to provide a more natural environment for the programmer.

5.4.1 Basic Method

Segmentation divides the program into several segments with varying purposes and variable length. A logical address space is a collection of segments, each with their own name and length. The addresses contain both the segment name and the offset within the segment. Segments are numbered and are referred to by their number, so a logical address consists of the tuple *¡segment-number, offset¿*. A C compiler might create separate segments for the following:

- The code
- Global variables
- The heap, from which memory is allocated
- The stacks used by each thread
- The standard C library.

5.4.2 Segmentation Hardware

To map these 2-dimensional tuple addresses to 1-dimensional physical addresses, a **segment table** is used. Each entry in this table has a **segment base** containing the physical starting address of the segment and a **segment limit** specifying the length of the segment. When looking up an address, the segment number is used as an index for the segment table and the offset needs to be lower than or equal to the segment limit or a trap will be sent to the OS. If successful, the physical address gets returned.

5.5 Paging

Paging is a memory-management scheme that allows memory to be allocated non-contiguous while avoiding external fragmentation. It also solves the problem of storing memory chunks of variable sizes in the backing store. The backing store has the same fragmentation issues as the main memory, but compaction is impossible due to the slow access times.

5.5.1 Basic Method

To implement paging, we break up physical memory into fixed-size blocks called **frames** and break up logical memory into blocks of the same size called **pages**. The backing store is divided into fixed-size blocks with sizes equal to one or more frames. This completely separates the logical address space from the physical address space. Every address generated by the CPU consists of a **page number** and a **page offset** which can be used to lookup a page in the **page table**. This page table contains the base physical addresses of the pages. The resulting address of the lookup in this table is sent to the memory unit. Page sizes are defined by the hardware and a power of 2, varying from 512 bytes to 1GB.

Because the paging hardware basically allows for a form of dynamic relocation, there is no external fragmentation. If the process-size is independent of the page size, we can expect about halve page of internal fragmentation per process. To minimize internal fragmentation, it would make sense to keep the page size low. This does however add a lot of overhead and disk I/O is more efficient when more data is transferred, so page sizes have in fact grown over the years.

An important aspect of paging is the clear difference in how a programmer thinks about the memory and how the physical memory actually works; the programmer thinks as the memory as one block of memory especially for his program, while his program might actually be scattered throughout the memory together with other processes. The address-translation hardware enables this way of working with memory.

Since the OS is in charge of managing memory, it has a **frame table** containing information about which physical page frames are available and which ones are occupied. To make sure the OS can also handle system calls with addresses (for example I/O), it also has a copy of the page table of each process to translate these addresses.

5.5.2 Hardware Support

How the page tables are stored depends on the hardware and OS; in the simplest case, the page table is implemented as a set of **registers**. This can be very fast but is only feasible for page tables smaller in size. However, modern computers can allow page tables to grow to millions of entries. Instead of storing the complete page table in the registers, we can store a pointer to the table in a special register called the **page-table base register (PTBR**. This allows for faster context-switches between processes since only 1 register needs to be updated, but does double the amount of memory accesses needed (because the page table needs to be accessed first before the actual data can be retrieved).

To solve this, we can use a special hardware cache called the **translation look-aside buffer (TLB)** which contains a few of the page table entries. The TLB-lookup is part of the instruction pipeline in modern hardware and adds basically no performance penalty. When the CPU calculates an address, its page number is immediately presented to the TLB. If the page number is present, its frame number is returned. If the page number is not present (known as a **TLB miss**), the full page table still needs to be accessed and the page number and frame number are added to the TLB. If the TLB is already full, either the OS or the hardware itself decide to remove an entry. To prevent entries from being removed, for example entries for key kernel code, some TLBs allow to **wire down** an entry so it cannot be replaced. Some TLBs store **address-space identifiers (ASIDs)**, unique ids for the process using the page, in each entry. This adds another layer of protection for the address-space of a process. When a process tries to lookup a page number of another process, the TLB will treat it as a TLB miss. If the TLB does not support ASIDs, it needs to be **flushed** on every context switch. The percentage of times that the TLB can provide the page number is called the **hit ratio**. The **effective memory-access time** is dependent both on the memory speed and the hit ratio of the TLB.

5.5.3 Protection

To prevent read-only pages from being read, protection bits can be added to the page table. To provide even better protection, we can add more bits for specific features or actions. One bit is generally attached to each entry in the page table: the **valid-invalid** bit, indicating whether the page is in the process' memory space and preventing access to pages outside of its memory space. Since most processes use far less than the total available memory space, it would be wasteful to keep page table entries for every possible page. Because of this, some systems provide hardware called a **page-table length register (PTLR)** to indicate the size of the page table.

5.5.4 Shared Pages

An advantage of paging is the ability to **share** pages of code between processes, as long as the code is **reentrant** (non-self-modifying). This way, heavily used programs like window systems and run-time libraries can be shared among all processes. The operating system should however still make sure that none of the processes can edit these shared pages, unless the page is meant for communication between processes.

5.6 Structure of the Page Table

5.6.1 Hierarchical Paging

Since most modern computers allow very large logical address spaces, the page table in these systems would become excessively large too. Because we don't want to keep such a huge table fully in main memory, we can use **forward-mapped paging (also known as *two-level paging*)**. With this scheme, the page table itself (the inner page table) is also paged and we only keep the page table for the inner page table (the outer page table) in main memory. Logical addresses consist of tree parts: the page number of the outer page, the page number of the inner page and the page offset. When the memory and the amount of pages increases, it is of course possible to extend this scheme with even more levels of paging.

5.6.2 Hashed Page Tables

A common approach to handle address spaces longer than 32 bits is to use a **hashed page table**, with the hash value being the virtual page number. Every entry of the table contains a linked list with all elements with the same hash. When a memory request is made, the virtual page number is hashed, compared to the table and the corresponding list of elements is traversed to find the right page frame to form the physical address. A variation of this scheme that allows for 64-bit address spaces is called **clustered page tables**, which are similar to hashed page tables except that each table entry references to multiple elements instead of just one. This scheme is particularly useful for **sparse** address spaces, where memory references are scattered throughout the address space.

5.6.3 Inverted Page Tables

Usually, each process has its own page table. These page tables match every virtual address to a physical address, regardless if the process can access those address spaces. This adds a lot of unnecessary redundancy, since each physical address has multiple references to it. A solution for this is to use one **inverted page table**, that maps real frames to virtual memory addresses. Each entry consists of the page number and the address-space identifier (section 5.5.2). This way, only table has to be kept in memory instead of a table per process. It is, however, slower to search this table since it is ordered by frame and not by virtual address, even though the lookup is done using the virtual address.

