Operating Systems

Summary

Fabian Damken

November 8, 2023

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Contents

1 Introduction

1.1 History

1.2 Definition over Hardware Abstraction

An operating system (OS) is a program or a set of programs providing:

- An execution environment for user applications (machine abstraction) and
- an interface between hardware resources and users with *resource arbitration/management*.
- The definition splits up where it goes to define what an OS contains:
  1. Kernel and the system libraries
     - Problem: What actually is a “system library”?
     - Solution: Standards (POSIX, LSB, WinAPI, . . .).
  2. Only the kernel, i.e. everything that runs in kernel mode
     - Problem 1: Need of a definition what “kernel mode” means.
     - Problem 2: Does not apply for hardware architectures that do not follow this definition.
     - Solution: Restrict to certain hardware architectures.

1.2.1 Portable Operating System Interface (POSIX)

- Specification of the OS interface (e.g. functions to open files), . . .
- Defined by IEEE 1003.x and ISO/IEC 9945.
- First published 1988 and maintained by Austin Group.
- Current edition (2018) available online: http://pubs.opengroup.org/onlinepubs/9699919799/
- API standard: Write once, run everywhere.
- POSIX compliance certification is available through IEEE and The Open Group.

1.2.2 Linux Standard Base (LSB)

- Specification of the interface of Linux.
- Maintained by the Linux Foundation.
- Large overlap with POSIX.

- Defined by ISO/IEC 23360 since version 3.1 (2006).
- First published 2001.
- Latest release available online: http://refspecs.linuxfoundation.org/lsb.shtml with lots of additional resources: https://www.linuxbase.org/download/
- API standard: Compile once, run everywhere.
- LSB compliance certification is available through the Linux Foundation.

1.2.3 x86 Rings

- Intel x86 defines multiple protection "rings".

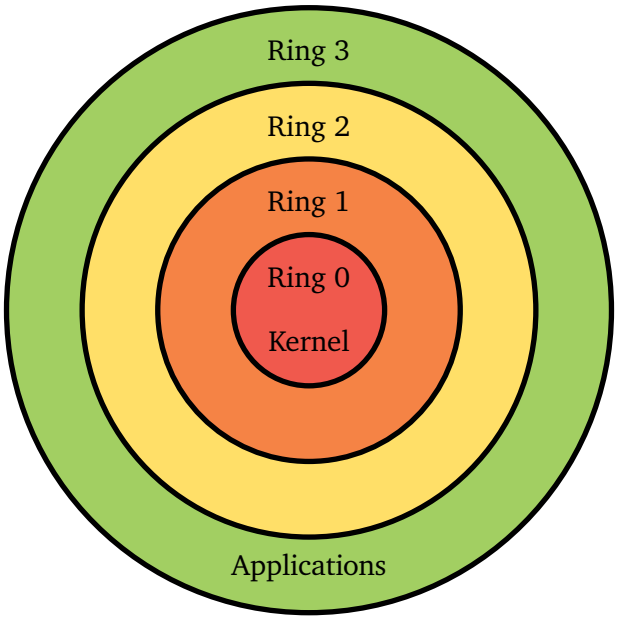

Figure 1.1: Intel x86 Protection Rings

- Some privileged instructions can only be executed in certain rings.
  Example: To modify the memory, a ring 0 instruction is needed.
- To prevent any program from breaking the OS, only the OS can execute in ring 0 ("kernel mode"), other programs can only execute in ring 3 ("user mode").
- Problematic: Where to set the boundary for what to execute in ring 0 and what not?

1.2.4 Monolithic vs. Micro-Kernel

Monolithic Kernel All parts of the kernel (task management, memory management, network stack, file systems, graphics driver, keyboard driver, ...) run in ring 0. This is the most used kernel type, however in Linux, the user can decide which modules should run in ring 0 and which in ring 3.

Micro-Kernel All parts of the kernel run in 3 excluding HAL and IPC. This is way more secure but has some performance drawbacks as the kernel has to do context switches all the time.

1.3 Definition over Coordination

An operating system (OS) is a *resource allocator* and a *control program* that

- manages all resources (hardware, applications, etc.),
- decides between conflicting requests for efficient and fair resource use and
- controls the execution of programs to prevent errors and wrong usage of resources (ordering, sequencing, . . .).

2 Processes and Inter-Process-Communication

2.1 Processes

A *process* describes a program in "execution".

- Program $\rightarrow$ *passive entity* (specification)
- Process $\rightarrow$ *active entity* (execution of the specification)

The fundamental abstraction that the OS provides is the illusion of own CPU(s) per process (processes see the CPU(s) as if they are alone on the system and the OS schedules the processes on the CPU).

2.1.1 Process Family and Hierarchy

Family Relations

- The *parent* process creates *children* processes which can then create other children processes forming a tree of processes.
- Options for resource sharing:
  1. Parent and child processes share all resources.
  2. Children share a subset of the parent's resources.
  3. Parent and child share not resources at all.
- Options for concurrent/sequential execution:
  1. Parent and children execute concurrently.
  2. Parent waits until all or some children terminate.
- Options for the programs to execute:
  1. Children are duplicates of the parent.
  2. Children have different programs than the parent.
- Options in POSIX:
  1. Fork system call creates new (clone) process.
  2. Execution of a system call after the fork system call to replace the memory space of the process with a new program.

Process Hierarchies

- Parent creates a child process and child processes *can* become a standalone process with a different program, different state but possibly sharing memory or files.
- Child processes with a terminated parent are called *zombie processes*.
- The parent/child relations result in a hierarchy.
  - **UNIX** calls this a “process group”.
    - The parent/child relation cannot be dropped.
    - Parent/child maintain a distinct address space and the child initially inherits/shares the contents of the parents address space *contents*.
  - **Windows** has different concepts of process hierarchy.
    - Processes can be creates without implicit inheritance relations, though the parent can control a child using a “handle”.
    - Child processes start with clean memory.

Example: Process Tree

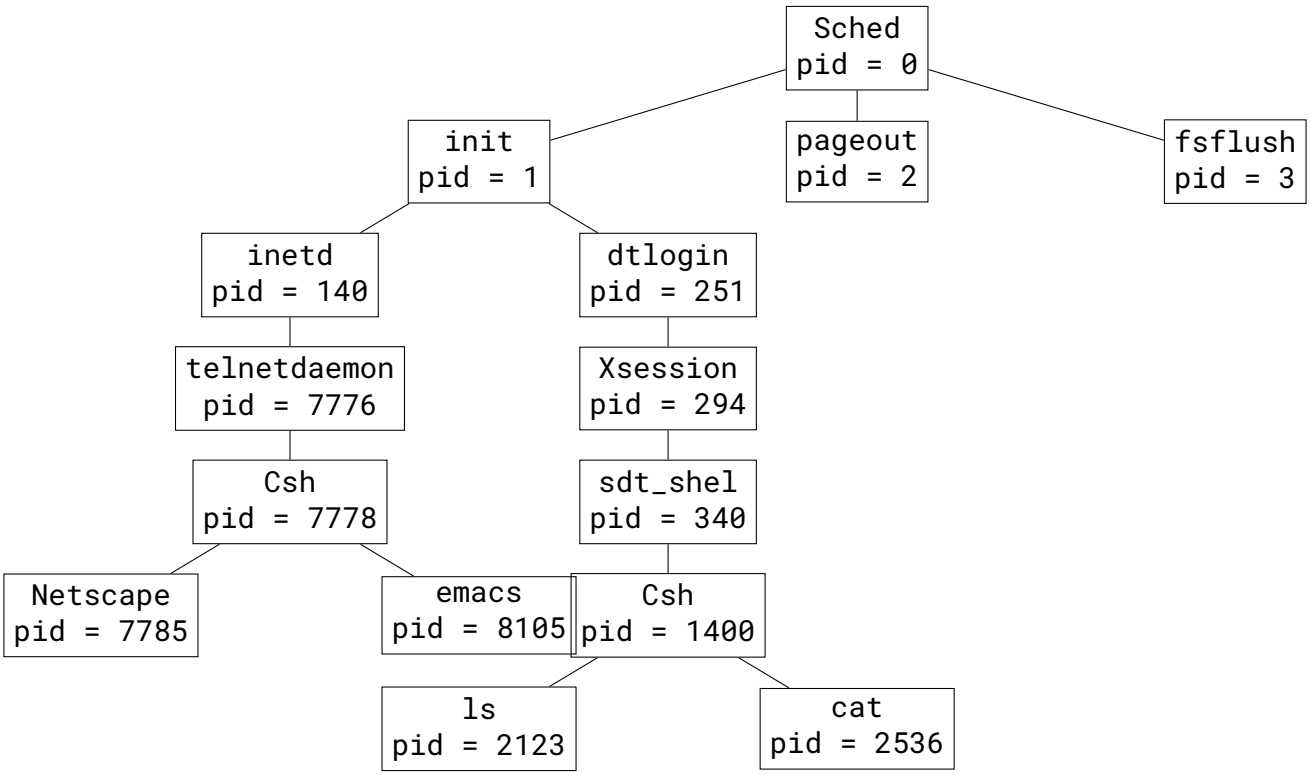

Figure 2.1: Process Tree: Solaris

2.2 Concurrency

- If more processes than CPUs exist, the processes must be executed *concurrently*. There are multiple mechanisms for concurrent execution:
  1. One after another
  2. Interleaved execution
- All these mechanisms need that
  - switching between processes is possible and that
  - switching does not operations of a process.

2.3 Process Management

Process management includes the following steps:

1. Running, suspending and resuming processes
2. Process creation
3. Process termination
4. Provide inter-process communication (IPC) for cooperating processes

This needs scheduling, synchronization and deadlock handling for all processes.

During the execution of a process, the process runs through multiple “process states”:

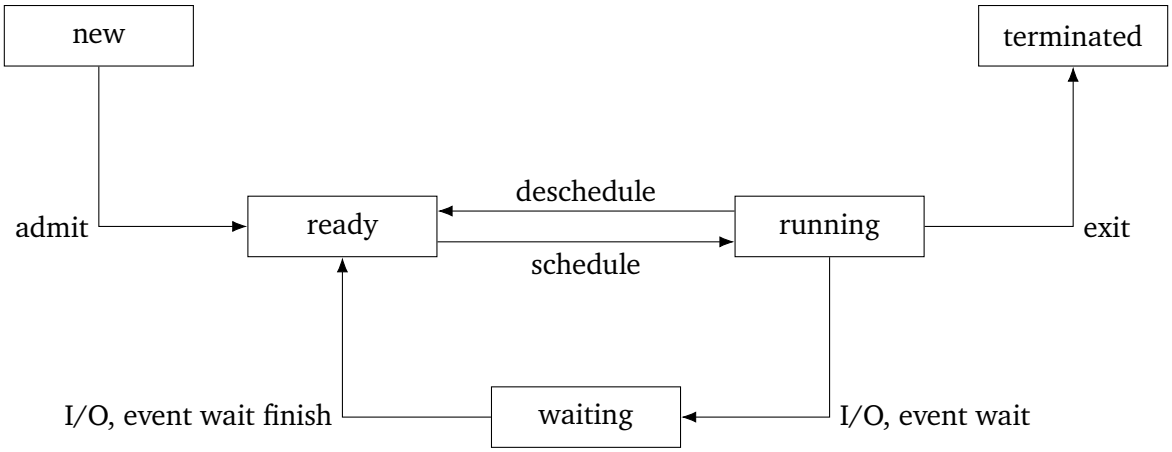

Figure 2.2: Process States

new Process initialized

ready Waiting for CPU assignment

running Instructions are executed

waiting Waiting for I/O or events

terminated Finished execution

2.3.1 Process Control Block

- Essential for switching between processes.
- Saves the context for a processes to survive the switch. The context contains:
  - Which instruction was executing (PC)
  - What is was doing (register contents)
  - Which resources where used (memory, I/O)
- The context switch is “overhead” on both OS and HW as the system does not do useful work while switching.

The PCB is used like this:

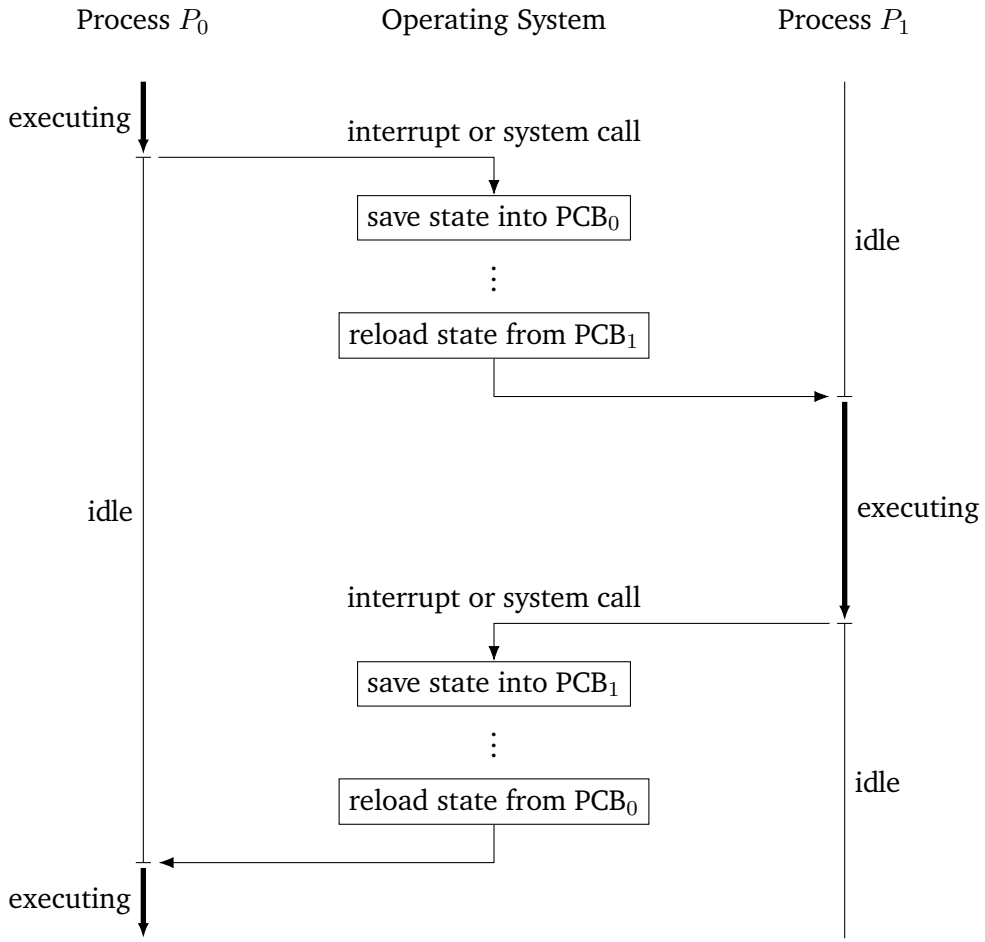

Figure 2.3: Process Control Block: Usage

2.3.2 Process Creation (POSIX)

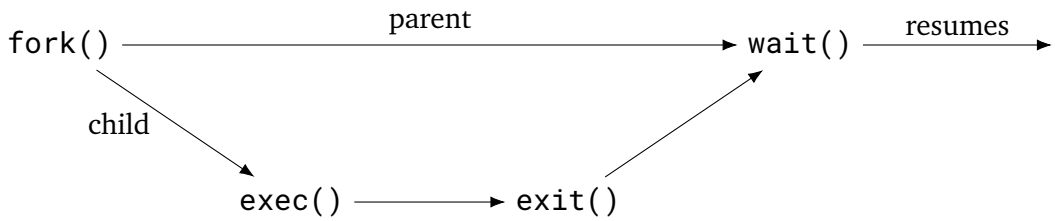

Figure 2.4: POSIX Process Creation

- The `fork` system call create a new (clone) process.
- The `exec` method is called after `fork` to replace the memory space with a new program.
- The parent `wait`s until the child finished the execution.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    pid_t pid;

    pid = fork();
    if (pid < 0) {
        fprintf(stderr, "Fork Failed\n");
        exit(-1);
    }
    if (pid == 0) {
        execlp("/bin/ls", "ls", "-l", "/home", NULL);
    } else {
        wait(NULL);
        printf("Child Complete\n");
        exit(0);
    }
}
```

Figure 2.5: Forking a Process in C (POSIX)

Example: Forking in C

2.3.3 Process Termination

- A process executes the last statement and asks the OS to delete it (`exit()`).
  - Returns a status value to the parent (via `wait()`).
  - The resources of the process are deallocated by the OS.
- The parent may terminate the execution of children processes (`kill()`, `TerminateProcess()`) if:
  - Child has exceeded allocated resources,

- task assigned to the child is no longer required or
- parent is exiting (some OS do not allow children to continue if the parent terminates (zombie control) → cascading termination).

2.3.4 Inter-Process-Communication

- Processes may be
  - **Independent**
    - A process is independent if it cannot affect or be affected by other processes executing in the system.
    - Independent processes do not share data with any other process.
  - **Cooperating**
    - A process is cooperating if it can affect or be affected by other processes executing in the system.
    - Cooperating processes share data with any other process.
- Cooperating processes are needed for information sharing, speedup, modularity, privilege separation, etc.
- But this brings up a few issues:
  - Data sharing
  - Processes need a communication channel or medium
  - Coordinate and synchronization (race conditions, deadlocks; critical sections, locking, semaphores, see 4, 5 and 6).

2.4 Inter-Process-Communication Models

There are two basic IPC-models:

1. *Shared Memory*
   Processes use shared memory region as communication medium.
2. *Message Passing*
   Processes use (abstract) communication link with primitive operations for sending/receiving messages.

2.4.1 Shared Memory

- Cooperating processes:
  - *Producer* process produced data items.
  - *Consumer* process consumes data items.
  - Both share a memory region named *buffer*.
- The buffer can either be bounded with a fixed buffer size or unbounded with no practical limit.

```c
// Buffer boundary (ten items).
#define BUFFER_SIZE = 10

typedef struct { ... } item;

// Shared circular buffer.
item buffer[BUFFER_SIZE];
// Next free position, shared.
int in = 0;
// First full position, shared.
int out = 0;
```

Figure 2.6: Bounded Buffer in C

Example: Bounded Buffer in C

- The buffer is empty if and only if `in == out` holds.
- The buffer is full if and only if `(in + 1)% BUFFER_SIZE == out` holds.

