OS Zusammenfassung

Grégoire Mercier

March 3, 2019

Abstract

This lecture notes are for me to learn LaTeX and review the operating system lecture

1 Introduction

1.1 Overview

The Operating System

- **Provides abstraction layer**
  Manages and hides hardware details
  Low-level interfaces to access hardware
  Multiplexes hardware to multiple programs
  makes hardware use effivient for applications
- **Provides protection** from
  users/processes using up all resources
  processes writing into other processes memory
- **is a ressource manager**
  Manages and multiplexes hardware ressources
  Decides between conflicting requests for resource use
  Strives for efficient and fair resource use
- **is a control program**
  Controls execution of programs
  Prevents errors and improper use of the computer

There are no universally accepted definitions

1.2 Hardware

CPU (Central Processing Unit)

- Fetches instructions from memory and executes them
- Internal registers store data and metadata during execution
- **User Mode (x86: "Ring 3" or CPL3)**
  Only non-privileged instructions, no hardware managment in this mode for protection
- **Kernel Mode (x86: "Ring 0" or CPL0)**
  All instruction allowed, including privileged instructions

RAM (Random Access Memory) keeps currently execting instructions and data

Caching

- Ram delivers instruction/data slower than the CPU can execute
- Memory references typically follow principle of locality
- **Caching** helps mitigating this **Memory Wall**
  Informations in use are copyed from slower to faster storage. When needed, check whether it is in faster storage before going down in the Memory Hierarchy, then copy it to cache to be used from there

Acces times

• CPU registers	1 CPU cycle
• L1 cache per core	4 CPU cycles
• L2 cache per pair of cores	12 CPU cycles
• L3 cache	28 CPU cycles (25 GiB/s) ‘
• DDR3-Ram	28 CPU cycle for LLC + 50ns (12 GiB/s)

CPU Cache Organization

- Caches divided up into cache lines (often 64 bytes each)
- Separation of data and instructions in faster caches
- **Cache hit**: Data already in cache
- **Cache miss**: Data has to be fetched from lower level first
- Types of Cache misses
  - **Compulsory Miss**: first reference miss, data has never been accessed
  - **Capacity Miss**: cache not large enough for Working Set of process
  - **Conflict Miss**: cache still has space, but collision due to placement strategy

Device Control

- Device controller accepts command from the OS via device driver
- Control by writing into device register and read status by reading it
- Data transfer by writing/reading device memory
- Port-mapped I/O (PMIO) special CPU instructions to access port-mapped registers and memory
- Memory-mapped I/O (MMIO) same address space for Ram and device memory

Devices can signal the CPU through interrupts

1.3 OS Invocation

Operating System Kernel does **not** always run in the background
Tree occasions invoke the Kernel and switch to kernel-mode

• System calls	User-mode process requires higher privileges
• Interrupts	CPU-external device sends a signal
• Exceptions	CPU signals an unexpected condition

System Calls
The main Idea behind System calls is the nessecity to protect processes from one another. So processes are running in User-Mode. The OS provides services, which the applications can invoke in System Calls/syscalls, in order to get the action performed by the OS, on behalf of application
Syscall interface between applicateions and OS provides a limited number of well-defined entry points to the kernel
Application Program Interfaces (API) brings another level of abstraction between applications and Programmers (API invokes Syscalls invokes Kernel-Mode operations)
One single entry point to the kernel for all System calls, the **trap**. Trap switches CPU to kernel mde and enters the kernal in the same, predefined way for every syscall. The system call dispatcher in the kernel acts as a multiplexer for all syscalls.
syscalls identifyed by a number, passed as parameter, **system call table** maps **system call number** to kernel funktion, dispatcher decides where to jump based on the number and table.
Programs have the System call number compiled in! Never reuse old numbers in future versions of kernel

Intrrupts
Interrupts are used by devices to signal predefined conditions to the OS, they are managed by the Programmable Interrupt Controller. When masked the interrupts are only delivered, when unmasked.
interrupt vector: table pinned in memory containing the adresses of all service routines **interrupt service routine**: takes the control in order to handle a specific interrupt. Saves the state of the interrupted process