6 Virtual Memory

D. Kroeb

6.1 Background

To ensure a more reasonable response time, **virtual memory** can be used. Virtual memory is a technique that allows the execution of a process that is not completely in memory.

The great advantage of this: Run programs that are larger than **actual physical memory**. Also note: allows for efficient process creation.

It seperates the memory into logical and physical memory. This is done by storing memory into a large uniform array. This also makes things easier for programmers as it removes (some) concerns over memory-storage and its limitations.

Virtual Address Space of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this logical address begins at 0. Virtual address space that has holes in it is called a **sparse address space**. This is actually beneficial, because the gaps can be later filled (when the heap or stack grow up or down, respectively).

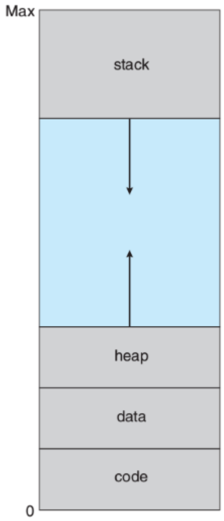


Figure 9.2 Virtual address space.

The **Memory manangement unit (MMU)** organises the page frames such that logical pages are mapped to physical page frames in the memory.

When using the fork() system call, the creation of a process is sped up because pages are shared between the parent and child processes.

6.2 Demand Paging

Demand paging is a technique used to only load pages that are demanded by an executing program. Pages that are not accessed, are not loaded into the **physical memory**. This is similar to a **swapping** system, where secondary memory (like a disk) is used as extra memory (refered to as pagefiles in windows).

Rather than swapping the entire process into memory, though, we use a lazy swapper. **A lazy swapper** never swaps a page into memory unless that page will be needed. In the context of a demand-paging system, use of the term **swapper** is technically incorrect. A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. We thus use **pager**, rather than swapper, in connection with demand paging.

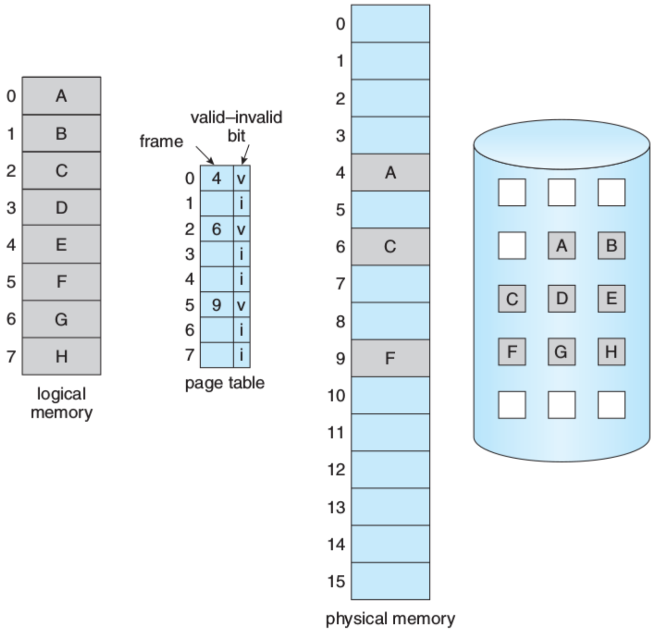


Figure 9.5 Page table when some pages are not in main memory.

We need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. For that we use **The valid –invalid bit scheme**. When this bit is set to “valid,” the associated page is both legal and in memory. If the bit is set to “invalid,” the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk.

Page faults occur when a page marked as invalid is accessed. Please see figure 9.6 below to find out how this is handled.

9.2 Demand Paging 403

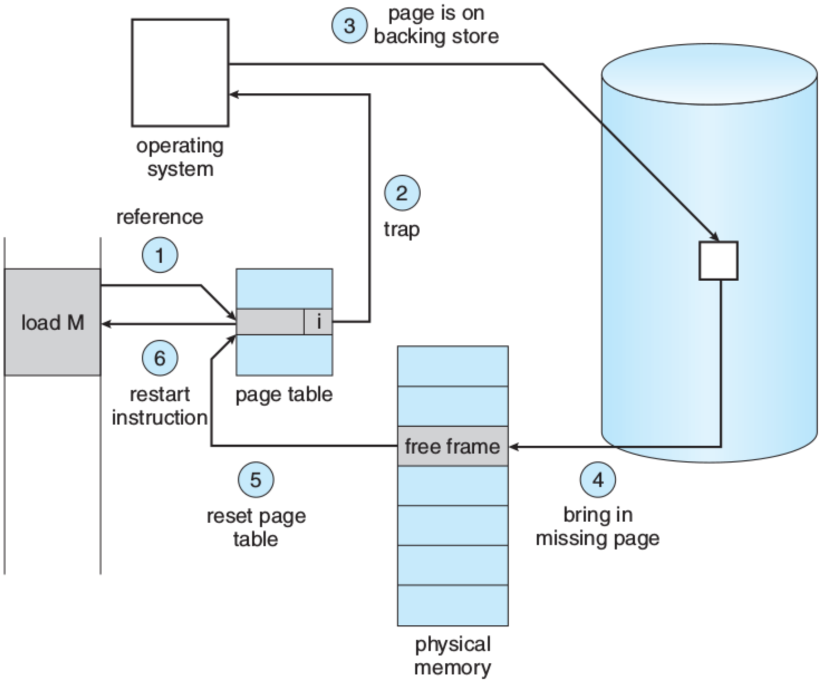
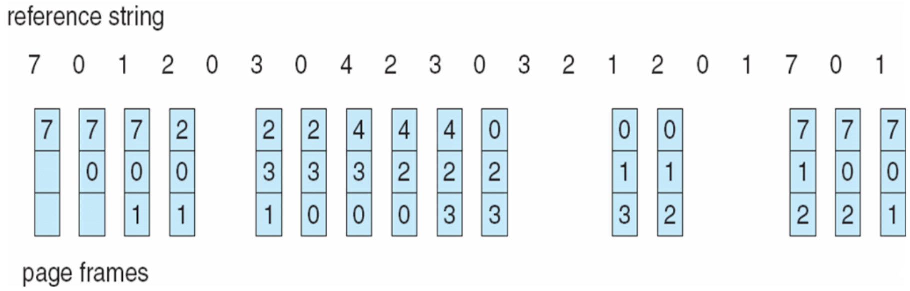


Figure 9.6 Steps in handling a page fault.