```c
// Producer: Waits if buffer is full and inserts new items.
while (1) {
    // Do nothing, buffer is full.
    while ((in + 1) % BUFFER_SIZE == out) { }

    // Produce a new item and place it in the buffer.
    buffer[in] = new_item;
    in = (in + 1) % BUFFER_SIZE;
}
```

Figure 2.7: Bounded Buffer in C: Producer

```c
// Consumer: Waits if buffer is empty and removes items.
while (1) {
    // Do nothing, buffer is empty.
    while (in == out) { }

    // Remove an item from the buffer and process it.
    current_item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    process(current_item);
}
```

Figure 2.8: Bounded Buffer in C: Consumer

2.4.2 Message Passing

- Processes use two primitive operations for sending/receiving messages (with either a fixed or variable size).
  - `send(message)`
  - `receive(message)`
- If two processes (say P and Q) want to communicate, they need to establish a *communication link* and exchange messages via send/receive.

- The implementation and design of the communication link may either be physical (hardware bus, network, ...) or logical (software abstractions, ...).
- The implementation leads to the following issues that have to be taken care of:
  - How to establish the link?
  - Can the link be associated with more than two processes?
  - How many links exist between every pair of processes?
  - What is the capacity of a link?
  - Is the size of a message fixed or variable?
  - Is a link unidirectional or bidirectional?

  → Addressing, naming (direct vs. indirect)

  → Synchronization aspects (blocking vs. nonblocking)

  → Buffering (link capacity)

Addressing

- The communicating processes need to name each other explicitly:
  - P: `send(Q, msg)` (Process P sends a message to process Q)
  - Q: `receive(P, msg)` (Process Q receives a message from process P)
- The addressing can take place symmetric or asymmetric, where the latter means that one process "listens" for other processes and then stores the corresponding PID (Process ID) for sending messages back.

  symmetric `send(Q, msg)`; `receive(P, msg)`

  asymmetric `send(Q, msg)`; `receive(&pid, msg)`
- Using indirect communication, messages are directed to and received from *mailboxes* or *ports*.
  - The communicating processes to not have to know the name of each other.
  - Each mailbox has its unique ID and can be used for send/receive.
  - Processes can only communicate if they share a mailbox.
  - Example: UNIX pipes like `cat xzy | lpr` (implicit FIFO mailbox).

Synchronization

- Message passing may either be *blocking* or *nonblocking*.
- *Blocking* (B) is considered synchronous:
  - `B.Send` The sender blocks until the message was received.
  - `B.Receive` The receiver blocks until a message is available.
  - `Rendezvous` Both sender and receive are blocking.
- *Nonblocking* (NB) is considered asynchronous:
  - `NB.Send` The sender sends the message and continues.
  - `NB.Receive` The receive receives a valid message or nothing (null).

Message Buffering (Queues)

The messages reside in temporary queues that can be implemented in one of three possible ways:

- *Zero capacity* - no messages can be stored
  The sender must wait for the receiver for each message (rendezvous).
- *Bounded capacity* - finite length of n messages can be stored
  The sender must only wait if the queue is full.
- *Unbounded capacity* - infinite length
  The sender never waits.

Pipes

- *Pipes* are conduits that allow two processes to communicate.
- Implementation issues:
  - Provide uni- or bidirectional communication?
  - What is the relationship between communicating processes (parent/child)?
- An ordinary (anonymous) pipe is:
  - Unidirectional,
  - creates using the `pipe()` system call,
  - is not accessible outside of the creating process and
  - creates two files descriptors:
    * `fd[0]`: read-end
    * `fd[1]`: write-end

Example: (A)LPC in Windows

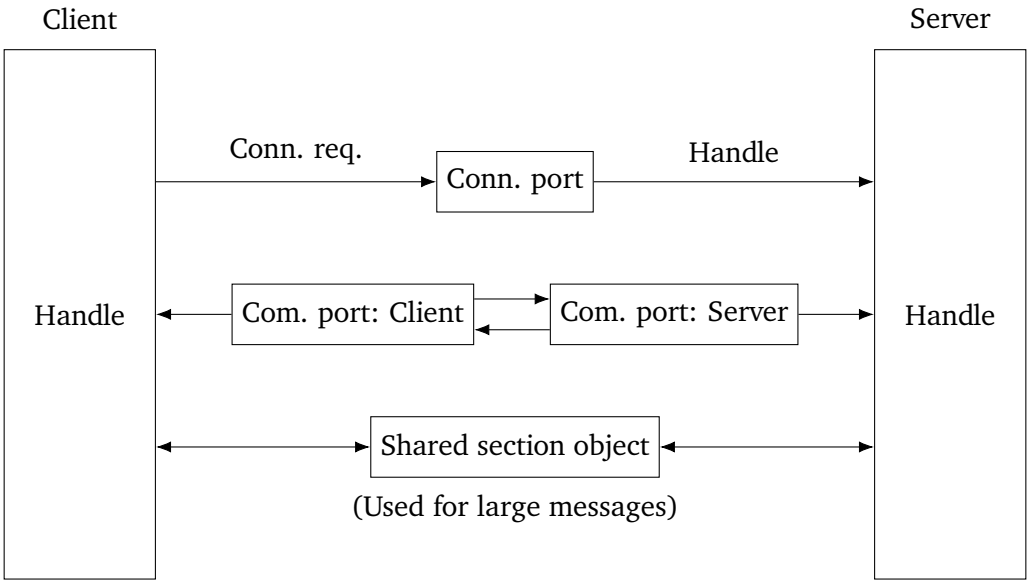

Figure 2.9: (A)LPC in Windows

1. The client opens a handle to the connection port object of the subsystem.
2. The client sends a communication request.
3. The server creates two private communication ports and returns the handle to the client.
4. The client and the server use the corresponding port handle to send messages (and callback to listen for replies).

3 Threads

- Programs often perform multiple tasks concurrently (writing data to disk, perform spell checking, ...).
- Problematic: Using separate processes with IPC is cumbersome and has lots of performance drawbacks.
- Solution: Allow processes to have multiple *threads* executing concurrently for different tasks.
- Benefits of multithreading:
  1. **Responsiveness**
     Allows a program to continue even if other parts of the program are blocked.
  2. **Resource Sharing**
     Threads share process resources which is easier than explicit IPC.
  3. **Efficiency/Performance**
     Creating and context switching of threads produced a lot less work than for processes.
  4. **Scalability**
     Processes can take advantage of multiprocessor architectures and worker threads can be dispatched to different processors.

3.1 Process vs. Thread Model

- **Process Model** (heavyweight single thread)
  - Each process has a discrete and distinct control flow and
  - unique PC, SP, registers and address space.
  - The processes interact via IPC.
- **Thread Model** (lightweight sub-processes)
  - Each thread runs independently and sequentially and has its own PC and SP like a process.
  - Each thread can create sub-threads, can request services and has a “state” (ready, running, blocked) like a process.
  - Difference to processes: All threads of a process share the same address space as the parent process:
    * access to global variables within the process
    * access to all files within the shared address space
    * reading, writing, deleting variables, files, stacks, ...
  - Therefore, threads are much faster and provide concurrency, but there is no isolation between them.
  - The speedup is best when all threads use different resources (e.g. first one uses the CPU, the second the I/O (keyboard, display, ...) and the third uses the storage).

3.2 Example: Multi-Threaded Web Server

- The request handling is decoupled from the request execution.
- The so-called *dispatcher thread* creates a new *worker thread* for each incoming request that then handles the request execution.
- With this method the web server is sped up as the requests can be handled concurrently.
- This multi-process/child model would also work with processes, however that produces a lot of overload due to process creation, context switching, scheduling, etc.

3.3 Theoretical Speedup (Amdahl's Law)

Amdahl's Law describes the *theoretical speedup* from adding CPUs to programs with parallelized parts.

Let S be the serial portion (percentage runtime) and N the parallel processing cores. Then the formula is:

$$\textit{speedup} \leq \frac{1}{S + \frac{1-S}{N}} \qquad \lim_{N \to \infty} \textit{speedup} = \frac{1}{S}$$

3.4 Implementations

- **User-Level**
  - \+ Each process can define its own thread policies.
  - \+ Flexible localized scheduling
  - – No kernel knowledge and therefore no kernel support for thread management.
- **Kernel-Level**
  - \+ Single thread table under kernel control.
  - \+ Full kernel overview and thread management.
- **Hybrid**
  - – Multiplexing of user-level threads onto kernel-level threads.
  - – Each kernel thread owns a limited sphere of user-level threads.

3.4.1 Many-to-One

- Many user-level threads map onto a single kernel thread.
- One thread blocking is causing all other threads to also block.
- Multiple threads may not run in parallel because only one is in the kernel at a time.
- Examples: Solaris Green Threads, GNU Portable Threads
- Pros/Cons

+ Flexible as thread management is in user space.
– The process blocks if a thread blocks.
– Only one thread can access the kernel at a time → no multiprocess support.

3.4.2 One-to-One

- Each user-level thread maps onto a kernel thread.
- Creating user-level threads results in creating a kernel thread.
- More concurrency than many-to-one.
- The number of threads per process is sometimes restricted due to overhead.
- Examples: Windows, Linux; Solar 9+

3.4.3 Many-to-Many

- Many user.level threads map onto many kernel threads.
- Allows the OS to create a sufficient number of kernel threads.
- Examples: Solaris prior version 9, Windows Fibers
- Pros/Cons
  + Large number of threads.
  + If the user-thread blocks, the kernel schedules another for execution.
  – Less concurrency than one-to-one mapping.

3.4.4 Two-level Model

- Similar to many-to-many, except that it allows that a user-level thread is bound to a kernel thread.
- Examples: IRIX, HP-UX, Tru64 UNIX, Solaris prior to version 8

3.5 Linux Threads/Tasks

- Linux calls threads *tasks*.
- Linux does not distinguish between processes and threads.
- Thread creation is done using `clone()` system call with flags indicating the sharing method.
- `clone()` allows a child task to share the address space/file descriptors/signals of the parent task with the following flags:

Flag	Meaning
`CLONE_FS`	Share file-system information.
`CLONE_VM`	Share memory address space.
`CLONE_SIGHAND`	Share signal handlers.
`CLONE_FILES`	Share set of open files

Table 3.1: Linux: Possible `clone()` flags

- `struct task_struct` points to the data structures of the process.
- Both functions `pthread_create()` and `fork()` are implemented using `clone()`.

3.6 POSIX Threads (C)

```c
int result = 0;

void* thread_func(void* arg) {
    for (int i = 0; i < MAX; ++i) {
        result += 42;
    }
    return NULL;
}