- Instruction pointer
- Stack pointer
- Status word

Exceptions

- Generated by the CPU itself, if an unusual condition makes it impossiible to continue processing
- CPU interrupts program and relegate control to the kernel
- Kernel determines the reason for exceptions
- If kernel can resolve the problem, it does so and continue the **faulting instruction**
- Otherwise process get killed

Interrupts can happen in **any** context, Exceptions always occur **synchronous to** and **in the context** of a process.

2 OS Concepts

Early on, programs were load directly into **physical memory**. If the program was too large, the programmer had to manually partition his program into **overlays**. OS could swap between disk and memory.
Problems: Buggy programs trash other programs, malicious jobs can read other program's operations, Jobs can take all memory for themself,...
adress spaces: every job has his own address space, so they can't reach other jobs adresses. Jobs only use virtual adresses.

MMU (memory management unit): translates virtual address (vaddr) to physical address (paddr)

- allows kernel-only virtual addresses
- can enforce read-only virtual addresses
- can enforce execute disable

Not all addresses need to be mapped at all times. If a virtual address is not mapped, the MMU throws a **page fault** exception. Handled by loading the faulting address and then continuing the program.
over-commitment: more memory than physically aviable. Page faults also issued by MMU on illegal memory access.
A **process** is a progam in execution, associated with a process control block (PCB) and with a virtual **address space (AS)**. AS is the only memory a program can name and starts at 0 for every program. AS are layed out in sections. Memory acces between those sections is illegal and causes a page fault, called **sementation fault**. Segmentation faults results in the process getting killed by the OS.
A section has the following layout:

- Stack: Function history and local variables
- Data: Constatnts, static variables, global variables, strings
- Text: Program code

Threads represents execution states of a program

- Instruction pointer (IP) register stores currently executed instruction
- Stack pointer (SP) register stores the address of the top of the stack
- Program status word (PSW) contains flags about executeion history
- ...

Two things to consider when designing an OS:
Mechanism: Implementation of what is done
Policy: The rules which decide when what is done and how much

Operating System need to handle multiple processes and threads in order to provide multi-tasking. The **scheduler** decides which job to run next, while the **dispatcher** performs the task-switching. Schedulers provide fairness while trying to reach goals after setted priorities.

Persistent Data is for users is stored in flies and directories. A file is associated with file name and offset with bytes. Directories associate directory names with eigher directory names or file names.
The **file system** is an ordered collection of blocks, what can be operated on by programmers, with operations like open, read, seek, ...
Processes communicate directly through a special **named pipe** file

Directories form a **directory tree/file hierarchy**. The **root directory** is the topmost directory of a directory tree. Files can be accessed by their **path name**.

OS abstract the view of information storage to file systems. Drivers hide specific hardware device. OS increases the performance of I/O devices by

- **Buffering**: Store data temporarily while it is being transferred
- **Caching**: Store parts of data in faster storage for performance
- **Spooling**: Overlap of output of one job with input of other jobs

3 Processes

3.1 Process Abstraction

Multiprogramming is the art of switching quickly between processes. Every process is processed in his own "virtual CPU". When switching processes, the execution context changes. On a **context switch**, the dispatcher saves the current register and memory mappings and restores those of the next process.
A program is a policy, the process is a mechanism.
With n processes with a process spending p of his time waiting for I/O to complete, then CPU utilization = 1 - p^n.