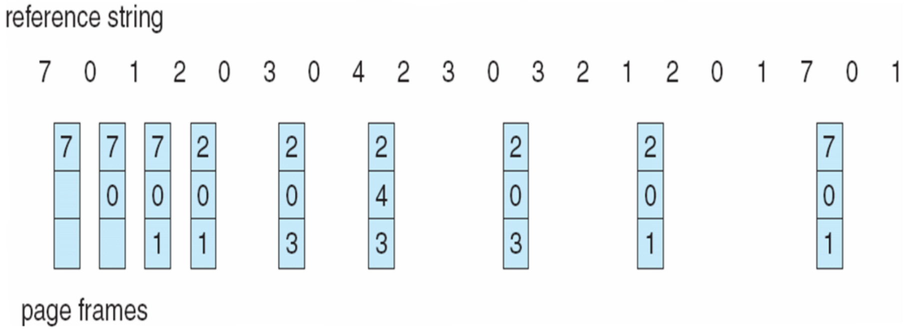
6.3 Effective Access Time and Page Replacement algorithms

Let p be the probability of a page fault ($0 \leq p \leq 1$). Let ma be **memory-access time**.

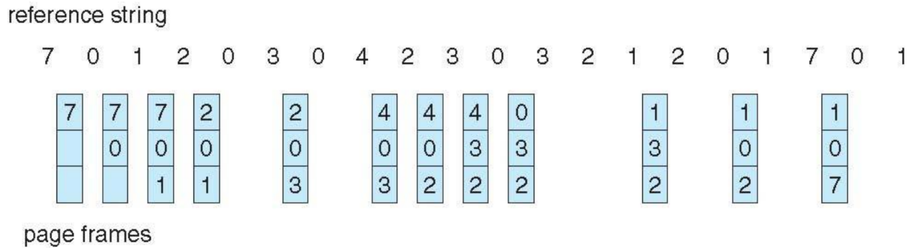
effective access time (EAT) = (1 - p) * ma + p * page fault time
To prevent over-allocation, page replacement is used. This can be done in FIFO order.



Optimal Algorithm: replaces the page which will not be used in a long period of time.



Least Recently Used (LRU) Algorithm: LRU works on the idea that pages that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too. While LRU can provide near-optimal performance in theory (almost as good as adaptive replacement cache), it is rather expensive to implement in practice.



Most Frequently Used (MFU) Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

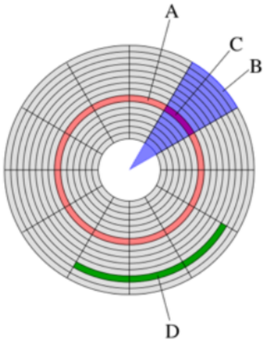
Both LRU and MFU are not common, because the algorithms are expensive and do not approximate OPT (optimal page replacement) replacement well.

7 Storage and file-systems

R.A.J. Wacanno

7.1 Disk structure

The platters of hard disk used for storing files is made up of the following parts:



A: **Tracks**: A continuous ring around the platter containing binary data.
B: **Sectors**: A radial slice of a track with a constant angular size. Its size in bytes depends on the track size in bytes and might also depend on the track radius.
D: **Blocks**: A group of sectors. It size is determined whilst formatting the disk.

Since a disk uses mechanical components to store and retrieve data, accessing a specific part of a platter takes up a certain amount of time. The time needed by the read-write head to travel from one track to another and find a certain sector is called the **seek time**. Accessing a part of a platter involves the following mechanical actions by the read-write head:

- **Speedup**: Arm acceleration.
- **Coast**: Arm moving at maximum speed. This is most prevalent during long seeks.
- **Slowdown**: Arm slows down when in close proximity to the desired track.
- **Settle**: Head is adjusted to reach and access the desired track.

7.2 Interface standards

Over the years a great number of interface standards have been used for communication between the disk and the CPU.

SCSI

Small Computer System Interface has historically been used in Apple Macintosh systems and old Unix systems like the PDP-11. The SCSI connection sends data in parallel and is able to reach speeds up to 80 MB/s. Since multiple SCSI devices can be connected to a singe port, it's considered more to be a bus instead of an interface.

(S)ATA
Serial AT Attachment exists in multiple revisions. Each revision continues to crank up the data rates. The newest revision, SATA 3, is able to reach a data rate of up to 6 GB/s. SATA has become the go-to interface in contemporary PCs because of its relatively low cost and high reliability. This last point is illustrated by its **MTBF** (Mean Time Between Failures) of 700K to 1M hours.

SAS
Serial Attached SCSI is — as the name might have suggested — an evolution of SCSI. Notable properties include:

- ability to connect up to 128 devices;
- data rates of up to 22,5 GB/s with the newest revision (SAS 4);
- support for hot plugging
- and a MTBF of millions of hours.

7.2.1 RAID

RAID (Redundant Array of Independent Disks) is used to increase the performance and/or reliability of computer storage. RAID exist in multiple varieties, 1 up to and including 6, who's specific properties are not within the scope of this summary.

Performance
To increase the performance of a cluster of disks, RAID uses a technique called **striping**. Striping can either occur on bit or block level. In case of of **bit-striping**, the bits which are part of a single byte are spread across multiple disks. This means that writing a byte to 8 available disks using bit-striping would result in a theoretical 8x increase of the access rate.

Block-striping is similar to bit-stripping in that it splits data across multiple drives, but, whereas bit-striping distributes individual bits across drives, block-striping distributes blocks. It usually does this according to this formula: $D = (i\ mod\ n) + 1$; where D is the disk number (1 to n) to which block i should be written.

Reliability
To increase drive reliability, in case of for example disk crashes, RAID employs redundancy. This means that a single piece of data is, in one way or another, stored on multiple disk to ensure data integrity in case one or possibly more of the drives used fails. The following is an example of how data can be recovered when using RAID:

The data blocks `101`, `010` and `011` are stored on drives 1-4.

- D1: `101`
- D2: `010`

- D3: `011`
- D4: `101 XOR 010 XOR 011` = `100` (**parity**)

If, in the example above, disk 2 happens to crash, its contents can be recovered using the following operation: `101 XOR 011 XOR 100 = 010`.