int main() {
    // Stores the thread ID (TID) for later communication.
    pthread_t tid;
    pthread_create(&tid, NULL, thread_func, NULL);
    ...
    // Wait until the thread is finished.
    pthread_join(tid, NULL);
    printf("Result: %d\n", result);
    return 0;
}
```

Figure 3.1: POSIX Threads in C

3.7 Single to Multi-Threaded

- If multiple threads access the same global variable at once, the changes of one of the threads may be lost.
- This problem can be solved by allowing threads to have a private copy of the global variables (thread-local storage, TLS).
- The TLS therefore keeps a private copy of the global variables.

4 Deadlocks and Resource Management

4.1 Resource Management

- The number of resources in a system is finite, but there are lots or processes created by the OS.
- The OS has to manage the resources addressing:
  - Resource constraints
  - Ordering of processes, IPC
  - Precedence relations
  - Access control for global parameters
  - Shared memory
  - Scheduling
  - and so on...

  This solutions must be *fair*, *efficient*, *deadlock-free* and *race-free*.

4.1.1 Google Chubby Lock Service

Google Chubby is a distributed lock service developed by Google to synchronize Client/Server interactions and accesses to shared resources. It is used by a lot of the infrastructure of Google (GFS, BigTable, MapReduce, ...).

Problems:

- One node is holding a consistency lock that others need to access to proceed. Therefore, if one node holding lock hangs, the client/server hangs.
- Multiple locks get set; old lock is not released leads to inconsistent masters.
- If server partitions happen, the resource access is inconsistent and R/W data consistency is lost.

4.1.2 Resource Sharing and Inter-Process-Interaction

The ordering of processes/threads for handling resource contentions include the following points to consider:

- What to do if a process starts writing to a section that another process is currently reading?
- What to do if a new process gets allocated the same resource already allocated to an existing process?
- What to do if a slow/stalled process blocks other processes from executing?
- ...

All in all this can be concluded as: How to have a shared resource (*critical section*, *CS*) be allocated by only one process at a time (*mutually exclusive access*, *ME*) such that the outcome is of a correctly ordered execution?

Mutual Exclusion

See 6.

4.2 Deadlock/Livelock and Starvation

A set of processes is deadlocked if every process waits for a resources held by another process.

Deadlock None of the processes can be run, releases or be awakened.

Livelock All processes (can be) run, but no progress is made (e.g. because of missing data of another process).

4.3 Deadlock Conditions

For a deadlock to be possible, all of the following conditions have to be fulfilled:

Mutual Exclusion Each resource has to be accessed mutually exclusive.

Hold and Wait A process holding at least one resource has requested and is waiting to acquire additional resourced currently held by another process.

No Preemption A resource cannot be acquired forcibly from a process holding it. It can only be release if the holding process releases it voluntarily after the process has completed its task.

Circular Wait A circular chain of more than one process exists. Let $\{P_0, P_1, \cdots, P_n, P_0\}$ be a chain of waiting processes where each process P_i waits for a resource held by $P_{(i+1) \bmod n}$ (forming a dependency cycle).

If either of these conditions is not given, no deadlock can occur.

4.4 Deadlock Modeling and Visualization

Deadlocks (more specifically resource dependencies) can be visualized as graphs (where circles are processes and rectangles resources):

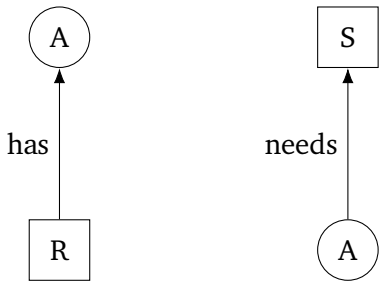

Figure 4.1: Deadlock Modeling: Graphs

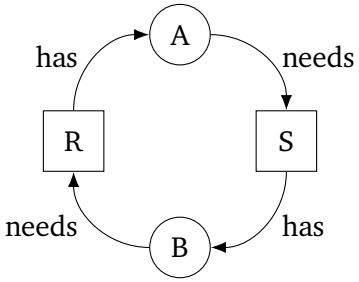

Figure 4.2: Example: Deadlock Modeling: Graph

Using graphs, it is extremely simple to spot cycles. These cycles can be removed by reordering the requests, but:

- Requests are often dynamic/streaming,
- The OS has no global overview and
- Tasks have precedence relations, priorities, timing duration, etc.

So this is a very hard task. This task is implemented by *ordering and scheduling algorithms*.

4.5 Deadlock Strategies

There are three general strategies for dealing with deadlock:

- **Ignore** the problem (it might just go away)
  - Implemented by Windows, UNIX, ...
  - It works, iff:
    - The cost (in terms of workload) for avoidance/prevention is high.
    - Deadlocks occur rarely enough or may possibly go away with random/exponential back-offs (timeouts) on service responses.
  - Using this method, there is a trade-off between convenience (cost, complexity, ...) and correctness (repeatability, predictability, ...).
- **Detect** and **Recover** (check, detect, recover)
- **Avoid** and **Prevent** (by design, negate one of the four conditions for deadlocks)

4.5.1 Detection

- Every OS has a “resource allocation checker” that performs static tests for deadlocks.
- Dynamic checking may vary from rigid/static processes to heuristic/adaptive tests.

One Resource per Type

- Develop resource ownership and requests graph.
- If a cycle is found, this indicates a deadlock.

→ Set up a DS (depth search, e.g. a spanniong) where a cycle is visible via node repetition in the DS. A spanning tree is a subset of the edges of a graph connecting all edges without any cycles.

Multiple Resources per Type

- Spanning tree computation is hard, so for multiple types, a simple matrix data structure is more convenient.

Let $E = (E_1, \cdots, E_n)^T$ be the vector of all *existing* resources of the resource types $R_1, \cdots, R_n$. Let $A = (A_1, \cdots, A_n)^T$ be the vector of *remaining* resources of the resource types. Notice that in any state, the formula $A_i \leq R_i$ holds true for every i. Using this lets define matrices $Q \in \mathbb{N}_0^{n \times m}$ and $T \in \mathbb{N}_0^{n \times m}$ where Q represents, which process (of a set of processes $\{P_1, \cdots, P_m\}$) holds how many resources of each type (the entry Q_{ij} indicates how many resources of type R_j the process P_i holds). Furthermore, the matrix T defines (in a similar way) which process requested how many resources of each time. Notice that in any state, the formula $\sum_{i=1}^{n} C_{ij} + A_j = E_j$ holds true for every j. A matrix $R \coloneqq T - Q$ contains the numbers how many resources are still needed (i.e. not yet provided).

In summary, the vectors and matrices look like this (the rows of the matrices show how many resources the process currently holds/needs):

$$E \coloneqq \begin{bmatrix} E_1 \\ \vdots \\ E_n \end{bmatrix} \qquad A \coloneqq \begin{bmatrix} A_1 \\ \vdots \\ A_n \end{bmatrix}$$

$$Q \coloneqq \begin{bmatrix} Q_{11} & Q_{12} & Q_{13} & \cdots & Q_{1m} \\ Q_{21} & Q_{22} & Q_{23} & \cdots & Q_{2m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ Q_{n1} & Q_{n2} & Q_{n3} & \cdots & Q_{nm} \end{bmatrix} \qquad T \coloneqq \begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

$$R \coloneqq T - Q$$

Progress via Process Selection Definition: Let $a, b \in \mathbb{N}^n$, than $a \leq b$ hold iff $a_i \leq b$ for all i.

If at least one row r exists in R with $r \leq A$, enough resources are available and the process is able to continue. If not, a deadlock *may* occur.

4.5.2 Recovery

- **Recovery through preemption**
  - Take a resource from an executing process (depending on the nature of the resource and if the process is interruptible).
- **Recovery through rollback or back-offs**
  - Periodically create checkpoints.

- Use this saved state and restart the process if a deadlock was found.

- **Recovery through killing processes** (Reactive or Proactive)
  - Crudest but simplest method to break the deadlock: Kill one on the processes in the deadlock cycle and release its resources.
  - Choose a process that can be rerun (e.g. from beginning or a checkpoint) and does not hold up other active processes with precedence dependencies.

4.5.3 Avoidance

- Assumes a priori resource information.
- The simplest and most useful model requires that each process a priori declares the maximum number of resources of each type that it may need.
- The deadlock avoidance algorithm then dynamically examines the resource allocation state to ensure that it is not possible to produce a waiting cycle (safe vs. unsafe scenarios).
- The resource allocation state is defined by the number of available and allocated resources and the maximum demand of the process.

Execution Trajectories

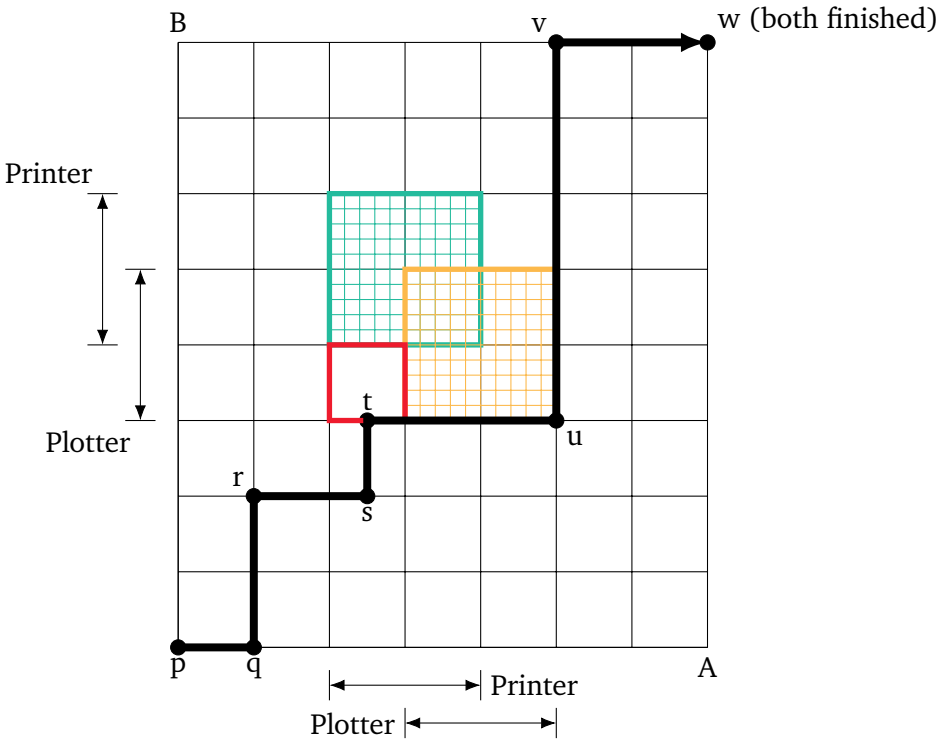

Figure 4.3: Example: Execution Trajectories

- This diagram shows the execution trajectory through two processes where either one uses both the printer and the plotter.
- The trajectory itself (the black line) is monotonically increasing and represents the execution process in process A (horizontally) and B (vertically).
- It cannot enter either one of the hatched fields, because this would require both processes to access the printer (green) or the plotter (orange), which is denied by the mutual exclusivity (ME). Entering these regions would cause a deadlock.
- The red marked region is an unsafe state as it is only possible for the trajectory to move onto a deadlock.

Safe/Unsafe States and Execution Orders

An execution order is *safe* iff a sequence order exists that satisfies even if all processes request their maximum resources (keeping in mind that only one process can execute at any time).

Banker's Algorithm for a Single Resource

If a safe order is possible from the current state, then grant access. Otherwise, deny the request.

This avoids deadlocks by design by only permitting safe states. In reality, the criteria for credits/loans can be based on:

- the past behavior of the processes, or
- bursty traffic analysis (likelihood of run on the bank), or
- constrained maxima (this method is the most common, but abused by DoS attacks).

Banker's Algorithm for Multiple Resource

This algorithm is based on the matrices defined in 4.5.1 and contains the following steps:

1. Look for at least one row in R that is smaller than or equal to A. If no such row exists, the system is likely to run into a deadlock since no process can finish its task.
2. Assume that the process of the chosen row requests all the resources it needs and finishes. Mark that process as terminated and add all its resources to A.
3. Repeat steps 1 and 2 until either all processes are marked as terminated, in which case the initial state was safe, or until a deadlock occurs, in which case it was not.

4.5.4 Prevention

Deadlock prevention is about preventing one of the conditions that must hold to allow deadlocks (e.g. by design). That is, "attacking" each of the conditions separately.

Attacking the Mutual Exclusion Condition

Change: Allow multiple process to access the same resource (via serializers).

- Example: Spooled/buffered printer access via a printer daemon.
  - Only the (single) printer daemon actually accesses the printer resource.
  - Deadlocks for the printer are eliminated.
- Problems:
  - Not all devices can be spooled.
  - Not all processes can be buffered and the daemon policy may not support streams.
- In general:
  - Minimize shared resource assignment.
  - As few processes as possible actually claim the resource.

Attacking the Hold and Wait Condition

To break: Guarantee that whenever a process requests a resource, it does not hold any other resources or a process that is holding a resource does not request more.

- This requires the process to request and be allocated to all its resources before it begins the execution (no wait during the execution), or allow a process to request resources only when the process has none.
  - Problem: The OS may not know "all" required resources at the start of run (e.g. in case of dynamic requests).
  - This ties up resources that other processes could be using $\rightarrow$ low resource utilization; starvation is possible.
- Variation: A process must give up all resources and then request all immediately needed.

Attacking the No Preemption Condition

Changes:

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently held are released.
- Preempted resources are added to the list of resources for which the process is waiting.
- Processes will be restarted only when it can regain its old resources, as well as the new requested ones.
- It is not always an option to preempt and it depends on the resource and the process type (e.g. when taking away a printer halfway in the job).

Attacking the Circular Wait Condition

Option 1: No chain: Processes are entitled to only hold one resource at a time. All processes that request a resource have to give up all other held resources first.

- Problematic Example: If a process wants to copy larges files from memory to printer, it needs two resources $\rightarrow$ fails.

Option 2: Requests are granted in some numerical or precedence sequence (e.g. next request only to a higher numbered resource).

- Low efficiency and long access times; varied resource response characteristics.
- High complexity for ordering with large resource set or short request/access patterns.
- How to produce consistent numbering across distributed systems?

5 Scheduling

Scheduling is a practical alternative to extended resource management and deadlock detection/prevention:

- Using algorithms to allocate resources
  - Example: Provide the resources for the shortest job first.
  - Works great for multiple short jobs.
  - But may cause indefinite postponement of long jobs even if they are not blocked (starvation is possible).
- Other solutions include:
  - First come, first serve policy
  - Shortest job first, highest priority, deadline first, etc.

All these are called *scheduling policies* that should be efficient, fair, deadlock and race free.

5.1 Scheduling Criteria (Optimizations)

- **CPU utilization** should be maximized (keep the CPU as busy as possible doing useful things).
- **Throughput** should be maximized (count of completed processes per time).
- **Turnaround time** should be minimized (average time to execute a process).
- **Waiting time** should be minimized (amount of time a process is ready and waits in the queue).
- **Response time** should be minimized (amount of time it takes from request submission until first response is produced).

5.2 Issues

- Issue 1: Process/Thread Preemptability
  - Non-Preemptable: An ongoing task cannot be displaced.
  - Preemptable: Ongoing tasks can be switched in/out. This might be useful but also cause overhead.
- Issue 2: No single process/CPU usage model
  - The workload has to be guessed or estimated.
  - Apps “typically” stay CPU or I/O-bound. Hybrid is “typically” for interactive apps.

5.3 Determining Length of next CPU Burst

- The length the task length can only be estimated (which is simple for batch jobs).
- The estimation uses the length of previous CPU bursts (history).

Let t_n be the actual length of the n-th CPU bust and τ_{n+1} the predicted value for the next CPU burst. Then, with a parameter $0 \leq \alpha \leq 1$, the value can be estimated using:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

With $\alpha = 0$ this is $\tau_{n+1} = \tau_n$ (directly follow the prior estimation). With $\alpha = 1$ this is $\tau_{n+1} = t_n$ (directly follow the actual length of the prior run).

Often a parameter $\alpha = \frac{1}{2}$ is used (use information from the prior run and a bit of information of the prior estimation).

5.4 Flavors of Scheduling

There are four basic flavors for scheduling:

- **First Come, First Serve** (**FCFS** or **FIFO**)
- **Shortest Job First** (**SJF**)
- **Round Robin** (**RR**)
- **Priority Based** (**PB**)

OS schedulers implement all of the scheduling policies and the choice of the policy highly depends on the nature of specific applications and the mixture (user-level, kernel-level) of the applications.

Definition: The execution length of a process P is given by $E(P)$ and the arrival time by $A(P)$.

5.4.1 First Come, First Serve (Non-Preemptive) (FCFS)

- The processes are executed in their arrival order.
- The waiting time can be reduced by reordering the processes (if possible, e.g. no precedence given) using SJF or EDF scheduling.

Properties Let $P_1, \cdots, P_n$ be processes in a queue that arrived in that order.

Waiting Time for process P_i

$$W(P_i) \coloneqq \sum_{j=1}^{i} E(P_j) - A(P_i)$$

Average Waiting Time

$$\frac{1}{n}\sum_{i=1}^{n} W(P_i) = \sum_{i=1}^{n}\left(\sum_{j=1}^{i} E(P_j) - A(P_i)\right) = \sum_{i=1}^{n}\sum_{j=1}^{i} E(P_j) - \sum_{i=1}^{n} A(P_i)$$

5.4.2 Shortest-Job-First (SJF; SRFT/A.SJF)

- Each job is associated with the expected length of the CPU burst.
- These lengths are used to determine the process with the shortest execution time which is then executed.
- Options:
  - **Non-Preemptive:** Once a CPU is assigned it cannot be preempted until the burst completed.
  - **Preemptive:** If a new process arrives and the expected burst length of the running process is shorter than the expected burst length of the new job, the running task is preempted and the new one is assigned (*Shortes-Remaining-Time-First*, *SRTF* or *Adaptive SJF*, *A.SJF*).
    - This is the optimal as it produces the minimal average waiting time.
    - But it can still lead to starvation if a new job keeps coming in and is getting considered "shortest" ranking, a "long" job can potentially starve (Convoy Effect).
    - Variation: Earliest Deadline First (EDF)

Properties (Non-Preemptive) Let $P_1, \cdots, P_n$ be processes in a queue that arrived in that order and let $E_1, \cdots, E_n$ and $A_1, \cdots, A_n$ the ordered execution and arrival times (ascending).

Waiting Time for process P_i

$$W(P_i) \coloneqq \sum_{j=1}^{i} E_j - A_i$$

Average Waiting Time

$$\frac{1}{n}\sum_{i=1}^{n} W(P_i) = \sum_{i=1}^{n}\left(\sum_{j=1}^{i} E_j - A_i\right) = \sum_{i=1}^{n}\sum_{j=1}^{i} E_j - \sum_{i=1}^{n} A_i$$

Properties (Preemptive) Let $P_1, \cdots, P_n$ be processes in a queue that arrived in that order and let $E_1, \cdots, E_n$ and $A_1, \cdots, A_n$ the ordered execution and arrival times (ascending).

Waiting Time for process P_i complicated

Average Waiting Time even more complicated

5.4.3 Round Robin (RR/TDMA) with Fixed Slots

- Circular queue Q of processes.
- Each process gets a fixed slice of CPU time (time quantum q).
- If one of these slots finishes, the process is preempted and added to the end of Q (sliding window).
- If n processes are in the queue, each process gets $\frac{1}{n}$ of CPU time in chunks of q time units. Therefore, no process waits more than $(n-1)q$ time units.

- Choosing q and n:
  - If q is too large, the system has a poor response time.
  - If q is too small, the system has to context switch all over the time which leads to a lot of overhead and reduces the efficiency.
  - To stop the first from happening, a *dynamic RR* can be used. If a process finishes in less than q time units, the next slot starts earlier.
- RR typically has a higher turnaround time, but better response time.
- If static scheduling is used, the correct choose of q is crucial for the system to work well.
- Alternative: Have quanta of different lengths (dynamic scheduling costs roughly as much as SJF).

Input Queue Policy

- **Static**
  - Preemptive FCFS scheduling with predictable waiting time.
  - Issue: Setting up the number and the size of slots based on the number of jobs.
- **Dynamic**
  - If Q keeps adding new jobs, the slot waiting time can vary (extend) for existing jobs, potentially leading to starvation.
  - Issue: Throttle the number of incoming requests.

Properties (Preemptive) Let $P_1, \cdots, P_n$ be processes in the queue and let the queue to have N spaces.

Waiting Time for process P_i

$$W(P_i) \leq (n-1)q$$

Average Waiting Time

$$\frac{1}{n}\sum_{i=1}^{n} W(P_i)$$

5.4.4 Priority Scheduling (PS)

- Each process gets assigned a priority number.
- The CPU is allocated to the process with the highest priority.
  Typically, the highest number has the highest priority, but in Linux the lowest number has the highest priority.
- Basically, all other scheduling policies can be broke down to PS:
  - FCFS: The priority is determined by the arrival time.
  - SJF: The priority is determined by the execution time.
  - RR: The priority is equal for all processes.

- Options:
  - **Static Priority:** The initial execution order is chosen by fixed priorities. Starvation problem: Low priority processes may never execute.
  - **Dynamic Priority:** The priority is modifiable and order at runtime. Aging solution: Adjust the priority of a process as time progresses.
- Calculation of the priority in Linux (recap: low number means large priority):

```
P_user-pri = P_USER + P_cpu / 4 + (2 * P_nice)
```

  - If the process uses the CPU, `P_cpu` increases and therefore the number goes up, the priority down.
  - If the process is waiting, `P_cpu` decreases and therefore the number goes down, the priority up.

5.5 Scheduler Location

- **Kernel-Level**
  - Flexible and efficient for quanta usage, but expensive full context switch for threads.
  - Monolithic kernel scheduler also avoids thread blocks.
  + Kernel has full overview and control.
  – Complex scheduling for high number of threads.
- **User-Level**
  - Simple process-level context switching, but thread blocking blocks the process.
  - Split level scheduling (kernel and dedicated application/process/thread specific), allows process/application specific schedulers.
  + Allows processes/apps to select customized schedulers.
  – Kernel loses overview and control on delegation.
- **Local Scheduling**
  The thread library decides which thread to but onto an available thread.
- **Global Scheduling**
  The kernel decides which kernel thread to run next.

5.6 Multilevel Feedback Queue

Using three queues for scheduling:

- Q_0 (RR with time quantum 8ms)
- Q_1 (RR with time quantum 16ms)
- Q_2 (FCFS)

1. New jobs are placed in Q_0 and receives 8ms of CPU time.

2. If the job is not finished, it is preempted and placed in Q_1 and receives 16ms of CPU time.
3. If the job is not finished, it is preempted and placed in Q_2 until is completes.

5.7 OS Examples (Solaris, Windows, Linux)

- All work with dynamically altering priorities and time slices.
- All have preemptive tasking, whereas kernel processes are generally non-preemptive and user processes are mostly preemptive.

5.7.1 Solaris

5.7.2 Windows

5.7.3 Linux

Two algorithms: time-sharing and real-time.

- **Time Sharing**
  - Prioritized credit-based (process with most credits is schedules next).
  - Credits are subtracted when time interrupt occurs.
  - When credit is zero, another process is chosen.
  - When all processes have a credit of zero, re-crediting occurs (based on factors including priority and history).
- **Real Time**
  - Soft real-time
  - Two classes: FCFS and RR, but highest priority always runs first.

6 Mutual Exclusion

6.1 Concurrent Access to Shared Resources

Mutual exclusion is concurrent accesses to shared resources by multiple processes/threads. If the access is blocking it may produce a deadlock, if not, it possibly crash. → mutual exclusion control.

6.2 Taxonomy of Software Misbehavior

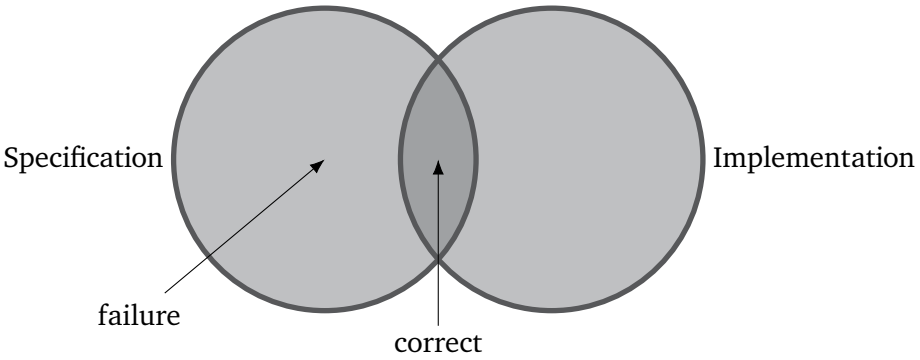

Figure 6.1: Taxonomy of Software Misbehavior

- **Failure:** Deviation from a specified service.
- **Service:** Sequence of external states.
- **External state:** Everything that is visible by other entities in the system.

6.2.1 Types of Bugs

- **Bohr bug:** A repeatable bug that manifests reliable under a possible unknown but well-defined set of conditions.
- **Mandelbug:** A bug whose underlying conditions are so complex and obscure as to make its behavior chaotic or non-deterministic.

6.3 Critical Sections

A set of instructions that cannot be accessed by more than one process at a time is called a *critical section* (CS).

6.4 Mutual Exclusion and Critical Sections

Mutual exclusion (ME) contains the following four elements:

Unity The same CS is only accessed by one process.

Fairness Each process has to move forward in its execution with a speed greater than zero.

Progress Processes outside of CSs should not block processes from accessing the CSs.

Bounded Waiting Every process must be able to enter the CS with a finite delay (i.e. it should not hang forever).

But mutual exclusion does not guarantee correctness if the execution order of two processes matters.

6.5 Mutual Exclusion Strategies

There are multiple basic ME strategies:

- Interrupts
- Process Alternations
- Locks
- Semaphores

6.5.1 Interrupts

- Core problem: Arbitrary instruction reordering.
- If no interrupts exist, no reordering takes place.
- Classical interrupt handling:
  - Every interrupt has a specific *interrupt priority level* (`ipl`).
  - For each process, the kernel maintains the `current_ipl`.
  - If `incoming_interrupt_ipl > current_ipl`, then handle the interrupt, else wait.
- For ME, interrupt masking is used (i.e. disable interrupts):

```
Enter CS
    // Disable interrupts.
    set current_ipl = max_ipl;

    // Execute CS.