Concurrency: Multiple processes on the same CPU
Parallelism: Processes truly rnning at the same time with multiple CPUs

4 Address Spaces

Programs can see more memory than aviable (80/20 rule: 80% of the process memory idle, 20% active working set). Keep working set in RAM, rest on disk.
address space layout: Organization of code, data and state within process Data can be **fixed sized**, **free'd in reverse order of allocation** or **allocated and free'd dynamically**
The **loader** determines based on an executable file how an executed program is placed in memory

Fixed-size Data and Code Segments

- Data in a program, what has static size, allocated when process created
- BSS Segment (Block Started by Symbol) contains statically-allocated variables and not initialized variabels. The executable file contains the starting address and size of BSS, the entire segment is initially zero
- Data segment contains fixed-size, initialized data elements such as global variables
- Read-only data segment contains constant numbers and strings
- Sometimes BSS, data, and read-only data segment are summarized as a single data segment

Stack Segment
Data is naturally free'd in reverse order of allocation. Fixed starting point of segment, store top at latest allocation SP (stack pointer). In current CPU, the stack segment typically grows downwards!

Heap Segment Some data needs to be allocated and free'd dynamically "at random", such as input/output, size of edited text, ...
Allocate memory in two tiers:
1. Allocate large chunk of memory (heap segment) fom OS, like stack allocation; base address + break

pointer (BRK), process can change size by setting BRK
2. Dynamically partition large chunk into smaller allocation dynamically, with *malloc* and *free*. This happens purely in user space!

4.1 Typical Process Address Space Layout

- **OS** Adresses where the kernel is mapped 0xFFFFFFFF
- **Stack** Local variables, function call parameters, return addresses
- **Heap** Dynamically allocated data (malloc)
- **BSS** Uninitialized local variables dclared as static
- **Data** Initialized data, global variables
- **RO-Data** Read-only data, strings
- **Text** Program, machine code

5 Threads

In traditional OS, each process has it's own address space, set of allocated resources and one thread of execution.
Modern OS handle the processes and treads of execution more flexibly. Processe provide the abstraction of an AS and address resources, while threads provide the abstraction for execution state of that AS/container. /par Why using multiple Threads?
So programs can handle many tasks at once. Without threads, some of the tasks could block each other. Many sequential threads are more easy to handle. It depents on what is to be done in order to choose between threads and processes. If processes share data, they do it explizitly. Threads allow multiple tasks at once in a single process.

5.1 Thread Libraries

Provide an API for creating and managing threads. Pthreads is the POSIX API for creation and synchronization. It specifies behaviour of the thread library.
Each **Pthread** is associated with an identifier (Thread ID(TID)), a set of registers (including IP and SP) and a stack area holding the execution state of that thread.

- Pthread_create Create a new thread, passing pointer to pthread_t (holding TID after successful call), attributes, start funcition and arguments, returning 0 on success or error value.
- Pthread_exit Terminate the calling thread, passing exit code, freeing ressources
- Pthread_join Wait for a specific thread to exit, passing pthread_t to wait for (or -1 for any thread), pointer to pointer for exit code, returning 0 on success, otherwise error value
- Pthread_yield Release the CPU to let another thread run

Multithreaded programming is challenging, because there is more shared state than with processes, so more can possibly go wrong. Programmer needs to care about dividing, ordering, and balancing activities, dividing data and synchronize access to shared data.
Processes group resources, threads encapsulate execution. There is a need to differentiate between

- **Process Control Block (PCB)**: Information needed to implement processes eg. Adress space, open file, child processes, pending alarms.
  The PCB is always known to the OS
- **Thread control Block (TCB)**: Per thread data, eg IP, Registers, Stack, state. Depending on thread model the OS knows about threads or not.

5.2 Thread Model Overviev

OS always knows of at least one thread per process. Threads that are known to the OS are called kernel threads. Threads that are known to the process are called user threads.

- **Many-to-One Model**/Threads fully implemented in user-space: Kernel knows only knows one of possibly multiple threads, user threads are called **User Level Threads(ULT)**
- **One-to-One Model**/Kernel fully aware of and responsible for managing threads: Each user thread maps to a kernel thread, user threads are called **Kernel Level Thread(KLT)**
- **M-to-N Model**:Kernel knows some threads per process, but others are known only to the process, flexible mappint of user threads to less kernel threads. Known as hybrid thread model.

