

OS Zusammenfassung

Grégoire Mercier

February 28, 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 efficient for applications
- **Provides protection** from
users/processes using up all resources
processes writing into other processes memory
- **is a resource manager**
Manages and multiplexes hardware resources
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 management in this mode for protection
- **Kernel Mode (x86: "Ring 0" or CPL0)**
All instruction allowed, including privileged instructions

RAM (Random Access Memory) keeps currently executing 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**
Information in use are copied 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

Access 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 necessity 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 applications 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 mode and enters the kernel in the same, predefined way for every syscall. The system call dispatcher in the kernel acts as a multiplexer for all syscalls.

syscalls identified 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 addresses 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 impossible 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 themselves,...

address spaces: every job has his own address space, so they can't reach other jobs addresses. Jobs only use virtual addresses.

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 available. Page faults also issued by MMU on illegal memory access.

A **process** is a program 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 running at the same time with multiple CPUs

4 Address Spaces

Programs can see more memory than available (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 variables. 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) from 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	Addresses where the kernel is mapped 0xFFFFFFFF
• Stack addresses	Local variables, function call parameters, return
• Heap	Dynamically allocated data (malloc)
• BSS	Uninitialized local variables declared 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 its own address space, set of allocated resources and one thread of execution.

Modern OS handle the processes and threads of execution more flexibly. Processes 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 depends on what is to be done in order to choose between threads and processes. If processes share data, they do it explicitly. 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 function and arguments, returning 0 on success or error value.
- **Pthread_exit** Terminate the calling thread, passing exit code, freeing resources
- **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. Address 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 Overview

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 mapping 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 can 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 reimplement 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 (\equiv **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 occurred. 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 policy 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.

Glossary

. 4

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

. 4