    // Restore old interrupts.
    restore_ipl
Exit CS
```

- If the old `ipl` is lost, the process hangs and other processes are blocked. Solution: Segregated kernel and user ipl levels.
- Multiprocessor-Systems: CPU 1 disabled, processes on CPU 2 can execute.

6.5.2 Process Alternation

- Both programs loop all the time.
- They both enter the CS all the time if the state allows them to do so (i.e. sets that its their turn).
- After one program was unable to execute its turn, it sets the turn variable so the next round is theirs.
- That way, the programs alternate during the execution.

Peterson's Algorithm

6.5.3 Locks

- If the lock is unlocked, then acquire a lock and enter the CS.
- After finishing the work, release the lock and leave the CS. This is called a *test-and-set lock* (TSL).
- But: Same problem as with process alternation, as locking does not happen atomically (it is first read and then written and not in one step).
- Solution: Provide a lock primitive for TSL that does the test-and-set action atomically.

Spin Locks

- A spin lock just loops doing nothing and waits for a condition to come true to enter the CS.
- But: The lock wastes CPU and has the risk of starvation.

Barriers

- A *barrier* is a synchronizer for all processes that
- blocks all arriving processes until all approaching processes arrived and then
- releases all at once.

6.5.4 Semaphores

- Semaphores is a software mechanism for synchronization on a higher level than TSL assembly.
- They contain an integer counter and a queue that contains all waiting processes. The queue may or may not be stored in the semaphore itself. If the integer is binary, it is called a *mutex* semaphore.
- The two primitive and atomic methods operations are:
  - `wait()`
    Decrements the value. Blocks the process and puts it into the queue if the new value is below zero.
  - `signal()`
    Increments the value. Starts the next process of the queue and pops it if the new value is at least zero.

Example: Ordering

- Consider two concurrent processes P_1, P_2 with statements S_1, S_2.
- Desired behavior: S_2 must only be executed after S_1 completed.

P_1:

```
S2;
signal(sync);
```

P_2:

```
S2;
signal(sync);
```

Whereas the semaphore must be initialized with the counter value zero to ensure the desired behavior.

Example: Producer/Consumer

```
// Number of slots in the buffer.
#define N 100;

// The special semaphore int.
typedef int semaphore;

// Controls access to the critical section.
semaphore mutex = 1;
// Counts empty buffer slots preventing a buffer underflow.
semaphore empty = N;
// Counts full buffer slots preventing a buffer overflow.
semaphore full = 0;
```

Figure 6.2: Example: Semaphores: Definitions

```
// Producer.
void producer() {
    int item;

    while (TRUE) {
        item = produce_item();

        // Decrement empty count or wait for free slots.
        down(&empty);

        // Ensure mutual exclusion and enter CS.
        down(&mutex);

        // Critical section (CS): Insert item into buffer.
        insert_item(item);

        // Release the lock and leave CS.
        up(&mutex);

        // Increment count of full slots.
        up(&full);
    }
}
```

Figure 6.3: Example: Semaphores: Producer

```c
// Consumer.
void consumer() {
    int item;

    while (TRUE) {
        // Decrement full count or wait for new items.
        down(&full);

        // Ensure mutual exclusion and enter CS.
        down(&mutex);

        // Critical section (CS): Take item from buffer.
        item = remove_item();

        // Release the lock and leave CS.
        up(&mutex);

        // Increment count of empty slots.
        up(&empty);