Many-to-One Model: User Level Threads (ULT)
The kernel only manages the process, multiple threads are unknown to the kernel. That allows faster thread management operations, a more flexible scheduling policy, fewer system resources and cen be even used when OS does not support threads. But there is no parallel execution possible and if only one thread blocks, the entire process blocks. Also it is needed to reimlement parts of the OS. Linux defines some acitons as followed: **mkcontext_t** and **ucontext_t** to keep thread state, *makecontext* (initialize a new context), *getcontext* (store currently active context), *setcontext* (replace current context with different one), *swapcontext* (user-level context switching between threads). Periodic threadswitching can be implemented using a **SIGALRM** exception handler.
Address Space Layout: The "main" part of the **Stack** is known to OS and is used by thread library. The own execution state for every thread is allocated dynamically on the heap, using *malloc*. There is possibly an own stack for each exception handler. Concurrent **heap** possible.

One-to-One Model: Kernel Threads (KLT)
The kernel knows and manages every thread, making real paralellism and individual thread block possible. On the downside, the OS manages every thread in the system, syscalls are needed for thread management and scheduling is fixed in OS.
Address Space Layout: There is an own execution state (≡**stack**). Possibly own stack for each exception handler. Parallel **heap** use is possible, but not all heaps are thread-safe.
Implementation and issues: all thread management data is stored in kernel, management funcions provided as syscalls. Signals are used in UNIX to notify a process that a particular event has occured. The signal handler can run on the process stack, on a stack dedicated to a specific signal handler or a a stack dedicated to all signals.

M-to-M Model: Hybrid Threads
M ULTs are mapped to (at most) N KLTs, using pros of ULT and KLT: non-blocking with quick management. Provides flexible scheduling policiy and efficient execution, but is hard to implement and to debug.
Implementation: Kernel is not involved in thread activities such as *create* and *join*. Reached by mapping multiple ULTs on each KLT, so when a ULT blocks, the user-space run-time system run a different ULT without switching to the kernel. **Upcalls**: Kernel notices, that a thread will block and sends a signal to the process. Upcall notifies the process of the thread id and event that happened. Exception handler of the process shedule a different thread in that process. Kernel later informs the process that the blocking event has finished via antother upcall.

6 Inter Process Communication (IPC)

Processes and Threads need to communicate with one another frequently. Process cooperate to share information, speed-up computing and provide modularity. IPC allows exchanging data between those processes, buy **Message passing** (explizitly send and recieve information using system calls) and **shared memory** (multiple processes using same memory regions).

6.1 Message Passing

Mechanism for process to communicate and synchronize their actions, providing operations to *send* and *recieve*. Implemented by using hardware bus, shared memory, kernel memory and the network interface card.

Direct vs. Indirect Messages
direct messages: processes name each other explicitly when exchanging, by *send(P, message)* (send a message to process P) and *recieve(Q, message)* (recieve a message form process Q).
indirect messages: can be sent and recieved from mailoxes. Each mailbox has a unique id. The first communicating process creates mailbox, last destroys mailbox. Process can only communicate if they share a mailbox..

Sender/Reciever Synchronization

- Message passing may be either **blocking** or **non-blocking**
- Blocking is considered **synchronous**
- Non-blocking is considered **asynchronous**
- Depending on buffering scheme, non-blocking sender can communicate with non-blocking reciever

6.2 Buffering

Messages are **queued** using different capacities while they are in-flight

- Zero capacity - 0 messages/no queuing: Sender must wait for reciever (**rendezvous**),message is transferred as soon as reciever becomes aviable (no latency/no jitter)
- Bounded capacity - finite number and length of messages: Sender can send before reciever waits for messages, sender can send while reciever still processes previous messages, sender musst wait if link full
- Unbounded capacity: Sender never waits, memory may overflow, potentially causing very large latency between send and recieve

6.3 Shared Memory

Communicate through a region of shared memory. Processes have shared regions in one another AS. Threads "naturally" share address space. The semantics are application-specific.
Tricky to get safety and high performance, especially if many processes and many CPUs are involved, due to **cache coherency protocol**, especially if there are multiple writers, due to race conditions.