7.3 I/O communication

-

7.4 I/O scheduling

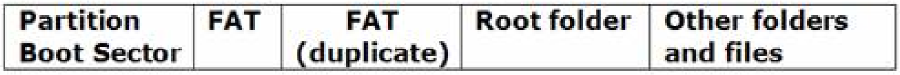
Disk I/O is very expensive. Some algorithms:

First Come First Served (FCFS)	Incoming requests are issued first
Shortest Seek Time First (SSTF)	The next shortest distance request is handled first
Elevator (SCAN)	Goes back and forth like an elevator
Circular Scan (C-SCAN)	Wraps around rather that reversing direction like SCAN
C-Look	Wraps around by going directly to the last request

Note that these are somewhat similar to process scheduling algorithms. Reducing latency is most important for Disk I/O because it is marginally slower than RAM.

7.5 Linked allocation (FAT)

The first large group of file systems is the class of **linked file-systems**. **FAT** (File Allocation Table) has been a historically prevalent member of this group. When using FAT, the disk will be layed-out as follows:

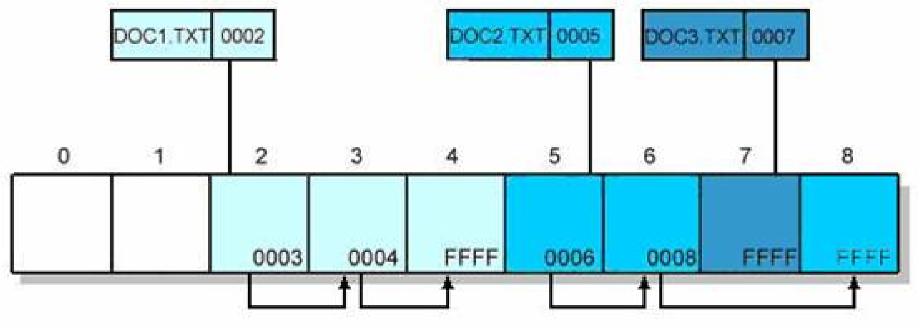


Partition Boot Sector	FAT	FAT (duplicate)	Root folder	Other folders and files

The **boot sector** is used to store crucial information pertaining to the boot process of the system and, thus, isn't allowed to be used by the user. The disk is spit up into **clusters** whose size depends on the size of the volume. Each cluster contains part of a file's or directory's binary data. The number of clusters can not exceed the range of a 16-bit integer (or 32-bit in the case of FAT32) and must be a power of 2. The **FAT** (File Allocation Table) contains entries, each corresponding to one of the clusters, which holds the number of the next cluster in the linked list of clusters which holds the file data. The FAT might be might be stored multiple times on the disk (as seen in the example above), but this depends on the implementation. This entry can have the following values:

- `0x0000`: Unused cluster
- `0x0001-0xFFF6`: Next cluster in the chain
- `0xFFF7`: Bad cluster
- `0xFFF8-0xFFFF`: Last cluster in a file's chain

This link scructure is illustrated in the following diagram:



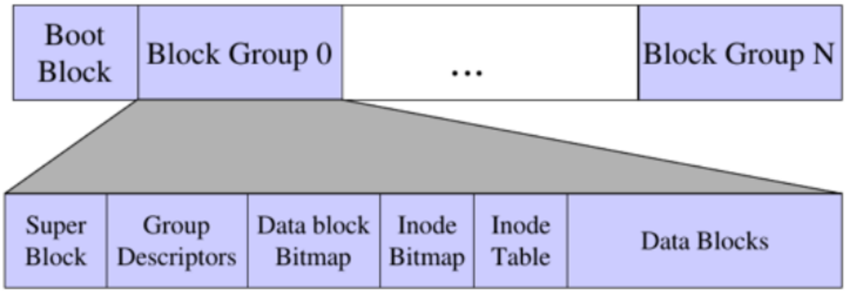
In FAT, each folder contains 32-byte entries for the files and sub-folders it holds. Each entry holds the following information:

- Name
- Attribute byte
- Create time
- Create date
- Last access date
- Last modification time
- last modification date
- Staring cluster number in FAT table
- File size

7.6 Indexed allocation (UFS & ext)

This method of allocation is used in Unix and Unix-derived systems. **UFS** and its descendent **ext** (also revisions ext2, ext3 and ext4) are widely-used examples of this **indexed** method of file allocation. The following explanation will focus primarily on the ext2 file-system.

The ext file-system spits up a given partition into equally sized **block groups**. The following diagram illustrates this division:



The **superblock** contains the lay-out of the file-system and has a copy is each block group. Together with the superblock, each block group also holds a copy of the **block group descriptor table**. The advantage of using this block group structure is reduced fragmentation, since the data of a single file is stored in one block group instead of fragments ending up all over the volume.

The superblock contains the following information:

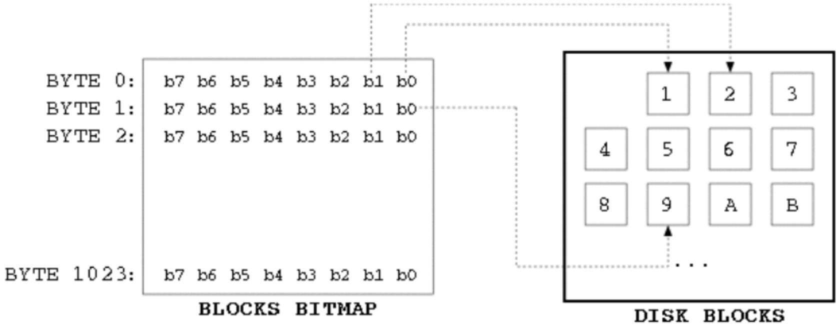
- Size of the file-system
- Number of free blocks
- List of free blocks (+ pointer to free block list)
- Index of the next free block in the free block list
- Size of the inode list
- Number of free inodes in the file-system
- Index of the next free inode in the free inode list

The superblock of block group 0 is always located after the boot block, which, in case of a block size of 4 kb is in block 2 or, in case of a block size of 1 kb, is in block 3.

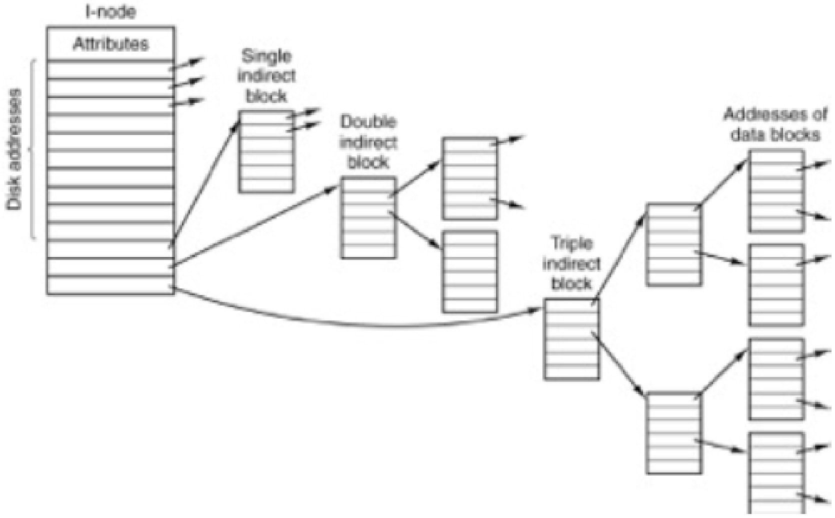
The **block groupt descriptor table** is an array of **block group descriptors** and is positioned after the super block in each block group. Each block group descriptor contains:

- the location of the inode bitmap;
- the location of the data block bitmap;
- a free blocks count
- and a free inodes count.