        // Do something with the item.
        consume_item(item);
    }
}
```

Figure 6.4: Example: Semaphores: Consumer

6.6 Classical Problems

- Synchronization problems (Dining Philosophers Problem)
- Access problems (Readers/Writers Problem)

6.6.1 Dining Philosophers

- Five philosophers sit around a table, each with a dish and two chopsticks.
- No one picked up yet its chopsticks, so it is unclear which is owner my whom.
- They can only pick one instrument at a time.

→ Deadlock, everyone is waiting on any other person.

- Solution is possible using semaphores.

6.6.2 Readers/Writers Problem

- A data set is shared among a number of concurrent processes. Readers can only read the database whereas writers can also write data.
- Problem: Allow multiple readers to access the resources at a time but only allow one writer to access it at a time.
- Solution is possible using semaphores.

7 Memory Management

7.1 Abstraction

- The goal is to let the development take place without making any assumptions on the system that may execute the program ((un)usable memory addresses, interference of memory accesses, ...).
- A program uses memory every time a variable is assigned, an array is created, ...
- Heap: Explicit memory allocations. Stack: Implicit/automatic allocations.
- Naive abstraction: Only one process in memory at a time. This is a massive performance drawback.

7.2 Avoiding Slow Disk Loads in Commodity Systems

The following parts are needed:

- sufficient memory
- an OS data structure that keeps track of which spaces are used or free
- relocation of addresses
- protected of the OS from processes and processes from each other

7.2.1 Address Spaces and Implementations

- Each program lives in its own memory with its own addresses (*address space*).
- No assumption is made about other processes and a process cannot access the memory of the OS or other processes.
- The basic approaches for implementing address spaces are:
  - Base and Bounds
  - Segmentation
  - Paging
  - other compiler-based approaches

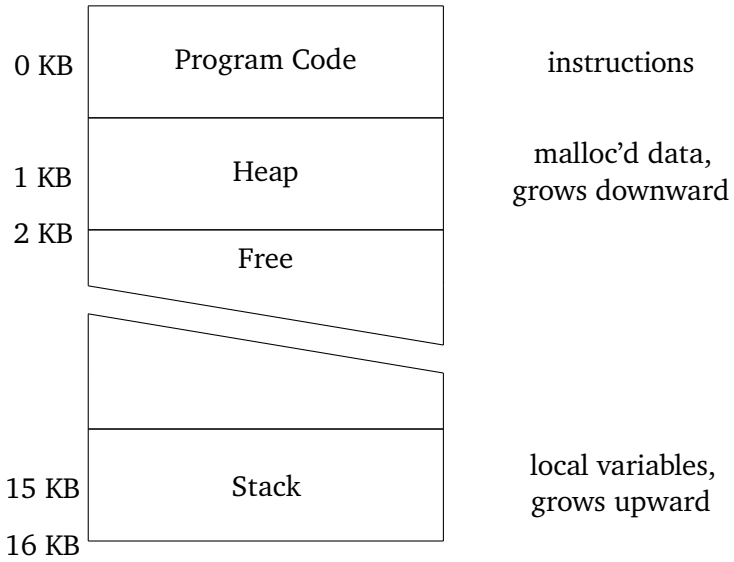

Figure 7.1: Address Spaces: Memory Layout

7.3 Base and Bounds

- Core idea:
  - Make all address spaces starting from zero.
  - Load the process to a specific address x.
  - Add x to every memory reference at run time.
- This needs either memory address rewriting at load time or hardware support.

7.3.1 Memory Layout

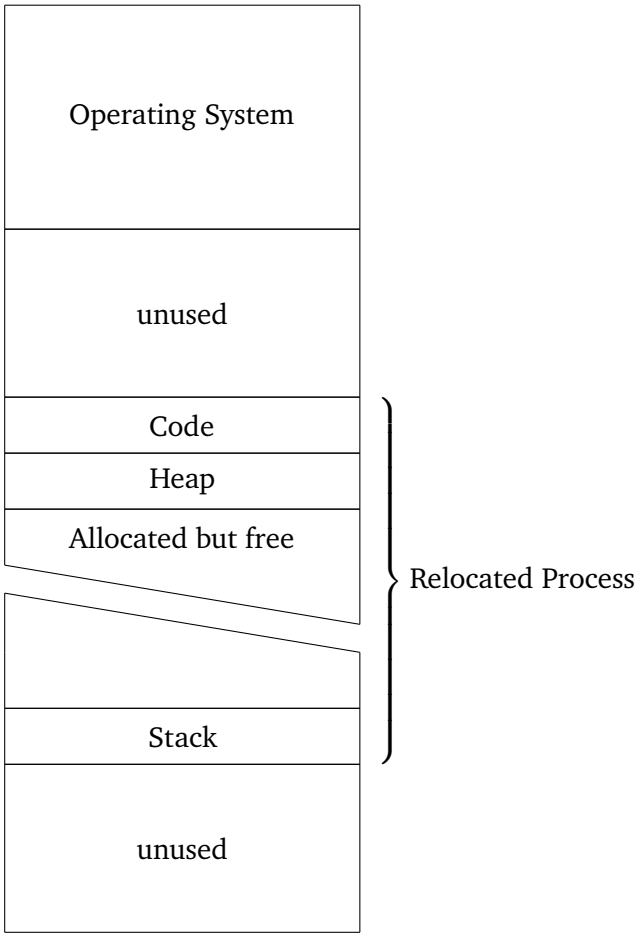

Figure 7.2: Base and Bounds: Memory Layout

- The *base register* contains the address there the program's address zero is assumed.
- The *bound register* contains the program's address where the available memory space ends.

7.3.2 Memory Safety

- OS safety: The base register is set by the OS to the lowest possible address.
- Process safety: Bounds register set by the OS is the highest accessible address.
- If a process exceeds this bounds, an exception (or any other interrupt) happens crashing the process.

7.3.3 Requirements

- Hardware
  - Base/bounds registers.

- Privileged mode to protect base/bounds registers.
- System calls to allow user processes to invoke OS code.
- Memory management units (MMUs) to offset memory addresses.
- Runtime exception mechanisms for access violations.

- Operating System
  - Track keeping of the un-/used memory.
  - Initialization and updating of the base/bounds registers.
  - Exception handling for access violations.

7.3.4 OS/Hardware Interactions

Boot

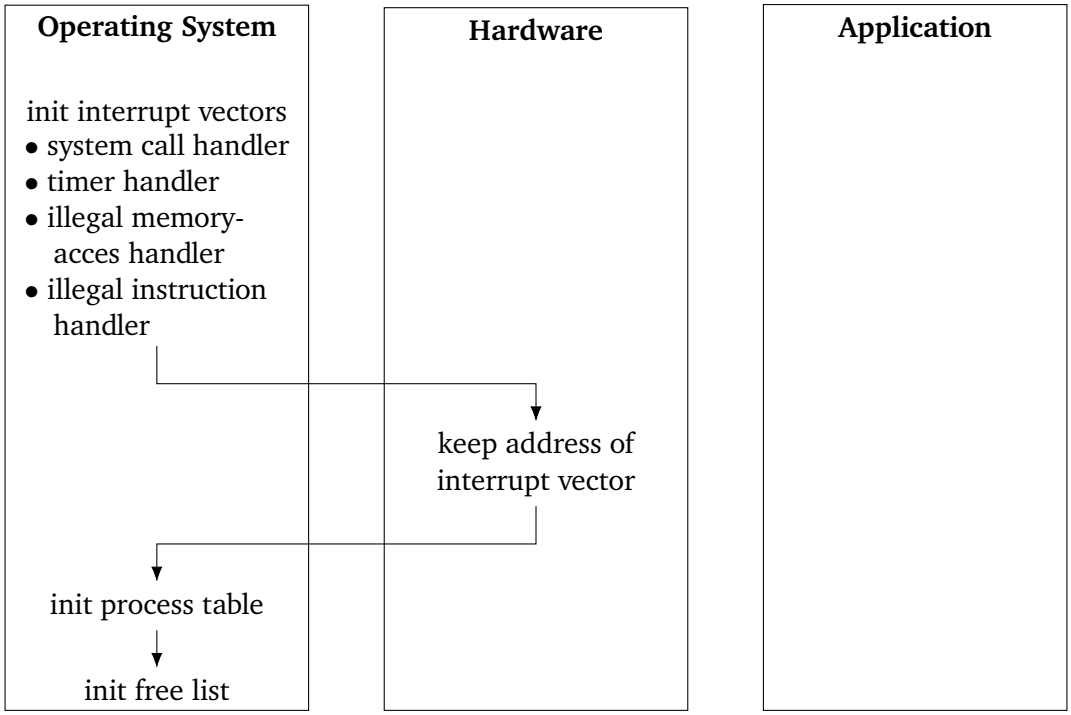

Figure 7.3: Base and Bounds: OS/HW Interactions on Boot

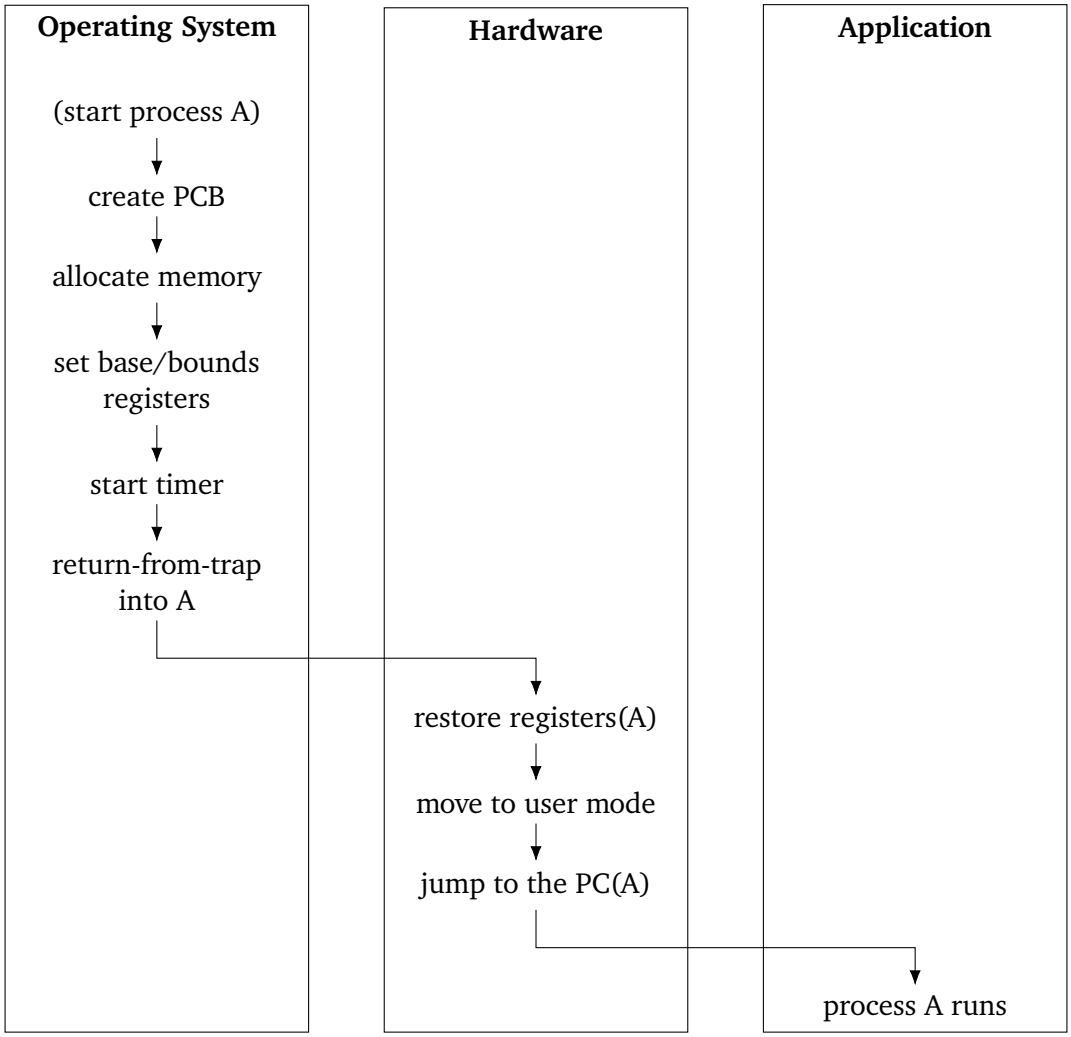

Figure 7.4: Base and Bounds: OS/HW Interactions on Process Creation

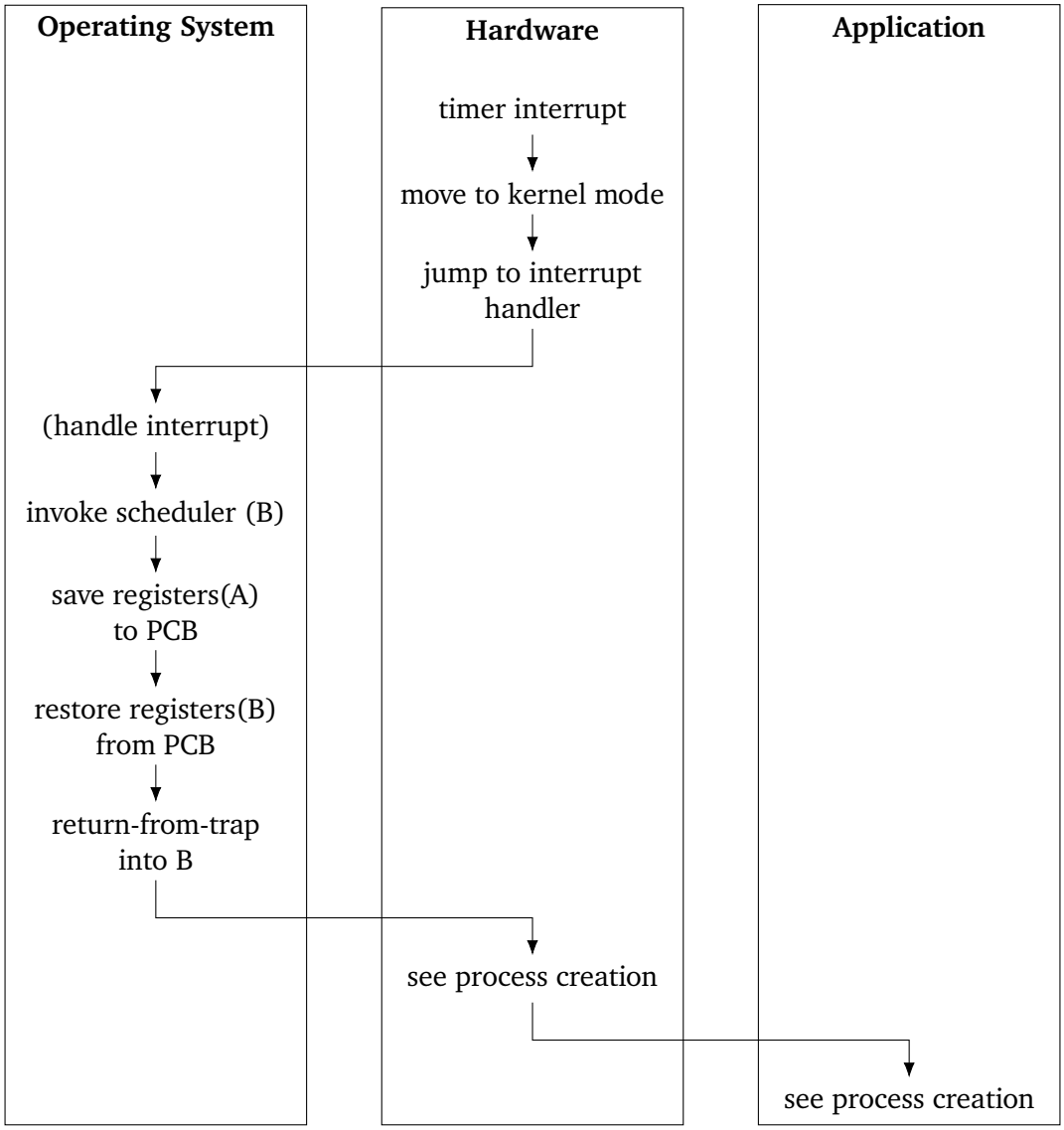

Figure 7.5: Base and Bounds: OS/HW Interactions on Context Switch

Process Execution (Memory Load)

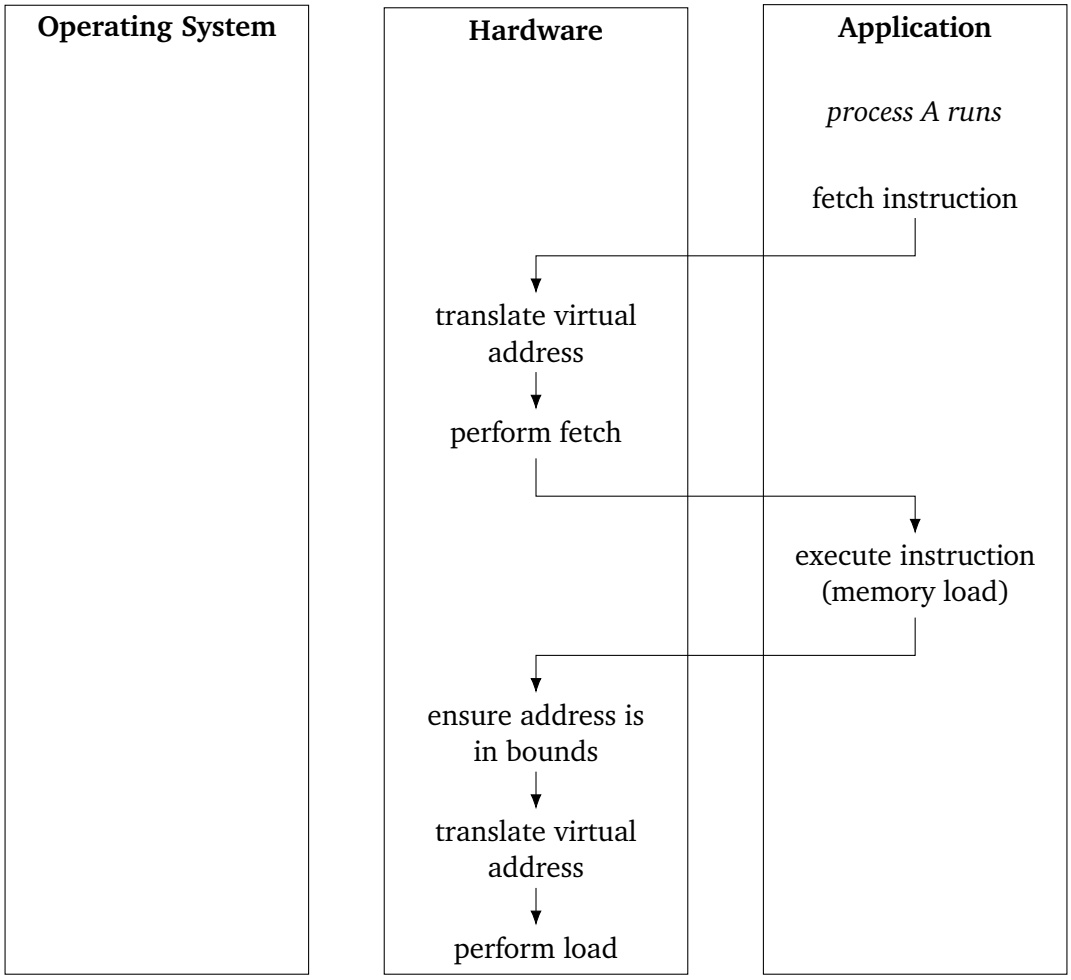

Figure 7.6: Base and Bounds: OS/HW Interactions on Process Execution (Memory Load)

Process Execution (Failure)

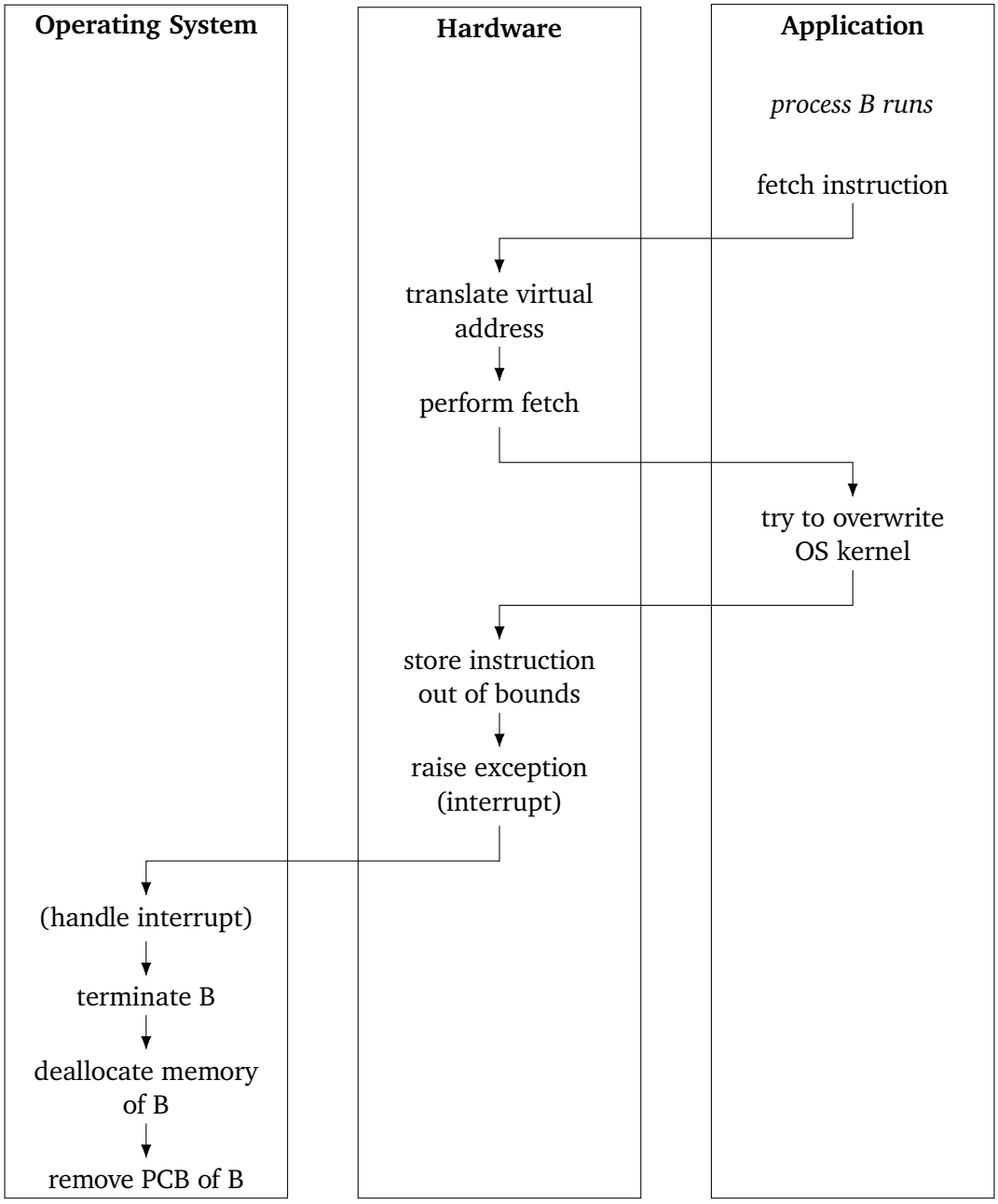

Figure 7.7: Base and Bounds: OS/HW Interactions on Process Execution (Failure)

7.3.5 Pros and Cons

+ Provides address space abstraction.
+ Provides OS and process memory safety.
+ Butter than the single process model (the naive abstraction model).

– Internal fragmentation.

7.4 Segmentation

- Core idea:
  - Base/bounds per segment.
  - More fine-grained distribution of process memory.
  - Sharing of read-only segments across processes.
- Needs more registers and a more complex address translation (based on which segment is addressed).

7.4.1 Memory Layout

7.4.2 Address Translation

7.4.3 Stack Address Calculation

7.5 Paging

- Goal: Tame external fragmentation.
- Idea:
  –
  - Split the address space in equally sized *pages*.
  - Split the physical memory in *page frames* with identical site as the pages.
  - Perform address translation per page.
- If there are a lot of pages, the base addresses have to be stored in memory rather than in registers doubling the memory latencies, which then can be reduced by using caches.
- Large address spaces yield large page tables that occupy memory.
- Memory utilization can be improved by applying segmentation of paging to the page tables itself (at the cost of higher latencies).
- Another advantage of paging is that the hard disk can be used as a memory extension relatively easily.
- Terminology:
  - VPN: Virtual Page Number
  - PFN: Physical Frame Number

7.5.1 Implementation

- Naive implementation: One PFN register per frame number.
  - This works if there are only a few pages in physical memory.
  - Few pages lead to large pages which then leads to internal fragmentation.
  - Realistic Example: 4 KiB pages in 32 bit address space $\rightarrow 2^{20} \approx 1$mio. registers needed.
- Better: PFN list in memory as the *page table*.
  - Has as many pages as pages in the logical address space.
  - The translation hardware holds the page table base pointer.
  - The logical page number is the index into the table.
  - The table holds the physical frame number, a validity bit and other management information.

7.5.2 Problems

- Problem 1: Size and memory
  - 4 bytes per entry $\times 2^{20}$ entries $\rightarrow 4$ MiB per process.
  - If kept in memory: 1 GiB memory for 250 processes...
  - If not: the information has to be load from disk every time a context switch happens (time consuming).
  - Neither is acceptable $\rightarrow$ hybrid paging (see 7.5.4), multilevel page tables (see 7.5.5).
- Problem 2: Latency $\rightarrow$ caching (see 7.5.3).

7.5.3 Caching (TLB)

- The table cache (TLB) contains:
  - VPN
  - PFN
  - Protection bits
  - Validity bit
  - Some bits for cache management (dirty bit, coherency config, etc.)
  - Sometimes: Address Space Identifier (ASID)
- Problem: Colliding TLB entries from different processes.
  - If the TLB has ASID fields, write the process ID in the ASID fields and keep entries on context switch.
  - If not, flush the TLB on each context switch.

7.5.4 Hybrid Paging and Segmentation

- Problem: The page table size is often bigger than needed.
- Idea: Create page table segments for the code, heap and stack pointers.
- This is called *hybrid paging and segmentation*.
- Pros/Cons:
  - \+ Simple utilization of existing mechanisms.
  - \+ Limits page table size effectively.
  - – Requires separation in segments (obviously).
  - – Dynamically growing segments cause external fragmentation.
  - – No more segmentation in 64 bit mode on amd64 CPUs.

7.5.5 Multi-Level Page Tables

- Facing the same problem as hybrid paging and segmentation.
- Idea: Do not segment the page table but rather page the page table again (*multi-level*).

7.6 Swapping Pages to Disk

- Swapping out blocks with same size as pages to the disk free memory (if there are more virtual pages than physical frames).
- This requires a bit in the page table to indicate the presence of a page (*present bit*).
- Requires swap space storage management by the OS.
- If a page is not present on a access attempt, this is called a *page fault*.
- If a page fault happens, the page has to be loaded from the disk. If no memory is free, a loaded page has to be replaced. There are multiple possible *page replacement policies*:
  - – Belady's optimal MIN policy
  - – First In, First Out (FIFO) and variants
  - – Random replacement
  - – Least Recently Used (LRU) and approximations (most commonly used)
  - – etc.

7.6.1 Belady's Optimal Replacement (MIN)

Principle: Swap out the page that next usage is the farthest away in the future.
Disadvantage: In theory none, in practice this algorithm is impossible to implement.

Example Assume 3 page frames for 4 pages with VPNs $0, 1, 2, 3$ and the VPN access sequence: $0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1$.

Reference	HitMiss	Evict	Present
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3 (or 0)	0, 1, 2
1	Hit		0, 1, 2

Table 7.1: Example: Belady's Optimal Page Replacement

7.6.2 First In, First Out Replacement (FIFO)

Principle: Maintain a list of pages in the order they come in and evict the page at the beginning the list (that came in first).

Disadvantage: The page that is longest in the memory may be used often but is still replaced.

Example Assume 3 page frames for 4 pages with VPNs $0, 1, 2, 3$ and the VPN access sequence: $0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1$.

Reference	HitMiss	Evict	Present
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	2	3, 0, 1
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Table 7.2: Example: FIFO Page Replacement

7.6.3 Second Chance FIFO Replacement

Principle: Each page gets a *second chance bit* (R bit) that is set to 0 for new pages and the pages are sorted in FIFO order. Set the R bit to 1 each time a page is referenced. When looking for a page to replace cycle through the list skipping all pages with the R bit set to 1 while setting it to 0 and evict the first page with the R bit set to 0.

Example Assume 3 page frames for 4 pages with VPNs $0, 1, 2, 3$ and the VPN access sequence: $0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1$. If the second change bit (R bit) is set on a page, it is marked with a tilde.

Reference	HitMiss	Evict	Present
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		$\tilde{0}$, 1, 2
1	Hit		$\tilde{0}$, $\tilde{1}$, 2
3	Miss	2	0, 1, 3
0	Hit		0, $\tilde{1}$, 3
3	Hit		0, $\tilde{1}$, $\tilde{3}$
1	Hit		0, $\tilde{1}$, 3
2	Miss	0	$\tilde{1}$, 3, 2
1	Hit		$\tilde{1}$, 3, 2

Table 7.3: Example: FIFO Page Replacement

7.6.4 Random Replacement (RR)

Principle: Replace a randomly selected page.

(Dis-) advantages: No bookkeeping nor logic is required, but the chances to pick the best and the worst are equal.

Example Assume 3 page frames for 4 pages with VPNs $0, 1, 2, 3$ and the VPN access sequence: $0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1$.

Reference	HitMiss	Evict	Present
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	3, 1, 2
0	Miss	1	3, 0, 2
3	Hit		3, 0, 2
1	Miss	3	1, 0, 2
2	Hit		1, 0, 2
1	Hit		1, 0, 2

Table 7.4: Example: Random Page Replacement

7.6.5 Last Recently Used Replacement (LRU)

Principle: Evict the page that was used least recently.

Example Assume 3 page frames for 4 pages with VPNs $0, 1, 2, 3$ and the VPN access sequence: $0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1$.

Reference	HitMiss	Evict	Present
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	0	2, 1, 3
1	Hit		2, 1, 3

Table 7.5: Example: Last Recently Used Replacement

8 Input/Output

8.1 Hardware

8.1.1 Device Types

- Keyboard
- Mouse
- Storage (HDD, SSD, USB drive, . . .)
- Network interface cards
- Graphics cards
- . . .

It is important to distinguish between specific devices in terms of what they do, but it is possible to hide the information how they do what they do.

8.1.2 Device Controllers

- I/O devices have sometimes mechanical components and always electronic components.
- The *device controller* is the interface that is accessible by the OS and may be able to handle multiple devices.
- Example controller tasks: Convert bit streams to block of bytes, perform error correction, . . .

8.1.3 Accessing Devices

- Most I/O device controllers provider in/out buffers, control and status registers.
- These are accessible from the OS and applications via I/O ports, mapped memory or a hybrid version of both.

Mapped Memory

The memory of the device is mapped onto the address space of the OS/application.
Pros/Cons:

+ No special assembler instruction is needed for communication.
+ No special protection is needed as it is inherited from the virtual memory.

+ It is possible to test the status registers directly.
– It steals addresses from the address space.

8.1.4 Direct Memory Access

- With *direct memory access* (DMA), the memory bus can be accessed directly without using the CPU. That way, the CPU is not utilized for such dumb operations and can be used for more important operations.
- DMA is done by using a DMA-controller that is attached to the memory bus.
- Notice that DMA is not a replacement for in/out buffers not mapped memory but can apply to both of these options.

8.1.5 Programmed I/O

- Reading the I/O device data is done by looping a read function constantly reading the data waiting for a specific response.
- This works, but produces lots of overhead because the CPU is always busy doing nothing but looping.

8.1.6 Interrupts

- The devices send interrupts to the interrupt controller of the CPU.

Multiprocessor Interrupts

- In multiprocessor environments, a local *advanced programmable interrupt controller* (APIC) is available per core for controlling internal CPU stuff.
- The I/O interrupts use a separate APCI outside of the cores (therefore centralized) that then handles the interrupts and decides where to handle the interrupt.

Interrupt-Driven I/O

- Using interrupt-driven I/O, the code only runs if an interrupt occurs.
- Therefore, the interrupt handling itself should be fast to not miss an interrupt.
- If there are a lot of things to do, the handling should be split up into two parts:
  1. Disabled interrupts, do only essential work.
  2. Enabled interrupts, do most the work.
- The following steps must be done in software when interrupts occur (not complete):
  1. Save registers that are not already saved by interrupt hardware.
  2. Set up the context for the interrupt service procedure.
  3. Acknowledge the interrupt controller and re-enable interrupts.
  4. Run the service procedure.
  5. Schedule and run a new process (and new context).

8.1.7 Buffer Strategies

Buffering can massively improve the performance and there exist four basic buffering options:

- unbuffered input
- buffering in user space
- buffering in the kernel followed by copying to user space
- double buffering in the kernel followed by copying to user space

8.2 Device Diversity and Drivers

- As only the device manufacturer exactly knows how its device work, it must provide a device specific *driver*.
- Such drivers include
  - interrupt handling code,
  - device initialization code,
  - power state handling,
  - device controller-specific operations and
  - synchronization across device users.
- This device diversity causes high OS diversity as each machine needs its own, very specific, drivers.
- Solution: Decouple driver code from the OS build process and create a dynamic plug-in architecture for the drivers. The drivers are then compiled against a specific kernel version and are loaded at runtime.
  + It is not needed that the supported devices are known during the OS compile time.
  + A common interface makes the driver programming easier.
  – A large number of drivers spread and the amount of code one blindly trusts enlarges.
  – The user has to trust the driver as it runs in kernel mode and is not isolated from the rest of the kernel (i.e. drivers can kill your system).
- Solution attempts to the trustworthiness problem:
  - User mode drivers
    * Too slow in general as I/O is on a slow path and adding mode switches does not help.
    * But works for some drivers that are not required to be extremely fast.
  - Split mode drivers (“Microdrivers”)
    * Analyze which parts of the driver are not performance critical and move these into user mode.
    * This only stops a few driver bugs.
  - Improve driver testing
    * Limiting tests for relevant driver state space.
    * Testing drivers in user mode.

8.3 I/O Interfaces and Abstraction

The abstraction of I/O interfaces is not standardized and thus every OS produces its own model.

Example: Linux Device Interface

- Devices are addresses using special file systems, e.g. sysfs or udev.
- Existence of special commands to access/modify dev files, e.g. mknod or udevinfo.
- Files are connection to drivers through their version number.
  - The major number identifies the device type (and driver).
  - The minor number identifies the (virtual) device instance.

9 File Systems

Preface: For this chapter it is assumed that disks are sequences of blocks that can only be accesses in sequence.
A file system has to:

- store the actual file data,
- store the metadata (access rights, creation/modification/access/birth dates, if a file is references by hard links) and
- relate the file data with the metadata.

9.1 Abstraction

A file system can be structured using folders and files which together form paths.

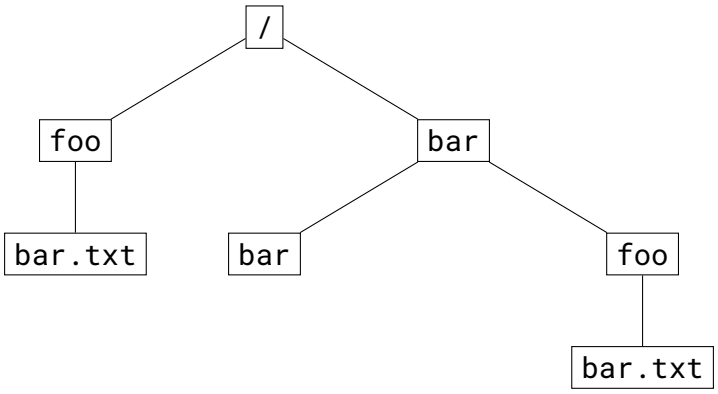

Figure 9.1: File Systems: Hierarchy

A *file extension* is commonly used as an indicator of the file content, e.g. `.txt` for text files, `.c` for C source code, `.sh` for shell scripts, etc. In general, these are not necessary (on the most systems) must helpful.

9.2 Common File Operations

9.2.1 File Descriptors

- A *file descriptor* (FD) identifies a specific file or similar.
- It is capable of performing a set of operations.
- Identified by consecutive integer number, whereas 0, 1, 2 are reserved for standard input (stdin), standard output (stdout) and error output (stderr).

- In POSIX, stderr is open for reading and writing.

9.2.2 `open()`