Sequential consistency (SC) " The result of execution is as if all operations were executed in some sequential order, and the operations of each processor occurred in the order specified by the program", means that all memory operations occure one at a time in program order, ensuring write atomicity.
CPU and compiler re-order instructions **execution order** for more efficient execution. Without SC, multiple processes on multiple cores behave "worse" than preemptive threads on a single core. They may give diffeerent results than when interleaving on one core.
Problems:

- Modern CPUs are generally not sequentially consistent, because it would complicate write buffers, complicate non-blocking reades and make cache coherence more expensive
- Compilers don not generate code in program order, they re-arrange loops for better performance, eliminate common subexpressions and cares about software pipelining

- As long as a single thread accesses a memory location at a time, this is not a problem

DON'T try to access the same memory location with multiple threads at the same time withouht proper synchroniyation!

7 Synchronization

Race Conditions (assuming to have sequential memory consistency)
Occures when two or more non-atomic instruction sequences operates at one time and may end up with a wrong result, because they were not done in the proper sequence. Even operation such as *add count 1* may create race conditions. Only **interlocked operations** are safe (that implicates that there is only a single interlocked operation for the problem). Interlocked operations are more expensive than regular operations.
General solution for the **critical section (CS) problem**: Put non-atomic instruction inside of a critical section.
Desired Properties for Solution to Critical-Section Problem

- **Mutual Exclusion**: At most one thread can be in the CS at any time
- **Progress**: No thread running outside of the CS may block another thread from getting in
- **Bounded Waiting**: Once a thread starts trying to enter the CS, there is a bound on the number of times other threads get in

Disabling Interrupts: While in CS, thread cannot be interrupted, implemented with a "do not interrupt" (DNI) bit. *enter_critical_section()* sets DNI bit, *leave_critical_section* clears DNI bit, so when interrupts disabled, scheduler is never called. That is easy and convenient in the kernel, but only works in single-core systems, and only feasible in kernel, don't want to give user this power.
Approach: **lock variable**: global *lock*, enter CS if lock is 0, set it to 1 when entering, otherwise wait (**buisy waiting**), but that doesn't solve the CS problem, reading and setting lock is still not atomic. Test and set *lock* atomically possible with x86 (*xchg* atomically excaange memory content with a register). Implemented as **spinlock**. Solves **mutual exclusion**, **progress**, but not **bounded waiting**. Also spinlock doesn't work well

- if the lock is **congested** (large CS or many threads trying to enter)
- if threads on different cores use the lock (expensive to keep memory coherency between cores)
- when processes are scheduled with static priorities such as **priority inversion**, causing unexpected behaviour

Nevertheless, spinlocks are widely used, especially in kernels

Semaphore
Idea: busy part of busy waiting is a spinlock limitation. So let threads sleep on locks and wake them up one at a time when lock becomes free.
Introduce two syscalls, operating on integer variables called **semaphore**: *wait(&s)* and *signal(&s)*. *s* is initialized to maximum number of threads that may enter the CS at any given time. A semaphore initialzed to 1 is called **binary semaphore**, **mutex semaphore** or just **mutex**. **counting semaphores** allows more than one thread in the CS.
Implementation Considerations: wait and signal need to be carefully synchronized, otherwise it could result in a race condition between checking and decrementing s. Aditionally, **signal loss** may occure when waking up threads and waiting at the same time. Each semaphore is associated with a wake-up queue: **weak semaphores** wake up a random waiting thread, **strong semaphores** wake up threads in the order in which thez started waiting.
Mutual exclusion, **progress** and **bounded waiting** are solved. But every enter and leave are

syscalls, which are slower than regular function calls.