The **block** and **inode bitmap** are, quite unsurprisingly, bitmaps with one bit for each block and inode respectively. A value of `1` represents a used block or inode and a value of `0` represents a unused one. The size if these bitmaps is equals to $n/8$; where n is the number of blocks or inodes. The following is a diagram illustrating the block bitmap:

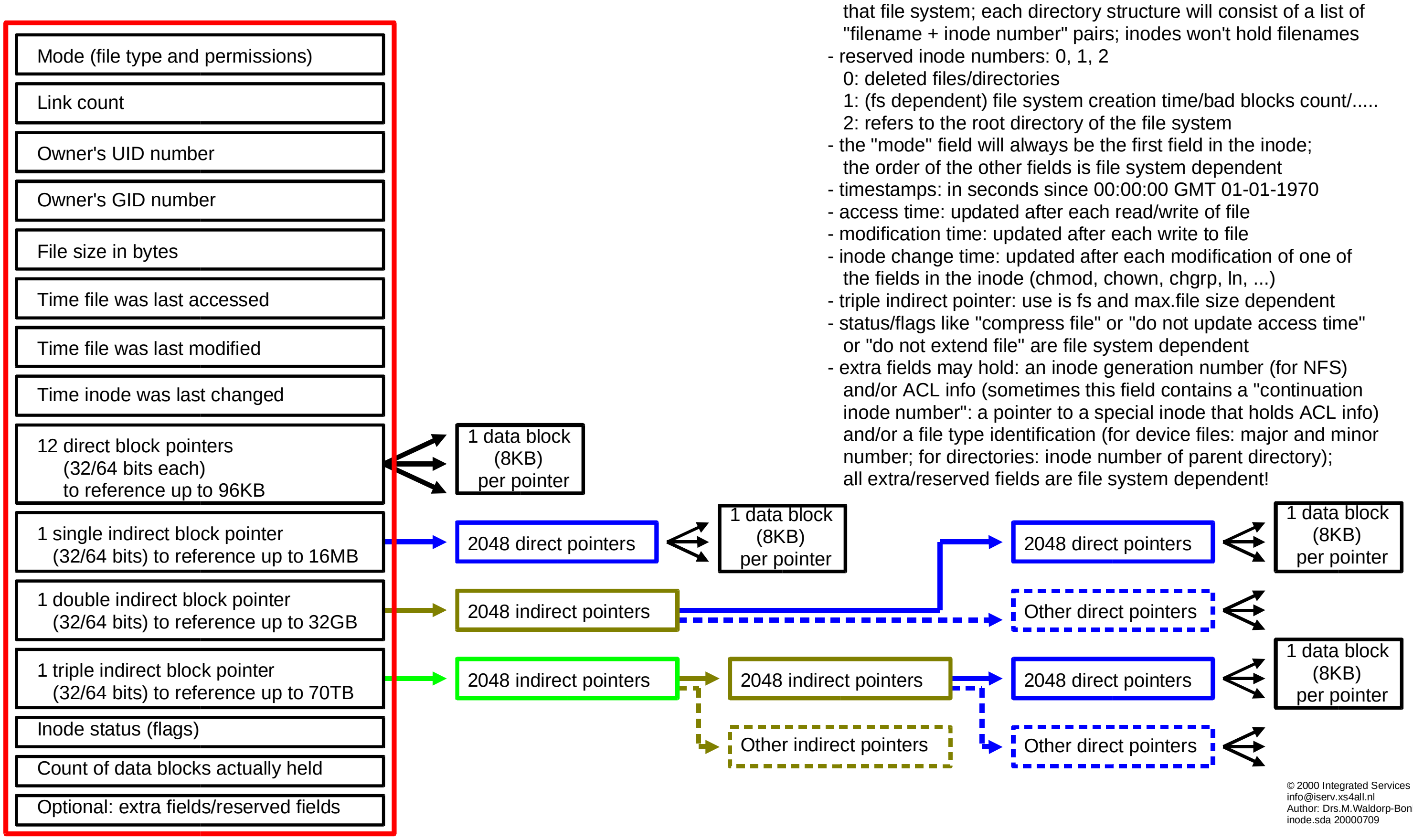


The file-system employs specific blocks to store the addresses pointed to by the **inode** data structures. Each file has its own inode. This pointing can either be done directly from the inode or indirectly through the aforementioned address blocks. Ext allows for a maximum of three levels of indirect pointing which greatly increases the maximum file size. The following diagram illustrates this organization.



UNIX INODE STRUCTURE

Legenda:
- each (unix) file system has its own inode table; on disk each cylinder group will hold a relevant part of that table
- each inode is referenced by a "device + inode number" pair
- each file is assigned an inode number which is unique within that file system; each directory structure will consist of a list of "filename + inode number" pairs; inodes won't hold filenames
- reserved inode numbers: 0, 1, 2
  0: deleted files/directories
  1: (fs dependent) file system creation time/bad blocks count/.....
  2: refers to the root directory of the file system
- the "mode" field will always be the first field in the inode; the order of the other fields is file system dependent
- timestamps: in seconds since 00:00:00 GMT 01-01-1970
- access time: updated after each read/write of file
- modification time: updated after each write to file
- inode change time: updated after each modification of one of the fields in the inode (chmod, chown, chgrp, ln, ...)
- triple indirect pointer: use is fs and max.file size dependent
- status/flags like "compress file" or "do not update access time" or "do not extend file" are file system dependent
- extra fields may hold: an inode generation number (for NFS) and/or ACL info (sometimes this field contains a "continuation inode number": a pointer to a special inode that holds ACL info) and/or a file type identification (for device files: major and minor number; for directories: inode number of parent directory); all extra/reserved fields are file system dependent!