```
int open(const char *path, int oflag, ...)
```

- Returns a file descriptor for the named file.
- The *path* argument points to a pathname that names the file to open.
- The *oflag* parameter is one of these flags:
  - `O_EXEC` Open for executing only (non-directory files).
  - `O_RDONLY` Open for reading only.
  - `O_RDWR` Open for reading and writing.
  - `O_SEACRH` Open directory for searching only.
  - `O_WRONLY` Open for writing only.

  combined (bitwise *or*) with any other flags defined in `<fcntl.h>`.

9.2.3 `read()`

```
ssize_t read(int fildes, void *buf, size_t nbyte)
```

- Reads (at most) *nbyte* bytes from the file associated with the file descriptor *fildes* and writes them into the buffer pointed by *buf*.
- If the file supports seeking (e.g. a regular file), the read starts at the position in the file given by the file offset that is associated with the file descriptor.
- Returns the the number of bytes actually read. If the end of file is reached, 0.
- After data was read, the last *data access timestamp* is set to the current timestamp.

9.2.4 `write()`

```
ssize_t write(int fildes, const void *buf, size_t nbyte)
```

- Writes *nbyte* bytes from the buffer pointed by *buf* into the file associated with the file descriptor *fildes*.
- If the file supports seeking (e.g. a regular file), the write starts at the position on the file given by the file offset that is associated with the file descriptor. After a successful write, the file offset is incremented by the actual bytes written.
- Returns the number of bytes actuall written.
- After the data was written, the *last modified timestamp* is set to the current timestamp.

9.2.5 `close()`

```
int close(int fildes)
```

- Deallocates the file descriptor *fildes* and makes the file eligible for `open()` again.
- Return 0 for success and -1 for failure.

9.2.6 POSIX File Redirection

- Input and output of a program can be redirected via the shell:
  - `>` writes the output to a file or file descriptor
  - `>>` appends the output to a file or file descriptor
  - For forward a specific file descriptor, write its number in front of the angle. To redirect all descriptors, use an ampersand (`&`). The implicit file descriptor is stdout.
- A file descriptor can be referenced using an ampersand (`&`) followed by its number.
- There exist a few special files that must only be used as described but provide very useful functionality:
  - `/dev/null` Discards every byte written to it. Returns EOF on every read.
  - `/dev/zero` Returns an infinite number of null bytes. Cannot be written.
  - `/dev/random` Returns an infinite number of random bytes. Cannot be written.
  - `/dev/urandom` Returns an infinite number of more secure random bytes. Cannot be written. Is not available on every POSIX system.

Examples

- Append all regular output to the log file and discard any error output:

```
$ ./a.out >>a.log 2>/dev/null
```

- Discard all output (both stdin and stdout and other file descriptors):

```
$ ./a.out &>/dev/null
```

- Create a file descriptor to access my server, send a HTTP request and read the response (working example):

```
$ exec 3<> /dev/tcp/dmken.com/80  # Open the file descriptor.
$ echo -en 'GET /\r\n' >&3        # Send the request.
$ cat <&3                         # Read the response.
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>302 Found</title>
</head><body>
<h1>Found</h1>
<p>The document has moved <a href="https://www.dmken.com/">here</a>.</p>
</body></html>
$ exec 3>&-                       # Close the file descriptor.
```

9.3 File Information (`stat`)

The `stat` command prints a lot of useful information, including:

- Name (and possible link information)
- Site
- Blocks
- IO Block
- Type
- Device
- Inode
- Links
- Access rights
- Owner (UID)
- Group (GID)
- Last access time
- Last modification time
- Last change time
- Birth time

9.4 Links

9.4.1 Hard Links

- Hard references in the file table.
- If one hard link is deleted, another to the same file still contains the information.
- Most of the time only one hard link exist for each file.
- Can be used to synchronize file contents in multiple locations.

9.4.2 Symbolic Links

- Kinda files containing just the path to the linked file.
- Must be resolved by software.
- If the original file is deleted, the link becomes invalid.
- Works across file system boundaries.
- Can also be used to synchronize file contents.

9.5 Reducing Operation Latencies

1. Keeping the storage information separate from the file data improves the operation latencies.
   This reduces the latencies as follows:
   - Time for reading: 8/2 reads (slow/fast)
   - Time for skipping: 8/2 and 56/2 reads
   - Time for reading a file block
2. To further reduce the operation latencies, the file metadata can be stored besides the storage information in a combined data structure, called *index node* (inode).
   This reduces the latencies as follows:
   - Metadata operations: 8/2 reads (slow/fast)
3. More optimization: Keep track of the free disk space in separate data structures called *bitmap*. This saves time as reading empty inode blocks can be skipped. Also improves file creation as “walking” all inodes is not required.
4. Even more optimization: Add a superblock on the first block that contains information where the other data structures is stored.

9.6 Associating Files with Data Blocks

Problem: How does the file system keep track about which blocks a file occupies and what their order is according to the file data sequence.

There are three different basic approaches:

- Linked lists (used by FAT)
- Indices (used by ext3)
- Extents (used by ext4)

9.7 Linked Lists

- A directory entry contains data and the address of the next block.
- The inodes therefore link to the first block.
- Disadvantage: Reading and especially searching through big files is slow as the complete linked list has to be scanned.

9.7.1 Indices (inodes) Problems

- Large files cause larges inodes.
- If the inode count cannot grow dynamically, it causes external fragmentation. If not, it causes internal fragmentation.

9.7.2 Indirect Addressing

- The root inodes reference an address block that itself references more address blocks containing inodes until the data is reached.
- Example: ext3 used triple indirect addressing.

9.7.3 Extents

- Indirect addressing is efficient for large files, but deletion takes long as all inodes have to be found.
- *Extents* are contiguous data blocks that are reserved after file creation and stored in the inode.
- They contain the address and length of referenced data blocks.
- This produces less metadata than with indirect addressing.
- Performance boost for sequential data accesses.
- Extents are used by ext4, btrfs, NTFS, HPFS and more.
- If no more extent places are free, the extent can be modified that it references another data block that itself contains more extents. In ext4, this can be done up to 5 levels of indirection.

9.8 Common File Systems

9.8.1 vsfs: Very Simple File System

- Just store files and metadata where there is space and
- mark free blocks with some magic number.
- Downsides: Operation latencies (64 slow vs. 2 fast reads, short 64/2). See 9.5 for methods to reduce this latency.

9.8.2 FAT: File Address Table

- Disk blocks are stored as linked lists.
- The inodes (*file address table*, FAT) contain the address of the first data block.
- The data blocks then contain the address of the next data block.

9.8.3 ext3: Third Extended File System

- Use indirect addressing with 3 levels of indirection.

9.8.4 ext4: Fourth Extended File System

- Use up to 4 extents in the inode and build a 5 levels deep tree if 4 extents are not enough.
- The extent structure is 12 bytes long
- ext4 has the following limitations:
  - Maximum file system site is 1 EiB.
  - Maximum extent size is 128 MiB (16 bit extent length and 15 bit for the address where the MSB is used as an initialization flag).
  - Maximum file size is 16 TiB.

9.8.5 FFS: The Fast File System

- Core motivation: improve FS performance.
- Core idea: Use knowledge about how the disk work:
  - Moving the heads to adjacent tracks is fast.
    $\rightarrow$ Place blocks of a file in a single cylinder group.
  - Moving the heads to central metadata is slow.
    $\rightarrow$ Replicate metadata in each cylinder group.
  - Platters rotate continuously.
    $\rightarrow$ Store data in sequential blocks.
  - Data block accesses exhibit locally.
  - Large files to not benefit from locality beyond raw transfer rate saturation.
- FFS is drastically faster than vsfs, but provides the same (standardized) interface.
- Drawback: FFS is not optimized for SSDs, so more file systems are needed.

9.8.6 VFS: Virtual File System

- Generic file system structure
  - Provided abstraction from specific file system implementations.
  - Concurrently supports many FS, both local and networked.
  - Uniform FS interface $\rightarrow$ FS is transparent for applications.
  - Mapping between VFS and actual FS is required.
  - Linux implements VFS.

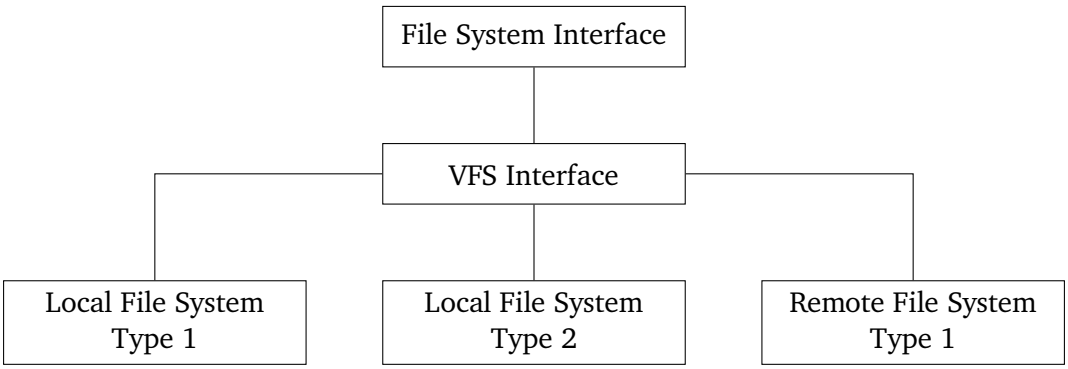

Figure 9.2: VFS: General Principle

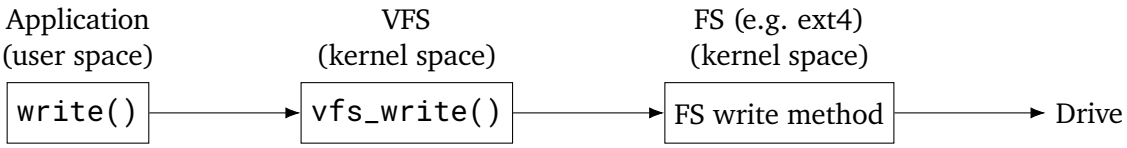

Figure 9.3: VFS: Write Access

- Is object-oriented (separation of generic operations from FS specifics, each object provides a set of operations).
- VFS has four important objects:
  - **inode** Index node that contains file information (FCB).
  - **file** An *open* file that is currently associated to a process.
  - **dentry** Directory entry that contains single components of a path.
  - **super block** Contains detailed information about a specific FS instance.
- Directories are handled like files, i.e. dentry objects could also be a file, the data structure does only exist in memory, not on the disk.
- Directories are special files that are stored in inodes.

inode Object

- UNIX: inodes can be read directly from the disk.
- Get created by the FS.
- In UNIX, everything is a file, so there are a few special inodes:
  - *i_pipe* Pipes.
  - *i_bdev* Block devices.
  - *i_cdev* Character devices.
- Supports direct and indirect data addressing.

- The file systems have to implement the inode operations of VFS and must map their structures and operations onto the provided inode interface. Excerpt of the methods to implement:
  - `int (*create) (struct inode *,struct dentry *, umode_t, bool)`
  - `int (*link) (struct dentry *,struct inode *,struct dentry *)`
  - `int (*unlink) (struct inode *,struct dentry *)`
  - `int (*symlink) (struct inode *,struct dentry *,const char *)`
  - `int (*mkdir) (struct inode *,struct dentry *,umode_t)`
  - ```
    int (*atomic_open)(struct inode *, struct dentry *,
                       struct file *, unsigned open_flag,
                       umode_t create_mode, int *opened)
    ```

9.9 Performance and Timing

When using vsfs with inode data structures, data block usage and inode block usage bitmap it is still pretty slow.

9.9.1 Recap: Hardware

- A HDD contains platters, cylinders, sectors, ... that are physically moving parts. These introduce latency to reading.
- The latency is made up two major components:
  - Seek time T_{seek}
    The time for the disk arm to move its heads to the cylinder containing the desired sector.
  - Rotational latency T_{rot}
    Additional time for the disk to rotate the desired sector to the disk head.
- The I/O rate (bandwidth) R_{io} that describes the total number of bytes that can be transferred against the total time it took can be described as:

$$R_{\text{io}} = \frac{\text{Size}_{\text{trans}}}{T_{\text{seek}} + T_{\text{rot}} + T_{\text{trans}}}$$

  where $\text{Size}_{\text{trans}}$ is the data size to transfer and T_{trans} is the raw transfer time between disk and host. If only the transfer rate R_{trans} is given, it holds $T_{\text{trans}} = \frac{\text{Size}_{\text{trans}}}{R_{\text{trans}}}$.
- Of course, the disk access times can vary depending on the head position and position of the wanted block. File systems can effect the position of the wanted block and have to optimize that one.