Fast User Space Mutex (futex)

- Userspace and kernel component
- Try to get into CS wit a userspace spinlock
- If CS buisy, use a syscall to put thread to sleep
- Otherwise enter CS lompletly in userspace

Classic synchronization Problems

Producer-Consumer Problem (bounded-buffer problem): Buffer shared between a producer and a consumer. *int count* keeps track of number of aviable items. Producer produces items and place them into the buffer, incrementing *count*. If buffer full, producer sleeps until consumer consumed an item. Consumer consumes items, removing them from the buffer and decrementing *count*. If buffer empty, consumer has to sleep until producer produced an item. Solved with a mutex and 2 counting semaphores or condition variables (CV), which allow blocking until a condition is met, with following ideas: New operation that performs unlock, sleep, lock atomically, and new wake-up operation that is called with lock held.

Readers-Writers Problem: Many threads compete to read or write the same data, **readers** only read data, do not perform any updates, **writers** can both read and write. It is unnecessary to use a single mutex for read and write, blocking multiple readers while no writer is present. Idea: If no threads writes, multiple readers may be present, if a thread writes, no other readers and writers are allowed

- 1st Readers-Writers Problem: Readers Preference: Writers cannot aquire acces to CS until last reader leaves the section
- 2nd Readers-Writers Problem: Writers Preference: No writers should be waiting longer than absolutely necessary
- 1st and 2nd readers-writers problem have the same issue: Readers preference -¿ writers can starve, writers prefernce -¿ readers can starve
- 3rd Readers-Writers Problem: No threads shall starve. Posix treads contains readers-writers lock to address this issue (*pthread_rwlock*). Multiple readers but only a single writer are let into the CS. If readers are present, while a writer tries to enter the CS, don't let further readers in, block until readers finish, let writer in. Really difficult to imlement!

Dining-Philosopers Problem
Five philosophers are sitting around a table, each one has a plate of spaghetti in front of him, and there is one fork between each one. They can only eat, if they have two forks in their hand. Their cyclic workflow is: 1. Think, 2. Get hungry, 3. Grab one fork, 4. Grab another fork, 5. Eat, 6. Put down forks. No communication allowed, no "atomic" grab of both forks. Problem: What if the all grab their left fork at once. This problem is called **deadlock**. Workarounds: 4 Philosophers allowed at a table of 5 (**deadlock avoidance**), odd philosophers take left fork firt, even philosophers take right fork first (**deadlock prevention**)

7.1 Deadlocks

Deadlocks can arise if all four conditions hold simultaneously:

- Mutual exclusion (Limited access to resource, resource can only be shared with a finite amount of users)

- Hold and wait (wait for next resource while already hold at least one)
- No preemption (once the resource is granted, it cannot be taken away but only handed back voluntarily)
- Circular wait (possiblity of circularity in graph of requests)

Deadlock countermeasures

- Prevention (pro-active, make deadlocks impossible to occur)
- Avoidance (decide on allowed actions based on a-priori knowledge)
- Detection (react after deadlock happened/revovery)

Deadlock Prevention
Negate at least one of the required deadlock conditions:

- Mutual exclusion - buy more resources, split into pieces, virtualize (”infinite” # of instances)
- Hold and wait - get all resources en-block, 2-phase-locking
- No preemption - virtualize to make preemptable
- Circular waiting - ordering of resources, prevent deadlocks with partial order on resources

Deadlock avoidance
On every resource request, decide if system stays in saft state, what needs a-priori information. Using Resource Allocation Graph (RAG).
RAG View system state as graph, processes are round nodes, resources are square nodes. Every instance of a resource is depicted as a dot in the resource node.
Resource requests and assignmeents are edges:

- Resource pointing to process: Resource is assigned to process
- Process pointing to resource: Process is requesting resource
- Process may request resoure: Claim edge, depicted as dotted line

Deadlock Detection Allow system to enter deadlock, detect it, apply recovery scheme.
Maintain **Wait-For Graph(WFG)**. Periodically invoce an algorithm that searches for a cycle in the graph, if there is a cycle, there exists a deadlock.
Recovery: Abort all deadlocked processes/Abort one process at a time until the deadlock cycle is eliminated.