Mode (file type and permissions)

Link count

Owner's UID number

Owner's GID number

File size in bytes

Time file was last accessed

Time file was last modified

Time inode was last changed

12 direct block pointers
(32/64 bits each)
to reference up to 96KB

1 single indirect block pointer
(32/64 bits) to reference up to 16MB

1 double indirect block pointer
(32/64 bits) to reference up to 32GB

1 triple indirect block pointer
(32/64 bits) to reference up to 70TB

Inode status (flags)

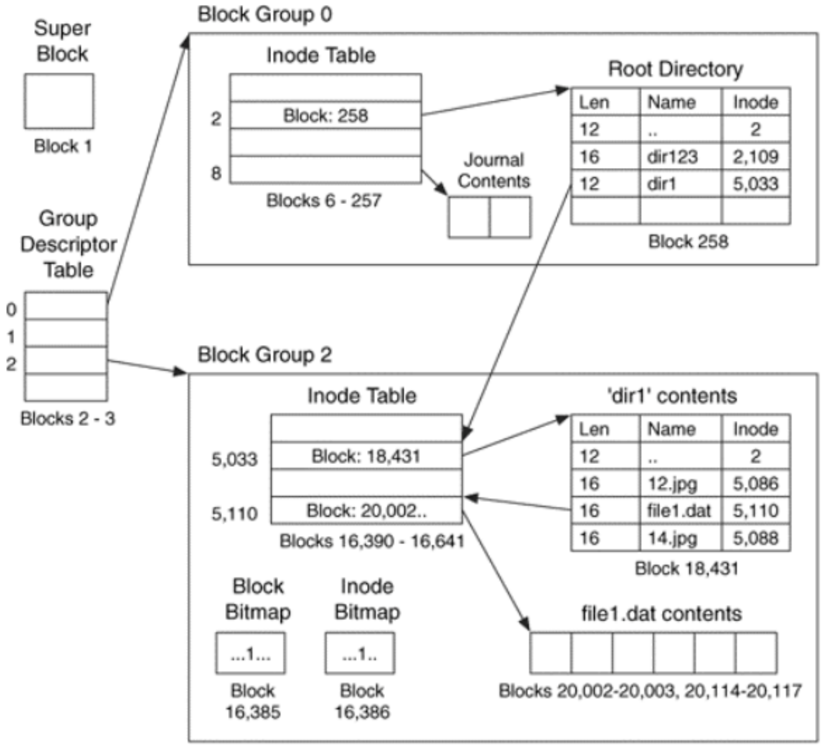
Count of data blocks actually held

Optional: extra fields/reserved fields

© 2000 Integrated Services
info@iserv.xs4all.nl
Author: Drs.M.Waldorp-Bonk
inode.sda 20000709

7.7 Finding data using inodes

The following illustrates the access of the file `/dir1/file.dat` in a indexed file-system:



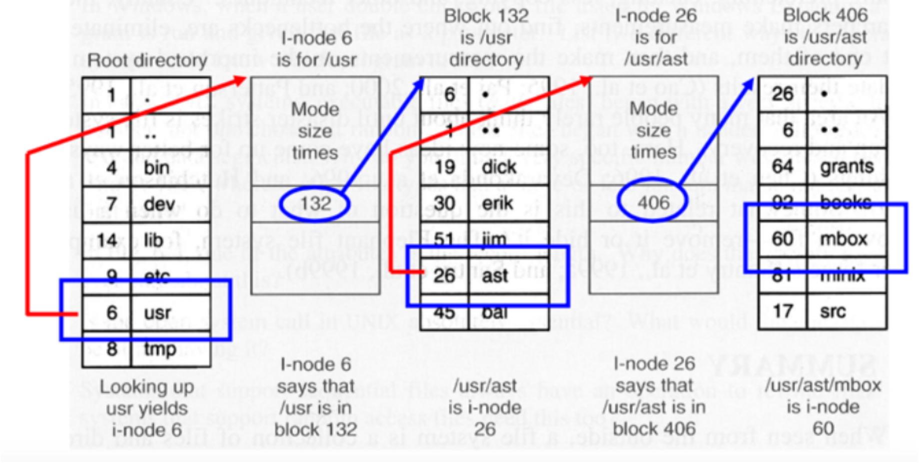
Root Directory

Len	Name	Inode
12	..	2
16	dir123	2,109
12	dir1	5,033

'dir1' contents

Len	Name	Inode
12	..	2
16	12.jpg	5,086
16	file1.dat	5,110
16	14.jpg	5,088

The following illustrates the access of the file `/usr/ast/mbox` in a indexed file-system:



Root directory
I-node 6 is for /usr
Block 132 is /usr directory
I-node 26 is for /usr/ast
Block 406 is /usr/ast directory
Mode size times
132
Mode size times
406
Looking up usr yields i-node 6
I-node 6 says that /usr is in block 132
/usr/ast is i-node 26
I-node 26 says that /usr/ast is in block 406
/usr/ast/mbox is i-node 60

Root directory

1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

Block 132 is /usr directory

6	•
1	••
19	dick
30	erik
51	jim
26	ast
45	bal

Block 406 is /usr/ast directory

26	•
6	••
64	grants
92	books
60	mbox
81	minix
17	src

7.8 Symbolic and Hard links

A relative or absolute link that refers to a file or directory. Windows uses symbolic links for mount points referring to specific directory on NTFS. A hard link is like a reference count and it refers directly to a file or directory. On linux, a hard link can only refer to a file. On mac, it can also refer to a directory. A hard link cannot refer to a file or directory on another file system partition.

7.9 Journaling File-systems

All file system operations are recorded in a journal, similar to a database transaction log. In case of a system crash, it can be rolled back eliminating file system corruption.

7.10 Network file-systems

8 Security

8.1 Security breaches

Security can be breached in several ways. Some of these methods include the following:

- **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 usage of resources.
- **Denial of service**. Prevention of usage of resources.
- **Masquerading**. Masquerading as sender to the receiver.
- **Man-in-the-middle attack**. Masquerading as sender to receiver and vice-versa.
- **Session hijacking**. Interception of an active communication.

8.2 OS hardening

OS hardening is the act of making the OS safer. The gist of OS hardening is:

- Begin clean by installing the base OS via a protected network (do not use default settings)
- Install safety programs (e.g. antivirus, firewall etc.)
- Enable logging
- Remove all unneeded applications, servers and protocols
- Test if system works
- Install needed third party applications
- Test integrity of system after each install
- Document everything
- Allow roll-backs
- Remove default accounts and groups (if unused)
- Root privileges only for those who need it
- Not all users should have access to all resources

8.3 Password control in unix

All encrypted passwords are kept in **/etc/shadow**. The shadow file is only readable and writable by a root user. As such, before logging in the login process has root access. Previously the encrypted passwords were kept in /etc/passwd, which was readable by everyone.
Passwords are encrypted using `crypt(3)`, a certain hashing algorithm and salt. `crypt` applies the used hashing algorithm a certain amount of times (default 5000) to the salt concatenated to the password.