Sequential vs. Random Access Sequential access is much fast than random access as neither the disk head nor the sectors have to be relocated:

Assume $T_{\text{seek}} = 9\text{ms}$, $T_{\text{rot}} = 4.2\text{ms}$, $R_{\text{trans}} = 105\text{MiB/s}$, $\text{Size}_{\text{block}} = 4\text{MiB}$ and $\text{Size}_{\text{trans}} = 100\text{MiB}$.
    ```

Sequential Access

$$R_{\text{io}}^{\text{seq}} = \frac{\text{Size}_{\text{trans}}}{T_{\text{seek}} + T_{\text{rot}} + \frac{\text{Size}_{\text{trans}}}{R_{\text{trans}}}} = \frac{100\text{MiB}}{9\text{ms} + 4.2\text{ms} + 952\text{ms}} \approx 103.6\text{MiB/s} \approx 9.7\text{ms/MiB}$$

$\rightarrow 970\text{ms} = 0.97\text{s}$ for 100MiB.

Random Access

$$R_{\text{io}}^{\text{rnd}} = \frac{\text{Size}_{\text{trans}}}{\text{Size}_{\text{block}}} \cdot \frac{\text{Size}_{\text{trans}}}{T_{\text{seek}} + T_{\text{rot}} + \frac{\text{Size}_{\text{trans}}}{R_{\text{trans}}}} = \frac{100\text{MiB}}{4\text{MiB}} \cdot \frac{100\text{MiB}}{9\text{ms} + 4.2\text{ms} + 952\text{ms}} \approx 0.3\text{MiB/s} \approx 3.3\text{s/MiB}$$

$\rightarrow 330\text{s} = 5.05\text{min}$ for 100MiB.

9.9.2 Scheduling

- In a multiprogramming environment, the storage requests have to be scheduled to minimize the head movement and maximize the throughput.

First Come, First Serve (FCFS)

Principle: Serve the requests in the queued order.
Pros/Cons:

+ Very simple.
– Lots of head movement.
– Poor latency.

Example Assume incoming requests for a disk with sectors 0 to 199 and the request queue

$$98, 183, 37, 122, 14, 124, 65, 67$$

and with the head starting at cylinder 53.
640 cylinders of disk movement:

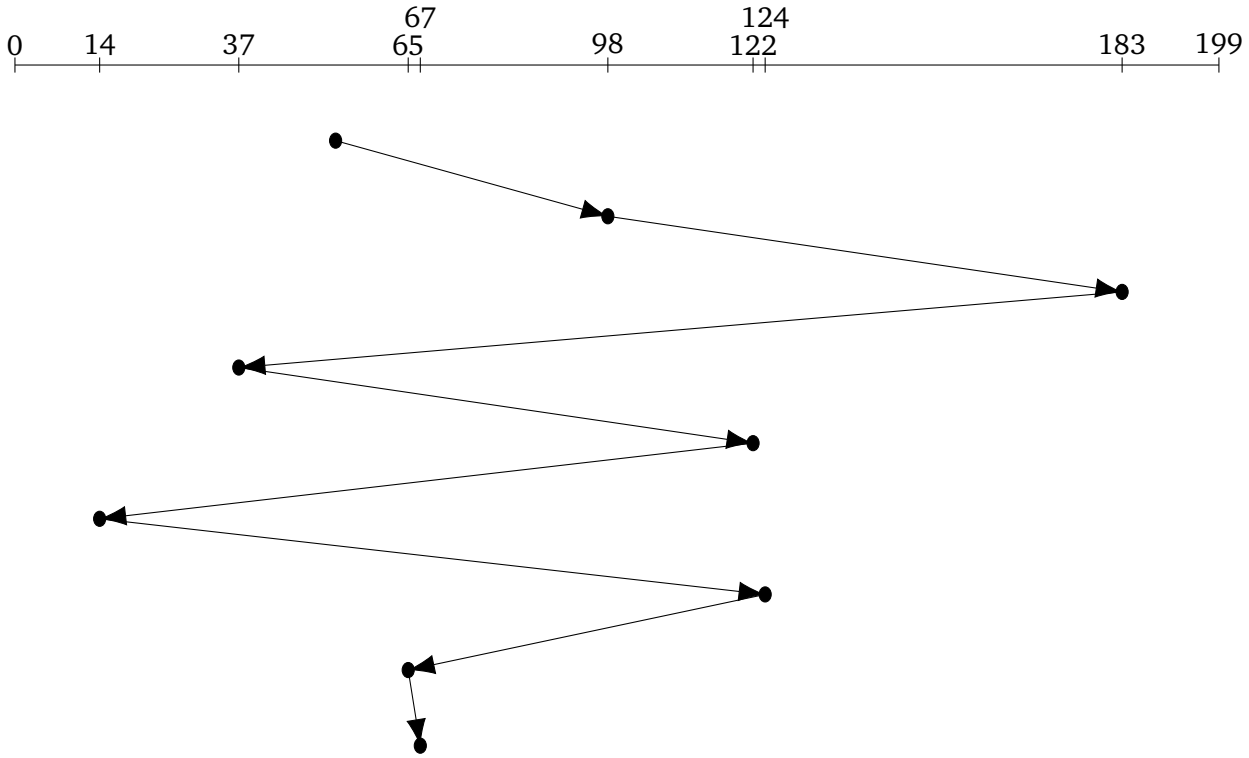

Figure 9.4: FS Scheduling: FCFS

Shortest Seek Time First (SSTF)

Principle: Serve the request the minimum expected seek time from the current head position. Similar to SJF scheduling, this has the problem of starvation.

Example Assume incoming requests for a disk with sectors 0 to 199 and the request queue

$$98, 183, 37, 122, 14, 124, 65, 67$$

and with the head starting at cylinder 53.

236 cylinders of disk movement:

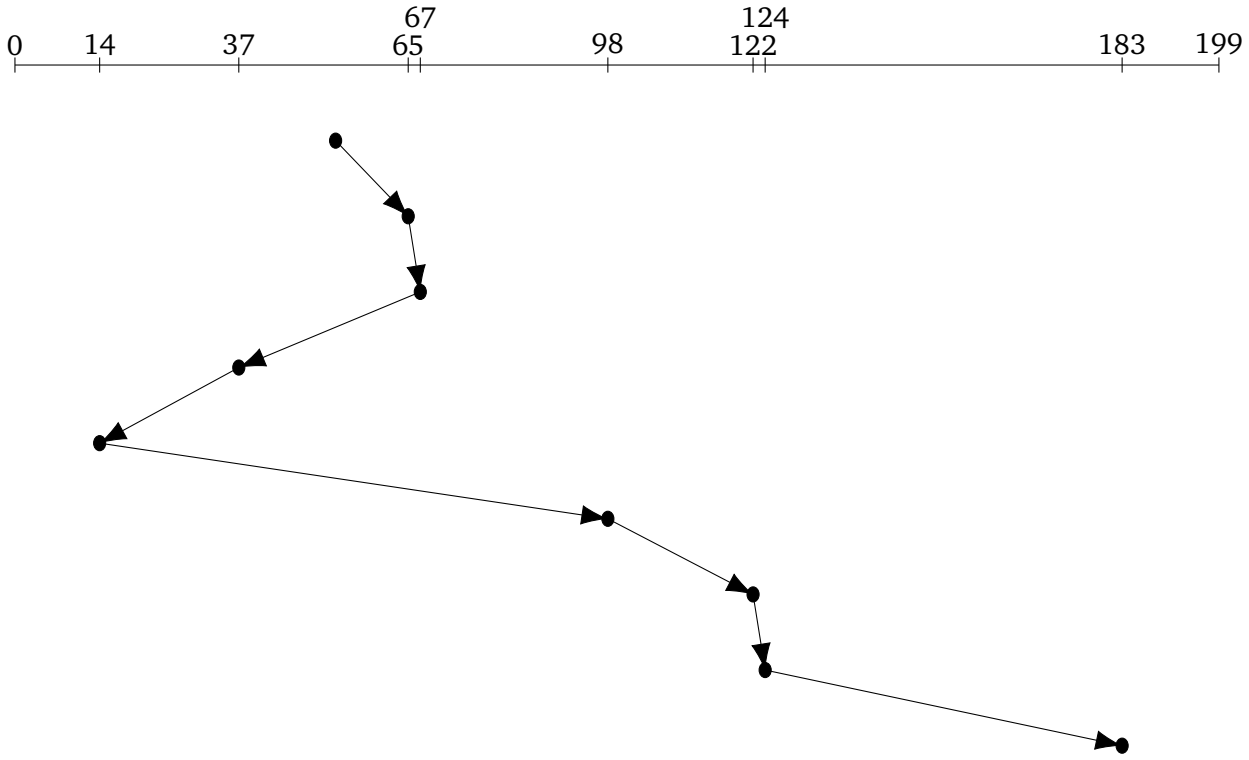

Figure 9.5: FS Scheduling: SSTF

SCAN or Elevator Algorithm (SCAN)

Principle: The disk arm starts at one end of the disk and moves to the start/end continuously, servicing all requests on the way.

Example Assume incoming requests for a disk with sectors 0 to 199 and the request queue

$$98, 183, 37, 122, 14, 124, 65, 67$$

and with the head starting at cylinder 53.

236 cylinders of disk movement:

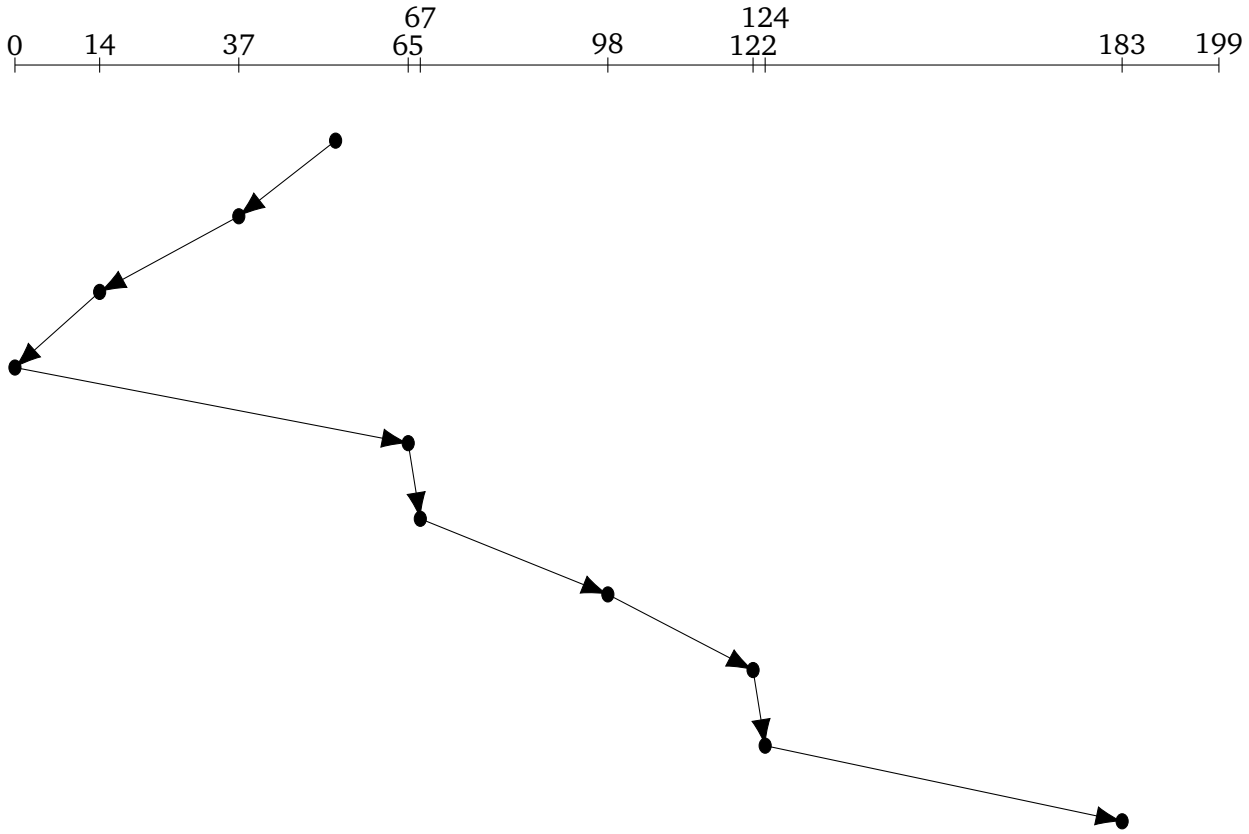

Figure 9.6: FS Scheduling: SCAN

Circular SCAN (C-SCAN)

Principle: Lice SCAN, but not serving requests on the way back.

Example Assume incoming requests for a disk with sectors 0 to 199 and the request queue

$$98, 183, 37, 122, 14, 124, 65, 67$$

and with the head starting at cylinder 53.

382 cylinders of disk movement:

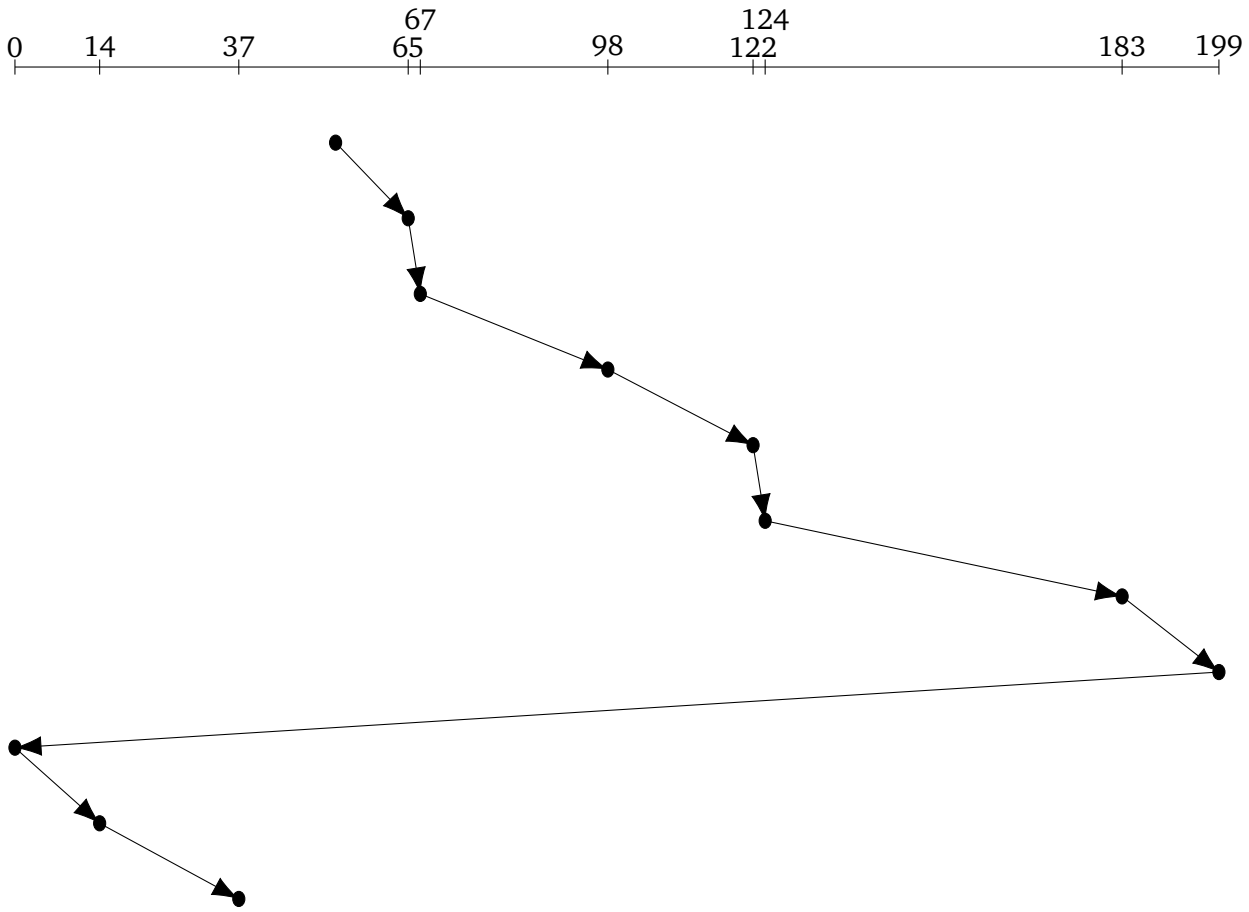

Figure 9.7: FS Scheduling: C-SCAN

SCAN and C-SCAN Variants (LOOK and C-LOOK)

Principle: Like SCAN/C-SCAN, but the arm reverses its direction directly and is not moving all the way to the end/start.

Example (LOOK) Assume incoming requests for a disk with sectors 0 to 199 and the request queue

$$98, 183, 37, 122, 14, 124, 65, 67$$

and with the head starting at cylinder 53.

x cylinders of disk movement:

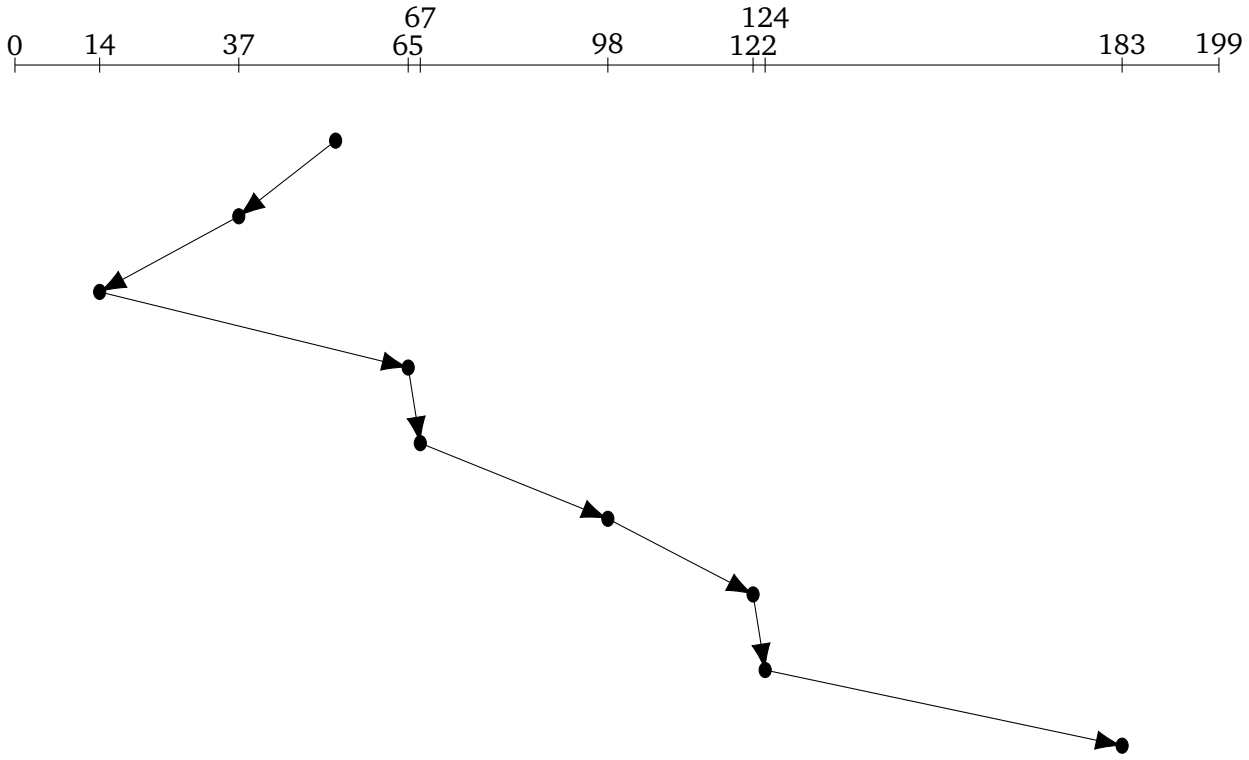

Figure 9.8: FS Scheduling: LOOK

Example (C-LOOK) Assume incoming requests for a disk with sectors 0 to 199 and the request queue

$$98, 183, 37, 122, 14, 124, 65, 67$$

and with the head starting at cylinder 53.