7.2 Implementation

Synchronisation problems occur very often when programming operating systems, the parallelism introduced by multiple processors and the concurrency introduced by multiprogramming needs to be considered carefully when writing an operating system, poorly synchronized code can lead to starvation, priority inversion or deadlocks.

8 Memeory Mamagement Hardware

Main memory and register are only storage that the CPU can access directly. Programs must be brought into memory from background storage and placed within a process' address space for it to be run. Multiple processes can be run concurrently even without memory abstraction (swapping, static relocation).
Swapping
Denotes saving a program's state on background storage (roll-out) and replacing it with another programs's state (roll-in).
Pros: Only needs hardware support to protect the kernel, but not to protect processes from each other.
Contras: Very slow, at every point in time only one process runs (no parallelism).

static relocation
Every Program gets the same address space, resulting in no protection, no dynamic allocation, wanting programs to co-exist peacefully.

Desired properties when sharing physical memory

- Protection
- Transparency
- Resource exhaustion

8.1 Memory-Management Unit (MMU)

Hardware support is needed to achieve safe and secure protection. Hardware device maps virtual to physical address, so the user program deals with virtual addresses.

Base and Limit Registers
Idea: Provide protection and dynnamic relocation in the MMU, introducing special base and limit registers. On every load/store the MMU checks whether the virtual address is larger or equal to *base* and smaller than *base+limit*, then use the virtual address as the physical address in memory.
In order to protect the OS from processes, the main memory is splitted in two partitions, the OS usually held in low memory with interrupt vector, user processes are held in high memory, so OS can acces all process partitions, but MMU denies processes access to OS memory.
Pros: Straigth forwared to implement MMU, and very quick at run-time
Cons: no possiblity to grow address space and no sharing of code and/or data

Segmentation
A possible solution for shortcomings of Base+Limit approach is using multiple base+limit register pairs per process, in order to keep some segments private, and share others.
Now a virutal address consists of a tuple (segment #, offset), so each process has a segment table, mapping virtual address to physical addresses in memory (**base** is the starting physical address where the segment resides in memory, **limit** is the length of the segment, **protection** provides access restriction to make safe sharing possible). The MMU has two registers that identify the current address space, the **Segment-table base register (STBR)** points to the segment table location of the cuffent process, the **Segment-table length register (STLR)** indicates the number of segments used (segment # is legal, if it is < STLR).
Pros: Makes data/code sharing between processes possible without compromising confidentiality and safety/security, Processes doesn't need larch contiguous physical memory area, don't need entire process in memory
Cons: Segment need to be kept contiguous in physical memory, Fragmentation of physical memory.

External Fragmentation
Sum of free memory satisfies requested amount of memory, although contiguous memory would be required. Can be reduced through compaction, means closing gabs by moving allocated memory in one direction, resulting in large free blocks on the other side. Thats only possible if relocation is dynamic and can be done at execution time. Its expensive though and should be avoided.

Paging
Dividing physical memory into fixed-size blocks called **page frames**, whose size are a power of 2 Bytes, typically 4KiB, 2 MiB, 4 MiB, and dividing virtual memory into blocks called **pages**, with the same size aviable as for frames. The OS keeps a **page table**, that stores mappings between **virtual page numbers (vpn)** and **page frame numbers (pfn)** for each AS. It keeps track of all free frames and modifies page tables as needed. So to run a program of n pages, its needed to find n free frames and load the program into them.
A **Present Bit** in the page table indicates if a virtual page is currently mapped to physical memory. The MMU reads the page table and autonomously translates valid mappings. If a process issues an instruction to acess a virtual address that is currentlz not mapped, the MMU calls the OS to bring in the data (**page fault**).
Virtual address is divided into **virtual page number** (Index into the page table which contains base address of each page in physical memory) and **Page offset** (concatenated with base address results in physical address)