8.4 Key authentication

8.4.1 Secure Shell key

A **Secure shell** (ssh) key enables transmission of sensitive data over a insecure network. The server and clients all have an ssh keypair. The public key can be distributed unencrypted so the other party knows how to encrypt data and sent to the other side. The private key is kept private and is the only key that can decrypt the data. A public key can only be used to encrypt the data.

8.4.2 Key Distribution Centre

A **Key Distribution Centre** (KDC) distributes access to services. Suppose Alice asks a KDC server to provide access to service provider Bob. Using cryptographic techniques, Alice authenticates and the KDC checks if Alice is allowed access. Once granted, Alice receives a ticket from the server based on some server private key. Alice asks Bob for its service and Bob can verify that Alice has been verified and granted access by the KDC.

8.4.3 Kerberos

Identical to KDC except that the Authentication Server (AS) and Ticket Granting Service (TGS) agents are partitioned.

8.5 file permissions

In the old days not many groups needed, pdp-7 and pdp-11 used just owner:group:others

8.5.1 old unix file permissions

read/write/execute access stored in octal number

8.5.2 ACL

users assigned to groups.

8.5.3 SElinux

SE-Linux has been developed by the National Security Agency (NSA) of the USA. The basic idea is that you can *enforce* rules and block access to certain resources *per process and per user*. When in **permissive mode**, it logs accesses it would have denied if it was in **enforcing mode**. SE-Linux can also be **disabled**, but this is bad practise. It is often forgotten to re-enforce SE-Linux after it has been disabled or set to premissive during debugging.

8.5.4 Setuid

A system call that increases privileges for the running process. It is often forgotten to reduce privileges for forked children and therefore a common attack practise.

8.5.5 Getgid

Retrieves group identity that represents group privileges.

8.5.6 Sticky bits

Files in sticky bit marked directories on linux may not be renamed or deleted by non-root users.

8.6 Jailing and Virtualization

It is usually to expensive to bootstrap a whole virtual machine just to run some programs. **Jailing** creates the illusion that the whole file hierarchy starts at another directory.

8.6.1 Chroot vulnerabilities

It is easy to circumvent jailed environment if one can access commands that can escape the environment. Another way is to create a symbolic links that refers to a file or directory outside the jail.

8.6.2 Virtual Machines

Using hardware virtualization, programs can run in virtual machines.

8.6.3 Hypervisor

The **machine monitor** is also called the **hypervisor**. It monitores the machine and

8.6.4 Docker

Docker provides a chroot contained environment using resource isolation, union mounting, cgroups and other means. This eliminates virtual machine overhead because it does not need one.

8.7 Program threats

J.Stalenburg

Most of the threats below need to either be executed by the user or use one of the above methods to do the things they do.

- **Parasitic code/viruses**: Injects itself into another programs to multiply. Can wait for a trigger to activate, a **logic bomb**. Can redirect interrupt handlers (e.g. ctrl-c, SIGINT) or OS functions which accesses disk sectors or specific files.
- **Worms**: Replicates itself without user trigger to spread to other computers in network. As such is harmful to the network (bandwidth). Oftentimes used to install backdoors. Works as follows:
  1. Get in to target (mail attachments, password hacking, buffer overflow, malformed packets etc.)
  2. Pull rest of needed code using mini server
  3. Select ip adresses randomly for next targets
  4. Scan new targets and infect vulnerable targets
  5. Execute worm main function, e.g. install backdoor, steal data, wait for triggers for DDOS attack etc.
- **Trojans**: A self-contained program downloaded and executed by the user which oftentimes creates a backdoor for another malicious party. Can also be used for **ransomware**.
- **Spyware**: A process used to gain info about user, e.g. a **keylogger**.
- **Rabbit virus/fork bomb**: A process which keeps replicating itself to deplete resources, a denial of service attack.
- **Honeypot** (briefly mentioned): A process which fakes to be a server/system on a specific port.
- **Bots** (briefly mentioned): In a malware sense, a process which infects a computer to be part of a botnet used for nefarious purposes.
- **Email spam messages** (briefly mentioned): Besides commercial purposes, can be used to distribute malware.

8.8 Common attack scenarios

F. van Verseveld

These are by no means exhaustive, but the most common ones.

8.8.1 Man in the middle

Fool the network to get a **Man in the middle** (MiTM).

8.8.2 Setuid

A common pitfall is to forget to lower/release privileges for forked children or after a critical operation.

8.8.3 ARP spoofing

An attacker manipulates network going from one gateway to another by responding to an ARP request. This is an MiTM attack.

8.9 Stack smashing

Manipulating the stack by writing beyond a buffer's size is called stack smashing.

8.9.1 NX

Disable execution where stack pages reside.

8.9.2 Address space layout randomization

A very good software mitigation on 64 bit operating systems is **address space layout randomization** (ASLR). For example, even a monolithic kernel is usually at most a few megabytes. Since the address space on x64 systems with PAE is $2^4$8, changes that an attacker guesses the kernel's location are very slim.

8.9.3 Stack canary

Embed checksum around stack and check integrity when leaving a function to prevent stack smashing.

8.9.4 Return Oriented Programming

A specialization of buffer overflow vulnerabilities is **return oriented programming** where the return address is manipulated. For example, suppose a program verifies user credentials and only when properly authenticated, calls 'secret'. An attacker can just overflow the buffer and manipulate the return address to 'secret', which defeats the whole authentication process. Libc also prevents this using ASCII armoring.

9 Miscellaneous Knowledge

M.Diallo

The following information can not be appropriately categorized. Yet it is imperative that one has knowledge of it before the exam. As such this section will go into detail on that.

9.1 Deadlocks

A deadlock is a state in which each member of a group is waiting for another member to take action, yet no progress can be made. There are four mandatory conditions that must be fulfilled simultaneously to exhibit a deadlock:

- Mutual Exclusion - At least one resource must be held in a non-sharable mode. Otherwise no process would be prevented from using the resource
- Hold and wait/Resource holding - A process is holding at least one resource and is requesting an additional resource held by another process.
- No Preemption - A process may only be released voluntarily be the process holding it
- Circular wait - Each process must be waiting for a resource held by another process. Which in turn is waiting for the first process to release the process.