322 cylinders of disk movement:

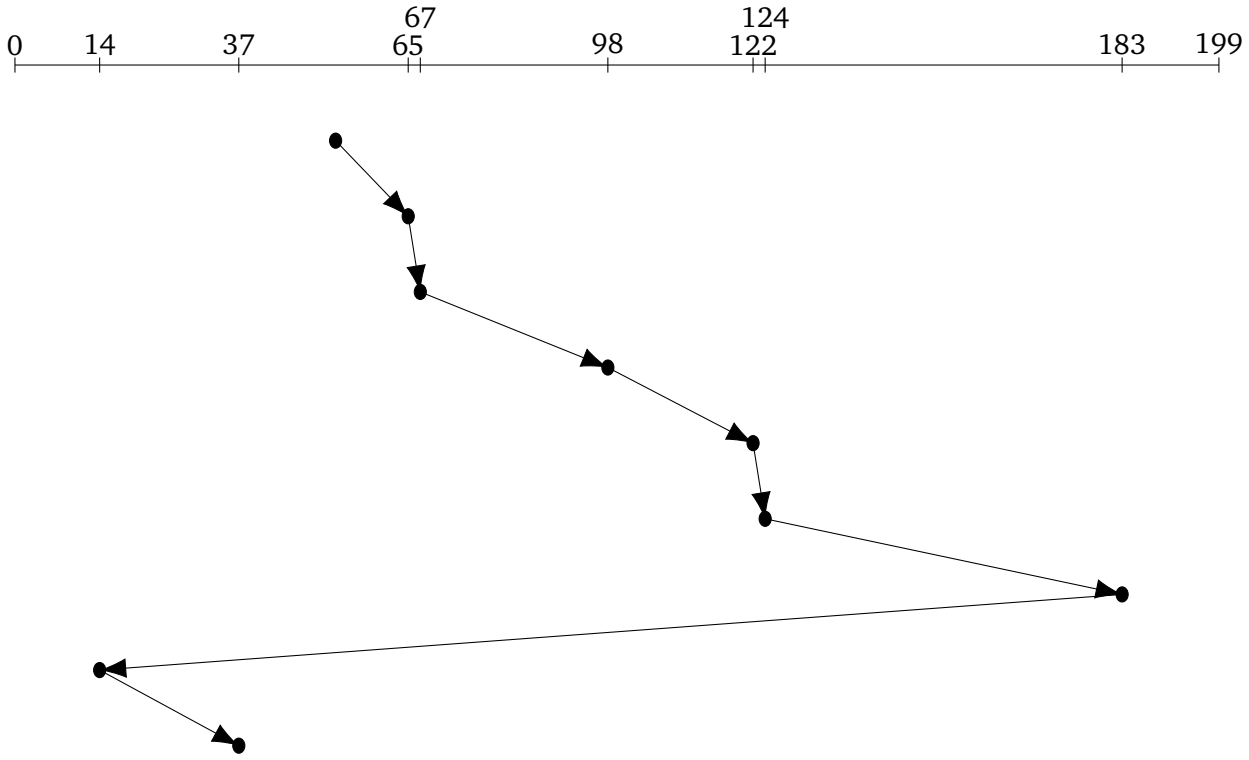

Figure 9.9: FS Scheduling: C-LOOK

10 Visualization

- *Virtual machines* (VMs) are hardware emulations to simulate a second machine and install a separate OS on it.
- Reasons to do this is separation, security and to reduce the cost (only one hardware server is needed in most cases and the rest can be virtualized).
- This also enabled cross-platform development (e.g. Android/iOS Apps) using a single hardware machine. Also for OS development!
- Because of all this pros, nearly everything is now virtualized.

10.1 Visualization Types

- **Type 1:** Install the hypervisor and *VM monitor* (VMM) directly on the hardware.
  Examples:
  - Xen
  - ESXi
  - Hyper-V
- **Type 2:** Install the hypervisor and VMM on a host OS that is installed on the hardware.
  Examples:
  - VirtualBox
  - QEMU
- **OS Level:** Partition the hardware virtually and run the VMs (or *containers*) on the host OS.
  Examples:
  - lxc
  - Docker
  - BSD jails
  - Solaris zones

10.2 VM Implementation Requirements

- Multiplexing is required for the CPU, memory, I/O, etc.
- The VM implementations have to maintain:
  - **Safety:** No VM can affect the correctness of another VM.

- **Liveness:** No VM can prevent another VM from making progress.
- **Confidentiality:** No VM can infer sensitive information of another VM. This is an extremely hard problem and is a critical task.

10.2.1 Threats to Safety/Liveness

Because of the way inter-OS memory management and I/O works, the OS cannot natively run next to the hypervisor.

10.2.2 Types of Instructions

Privileged and sensitive instructions shall only be executed in ring 0 which leads to the definition of *virtualizability* iff all sensitive instructions are privileged $\rightarrow$ x86 is not virtualizable.

10.3 Trap and Emulate

If all sensitive instructions were privileged, then:

1. Start hypervisor in ring 0.
2. Register protection fault handler.
3. Start OS in ring 1.
4. If a sensitive instruction occurs $\rightarrow$ protection fault (*TRAP*).
5. Switches to the handler routine of the hypervisor.
6. Do what the OS wants in a way that it does not violate safety or liveness (*EMULATE*).

Problem: Trap and Emulate is also used by the OS.
If not all sensitive instructions trap (x86), there are three options:

- Option 1: Paravirtualization
- Option 2: Binary rewriting
- Option 3: Hardware extensions (Intel VT-x, AMD SVM)

10.3.1 Option 1: Paravirtualization

- Ask the OS developers to take care of:
  - The OS shall not use sensitive instructions.
  - Replace all sensitive instructions by explicit traps to the hypervisor.
- Problems: Large code basis and it has to be done for each different hypervisor.

10.3.2 Option 2: Binary rewriting

- Perform the following steps for each basic block (BB) of the OS machine code:
  1. Scan for sensitive instructions.
  2. Replace sensitive instructions by TRAP instructions.
  3. Execute modified BB.
  4. Set and prepare the next BB according to the last instruction of the current BB.
- No need to rewrite the user mode code, but there is still some overhead.

10.3.3 Option 3: Hardware extensions (Intel VT-x, AMD SVM)

1. Start the hypervisor in ring 0.
2. Register *sensitive instruction handlers*.
   - Specify on which instructions the processors traps.
   - Register the corresponding handlers.
3. Start the OS in ring 1.
4. If a sensitive instruction occurs $\rightarrow$ protection fault (TRAP).
5. Switch to the handler routine of the hypervisor.
6. Do what the OS wants in a way that it does not violate safety or liveness (EMULATE).

10.4 Memory

10.4.1 Shadow Page Tables

Principle:

1. Start hypervisor.
2. Register privileged and sensitive instruction handlers.
3. Start OS.
4. OS sets register pointing to the top level page table $\rightarrow$ TRAP (privileged).
5. Mark the top level PT as read-only and resume.
6. OS modifies top level PT to add PT entries $\rightarrow$ TRAP (memory access violation).
7. If new lower PT are added, mark them as read-only and resume.

Problems:

- Frequent page fault handling.
- Expensive operations (VMM execution leads to a context switch).

10.4.2 Extended Page Tables

Principle:

- Allocate machine memory and simulate "physical memory" on it.
- The hypervisor holds an *extended page table* (EPT) that maps PFN onto MFN.
- The OS then simply holds its own table mapping VPN onto PFN.

Problems:

- All of the machine memory is already allocated.
- If a guest needs more memory, there is a problem.

Solution: Just do not use more memory (you'll get a cake[1] if you can handle this). Swapping is also an alternative. Ballooning is a solution.

Ballooning

- Install a VMM-controller driver in the guest OS (paravirtualization).
- The driver can then inflate (allocate memory) and deflate (free allocated memory) the memory.
- This way, the memory pressure can be controlled from withing the guest forcing the guest to swap.
- The VMM knows that balloon memory will not be accessed, so no physical to machine memory mapping is used.

10.5 I/O

- Port I/O works as the instructions are privileged and trap and emulate can be used.
- Memory mapping is more complicated as EPT cannot be used without extra safety measures as it shadows the page tables for the I/O memory regions.
- In general, I/O needs emulation (in cannot be just passed through, although this is possible in some cases).

10.5.1 Interrupts

- Normal interrupt handling works, but doubling the latency until interrupts are enabled again is the best case and most of the time it is even worse.
  - The necessary context switch is overhead.
- Alternative: Allow the VMM to handle the basics (re-enable interrupts) and then pass the interrupts to the OS.
  - This causes two "VM exists", which are massive overheads because a full context switch is executed.

[1] The cake is a lie.

  - But the latency until interrupts are enabled again is less.

- Solutions:
  - **Interrupt coalescing**
    - Collect interrupts at VMM for some amount of time.
    - The deliver all the interrupts to the OS as a batch process.
    - Also receive the ACKs as a batch process.
  - **SR-IOV**
    - Virtualizable devices.
    - Replicated control and data registers and buffers.
    - Grant the guest the access to the device directly.
  - **Para-Virtualization** (e.g. virtio)

11 OS Security

See also Chapter 7 of the summary of “Computersystemsicherheit” at https://dmken.com/cs (German).

- Since Multics, there are multiple users on the same system at the same time.
- The OS has to provide safety (prevent accidental conflicts) and security (prevent intentionally harmful actions).
- This chapter covers the following CIA-security-properties:

  Confidentiality No (unauthorized) access to private data.

  Integrity No (unauthorized) modification of private data.

  Availability No (illegitimate) prevention of access to shared resources.
- Computers rarely work as expected (hardware bugs, software bugs):

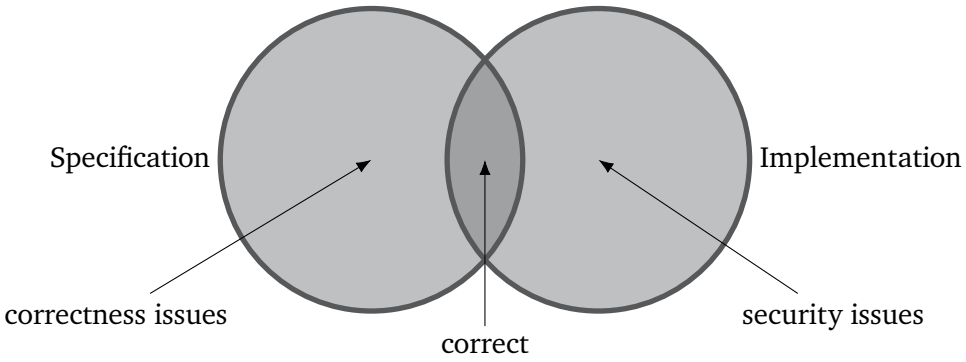

Figure 11.1: Software Bugs

11.1 Bell-LaPadula Model

- Each element (subject, object) of the system is assigned a number (clearance, classification).
- A higher number means that it is more critical.
- The model faces the confidentiality of the data.
- **No-Write-Down:** A high clearance subject must not be able to write less classified objects.
- **No-Read-Up:** A low clearance subject must not be able to read higher classified objects.

11.2 Biba Model

- Much like the Bell-LaPadula Model, but a lesser number means that it is more critical.
- The model faces the integrity of the data.
- **No-Write-Up:** A low clearance subject must not be able to write higher classified objects.
- **No-Read-Down:** A high clearance subject must not be able to read less classified objects.

11.3 Multiuser System Model

- On a multiuser system, CPU, memory and I/O are shared.
- The CIA-properties are required
  - between processes,
  - of the OS in case of malicious processes and
  - of the processes in case of malicious OS.

11.4 Inter-Process-Protection

- Confidentiality
  - Registers: per process
  - Memory: address space isolation
  - Files: somehow access control
- Integrity
  - Registers: per process
  - Memory: address space isolation
  - Files: somehow access control
- Availability
  - CPU: scheduler design and preemption
  - Critical sections (especially the “progress” condition of ME)
  - Memory: Virtual $\rightarrow$ physical mapping and swapping
  - I/O devices: assignment and revocation by the OS kernel

11.4.1 Access Control Lists (ACLs), UID and GID

- In Linux, the ACLs are simple and straight forward.
- Each file has an owner, called *User ID* (UID) and a group, called *Group ID* (GID).

- There is also the subject *others* which includes all users that are neither the owner nor a member of the group.
- Every subject (user, group and others (u, g, o)) gets assigned access rights of the following set:
  - **`r`** The subject is allowed to *read*.
  - **`w`** The subject is allowed to *write*.
  - **`x`** The subject is allowed to *execute*.
- Normally, the permissions are written like `uuugggooo`, whereas the first three letters define the permissions of the user, the second the permissions of the group and the latter the permissions of others. Im a permission is not given, it is indicated with a dash (`-`).
- Examples:
  - `rwxr-xr-x` The owner is allowed to read, write, execute; group members and others are only allowed to read and execute.
  - `rwxrwxrwx` Everyone is allowed to do anything.
  - `rw-r--r--` The owner is allowed to read, write; group members and other are only allowed to read.

11.4.2 The Superuser

- The *superuser* (named *root* on most *NIX systems) is allowed to do anything.
- Any permitted user (usually the administrator) can switch to "root mode" by issuing `sudo` before the actual command.
- Example: Consider a file `foo` with permissions `---------`. It is not accessible by anyone using something like `cat foo`. When issuing `sudo cat foo`, it works and the content is printed.
- One can also become root itself by issuing `sudo -i`.
- Notice: Not every user can execute `sudo`, the have to be allowed to do so by the configuration file `/etc/sudoers`.

11.4.3 Alternative to ACLs: Capabilities

- Used in L4 microkernels.
- A *capability* is a token that grant a privilege, e.g. replacing the scheduler policy or accessing a file (descriptor).
- It can be passed to child processes.
- Notice: These are not POSIX capabilities, they are a completely different thing, e.g. to fine grain the superuser rights.

11.5 Protection from Processes

- Confidentiality and Integrity
  - The kernel shares the address space of processes.
  - The kernel pages are protected and can only be accessed in privileged execution mode.
- Availability
  - Interrupt handlers in the OS.
  - A timer ensures regular interrupts.
  - Disable interrupts is a privileged instruction.
  - All resources of the system are accessible to the OS.

11.6 Hardware Bug: Meltdown

1. Attach to the speculative/out-of-order execution (exploit processor pipelining).
   - The speculative execution loads stuff into memory.
   - The loaded memory is then put into the data cache.
   - This data cache can be exploited to read unauthorized data.
2. Caching side channel attack.

11.6.1 Mitigation: Kernel Page Table Isolation (KPTI)

- Hardware bugs need hardware fixes and cannot be fixed by simple updates.
- Alternative: Stop relying on unsafe features in the software (which makes it less performant).
- *Kernel space table isolation* (KPTI) divides the memory into to tables, one for the kernel and one for the user space.
- This way, exploited speculative execution cannot read kernel data.
- But a lot of context switches have to happen which are slow. Depending on the type of used software, this can cause an overhead up to 30%, but is usually around 5%.

11.7 Exploitable Software Bugs

11.7.1 Buffer Overflow

- Variable bounds within the heap/stack are not enforced.
- A malicious program can escape from its desired address space (e.g. by *smashing the stack*).
- Mitigation: Ad a *canary* when pushing data an the stack and verify its value when reading from the stack. If the value is not the same anymore, halt.

11.7.2 String Format Vulnerabilities

```c
#include  <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
    char buf[100];
    int x = 1;
    snprintf(buf, sizeof buf, argv[1], argv[2]);
    buf[sizeof buf - 1] = 0;
    printf("Buffer size is: (%d)\nData input: %s\n", strlen(buf), buf);
    printf("X equals: %d; in hex: %#x\nMemory address for x: (%p)\n", x, x, &x);
    return 0;
}
```

Figure 11.2: Security: String Formatting

- When using some input like `Bob`, everything is fine.
- But using `Bob %x % %x %x` prints "Bob" but also some additional addresses that where stored behind the string.
- That way, the attacker could figure out the memory and potentially read unauthorized data.
- Fix: Do not just put user-generated content into the format string of `printf` or similar function.

11.7.3 Return-Oriented Programming (ROP)

- The return address of a program is typically stored beneath the stack.
- By writing out of the stack, the return instruction can be overwritten and the attacker can cause that the program returns somewhere else.
- This way, the attack can execute any code he wants as he can inject code into the stack and then jump to it.
- Mitigation: Use *(kernel) address space layout randomization* ((K)ASLR). This makes it harder for the attacker, but not impossible.