Hierarchical Page Table
The complete table is not needed for each AS, most virtual addresses are not used by process. So the virtual addresses are subdivided into further multiple page table indices p_n, forming a hierarchical page table.
Intel x86-64 Page Table Hierarchy as example

Page Table Entry Content

- **Valid Bit**: Whether the page is currently aviable in memory or needs to be brought in by the OS, via a page fault, before accessing it (Present Bit)
- **Page Frame Number**: If the page is present, at which physical address the page is currently located
- **Write Bit**: If the page may be written to. When a process writes to a page with a clear bit, the MMU halts the poeration and raises a page-fault
- **Caching**: If this page should be cached at all and with which policy
- **Accessed Bit**: Set by the MMU if page was touched since the bit was last cleared by the OS
- **Dirty Bit**: Set by the MMU if this page was modified since the bit was last cleared by the OS

The OS performes all operations that require semantic knowledge:

- Page allocation/bringing data into memory
- Page replacement
- Context switch

Internal Fragmentation
Paging elimiates external fragmatation, but internal fragmentation becomes a problem: Memory can only be allocated in fixed page frame size, but an allocated virtual memory will generally not end at a page boundary. The unused rest of the page cannot be used by other allocations and is lost

Linear Inverted Page Table
Following Problem: Large AS (64-bit) but only few virtual addresses are mapped. That results in much memory wasted on page tables in the system, and slow lookup due to many levels of hierarchy. That can be solved by Inverted page table mapping: Map physical frame to virtual page instead of the other way around, what leads to a single page table for all processes, and one page table entry for each physical page frame.
Pros: Less overhead for page table meta date
Cons: Increases time needed to search the table when a page reference occurs

Hashed Inverted Page Table
Hach anchor table is added before the actual page table. It is at least as large as the page table, and maps virtual page numbers and process IDs to page table entries.

8.2 Translation Lookaside Buffer

Note: Naïve paging is slow
Make it faster by adding a cache that stores recent memory translations: **Translation Lookaside Buffer (TLB)** maps (vpn) to (pfn, protection).
On every load/store check if translation is already chached in TLB (**TLB hit**), otherwise walk page tables and insert result into TLB (**TLB muss**). Can compare many TLB entries in parallel in hardware.

TLB Miss
On TLB miss, evict an entry vrom TLB, in order to get space for the looked-up one, and load the an entry for the missing virtual address into the TBL.
TLB can be software-managed or hardware-managed:

- **Software-managed TLB**: OS recieves **TLB miss exception**, decides witch entry to evict from TLB and walks page tables in software to fill new TLB entry. TLB entry format is specified in **instruction set architecture (ISA)**. (E.g. MIPS)
- **Hardware-managed TLB**: Evict a TLB entry based on a policy encoded in hardware without involving the OS. Walke page tables in hardware to resolve address mapping. (E.g. x86-64, ARM)

Address Space Identifiers
Problem: vpn is dependent on AS, vpn in different AS can map to different pfns, so it is required to clear TLB on AS switch.
Add additional identifiers in the TLB to solve this problem. Now the TLB has Address space Identifiers (ASID) in every entry, it maps (vpn, ASID) to (pfn, protection), resulting in less TLB misses.

TLB Reach
Also named **TLB Coverage**: The amount of memory accesible with TLB hits (TLB reach = TLB size * Page size). Ideally the working set of each process is stored in the TLB, otherwise there is a high degree of TLB misses.

Effective Access Time

- Assiocative lookup takes τ time units
- A memory cycle takes μ time units
- TLB hit ratio α (Percentage of all memory accesses whose translation is already cached in)

Effective Acces Time (EAT) for linear page table without cache:
EAT = $(\tau + \mu) \cdot \alpha + (\tau + 2 \cdot \mu) \cdot (1 - \alpha) = \tau + 2 \cdot \mu - \mu \cdot \alpha$

Glossary

. 4

process control block (PCB) Informations about allocated resources of a process. 4

. 4