9.2 Atomic Instructions

9.3 Spin Locks And Busy Waiting

Remember that a process can be in the running state. What happens when a resource it wants access to is locked? Assuming protection measures are in place it will continue waiting until said resource is available. **Busy waiting** occurs when the process is waiting while consuming CPU time. This prevents other processes, that are not in a waiting stage from using CPU time. In modern operating systems, system calls are available to prevent this from happening.
A **spinlock** is a lock which causes a thread trying to acquire it to remain in a loop(hence the word spin) while checking if the lock is available. Since the thread is still active and using CPU-time, this is a form of **busy waiting.**]

10 TL;DR Exam Topics

M.Diallo

This is a short review of the exam topics to fresh up on knowledge.

10.1 Processes: lifecycle of processes (different process states)

A process has the states: new, ready, running, blocked and finished

10.2 context switches, PCB, fork, exec, communication between processes

A **context switch** is what happens when one changes from one process to another. During that, data of the old process is stored so that it can be resumed later. This is a costly operation.

The **PCB or process control block** is the data structure in the operatingsystem kernel required for scheduling a particular process. It contains the process identification, state and control data.

Fork is when one makes a copy of an existing process. The newly created child will be a child of the caller.

Exec is when one replaces the data of the current process by the binary file specified as argument.

11 Apendices

F. van Verseveld

12 Appendix A - Terminology

12.1 Processes

State	new, ready, blocked/waiting, running, finished/defunct
Process control block	holds process identifier (pid), CPU registers state, allocated resources, process relation, statistics and PCB list (for traversing)

12.1.1 Process creation and execution

Unix uses `fork(2)` and `exec(2)`. Windows uses `spawn`. Fork duplicates the address space maintaining a parent/child relation. It is expensive if an MMU and virtual memory are unavailable. **Copy on write** copies pages on demand improving performance. Once a child is defunct, the parent should reap/wait the **zombie**. Children that lose their parent are **orphans**.

12.2 Process communication

Interprocess	signals, message queues, semaphores, . . .
System local	program arguments, shared memory, file descriptors (pipes, configuration files)
External	network, . . .

12.2.1 File descriptors (fd)

Allocated resources on Unix are represented by a integer, the file descriptor. One defining feature of Unix: 'Everything is a file on Unix': Sockets, directories, anonymous pipes, . . .

12.2.2 Signals

Immutable signals: `kill`, `stop`. All signals' effects:

Termination	hup, int, pipe, alrm, term, usr1, usr2
Core dump	quit, ill, abrt, fpe, segv, bus, sys, trap, xcpu
Stop	stop, tstp, ttin, ttou
Ignored	chld, urg, winch
Continue	cont

Two standard interfaces for configuring signals: `signal` (libc), `sigaction` (POSIX)

12.2.3 Pipes

Unidirectional channel that redirects a file descriptor (e.g. stdin, stdout). On Windows, the OS cheats by creating a temporary file. E.g.: on Unix, one could do: `yes | ./install` where the first process prints ‘y’ indefintely to make sure the install script is automated. Once ‘./install’ terminates, ‘yes’ will terminate because the pipe’s writing end is closed. On windows, such a command (if these would exist), never stops.

12.3 Scheduling

Scheduling *distributes limited resources*, e.g.: CPU time, I/O and memory scheduling. Some algorithms:

Uni/Multiprocessor First Come First Served (FCFS), Shortest-Job-First (SJF), Priority Scheduling, Round Robin (RR), Multilevel Feedback Queue, Completely Fair Scheduling (CFS)

Realtime Priority based (PB), Earliest deadline first (EDF), Rate monotonic (RM)

12.3.1 General procedure

When a scheduler issues a new task, the **context switches**. The response time on interactive systems is the system respond time, while on realtime systems, this is the interval of starting and finishing a task. The absolute time it takes for a process to become ready is the release time. Finally, the turnaround time is the total amount of time a request takes to fulfill.

12.3.2 Objectives

There is no best scheduling algorithm, because can’t time travel in the future, only predict. A good scheduler ensures fairness, non-starvation, prioritization, is responsive and minimizes turnaround and throughput.

12.4 Synchronization

Parallelism is important on multi-core processors because this is the only way to greatly improve performance. Care must be taken to prevent **race conditions**. E.g.: producer/consumer model. Solutions are:

Mutual exclusion locking exclusive access during the **critical section**.

Progress minimize delay if it is not being used.

Bounded waiting limits waiting time for other processes.

13 Appendix B - Formulae

From 4.1.6 Schedulability Utilization

$$\sum_{i=1}^{n} \frac{C_1}{P_i} \leq n * (2^{1/n} - 1)$$

From 6.3 Effective access time (EAT)

$$\text{effective access time} = (1 - p) * \text{memory access} + p * \text{page fault time}$$

From 7.2.1 RAID block stripping

$$D = (i\ mod\ n) + 1$$

14 Appendix C - Miscellaneous

14.1 Common C security issues

- Buffer overflow (e.g.: `char buf[8]; strcpy(buf, "oops, too long");`)
- Attacker controlled string (e.g.: `printf(user_input);`)
- Attacker controlled data (e.g.: `int index = ask_index(); data[index] = 0xDEADBEEF;`)

14.2 General security issues

Citing relevant points from top 10 security issues:

Remote:	Injection, cross side scripting
Local and remote:	Broken (e.g. default) authentication/configuration
Carelessness:	access control (e.g. disabling SE linux), using known vulnerable components (e.g. heartbleed)

14.3 Security mitigation

Hardware issues (e.g.: meltdown, spectre) cannot be fixed in software, only mitigated (i.e. risk reduced). For meltdown and spectre, less sensitive information is exposed and the kernel better tries to randomize their location.

14.4 Manpages

`man` is a manual pager providing reference information. Example for libc's read function: `man 2 read`. The section number (i.e. 2 in the example) is optional, but required if ambiguous (i.e. available in multiple sections). All sections:

1. Executable programs	ls, bash, printf
2. System calls	stat, open, close, read, write
3. Library calls	fgets, printf, strncpy
4. Special files	/dev/shm, /dev/tty0, /dev/urandom
5. File formats	man 5 elf, man 5 fstab
6. Games	supertux
7. Miscellaneous	man, groff

Discussed topics in reference are usually: synopsis, description, return value (for functions), attributes, confirming standards, notes, bugs, code example and related topics. For libc's printf however, we must use `man 2 printf` because `man printf` is interpreted as `man 1 printf`.