Advanced Operating Systems

Laboratory

Christian Rossi

Academic Year 2024-2025

Abstract

This course provides an overview of key topics related to operating systems, focusing on design patterns, resource management, and peripheral interaction. It begins by introducing the main goals of operating systems, detailing design patterns and mechanisms for mediating and regulating access to system resources. It also categorizes operating systems into different types, such as monolithic, microkernel, hybrid, and uni-kernel, with examples for each.

The section on resource management and concurrency covers support for multi-process and multi-threaded execution, CPU scheduling for both single and multiprocessors, and modern load-balancing techniques for NUMA systems. Topics such as memory consistency, multi-threaded synchronization, advanced locking techniques, lockless programming, and inter-process communication (IPC) primitives are discussed in detail. Additionally, asynchronous programming, deadlock, starvation, virtual memory management, and the basics of software and hardware virtualization, including containers, are explored.

In peripheral and persistence management, the document addresses low-level peripheral access, communication buses like PCI, programmable interrupt controllers, and interrupt management. Topics include character-based and block-based I/O, the development of device drivers, and an overview of modern storage devices (e.g., HDDs, SSDs) and file systems, emphasizing the filesystem-centric view of peripherals.

Lastly, run-time support is discussed, including boot loaders, kernel initialization, device discovery mechanisms (ACPI and device trees), C runtime support, the structure of dynamic libraries, and linking. Tools for the development, analysis, and profiling of embedded code are also reviewed.

Contents

CHAPTER 1

Kernel

1.1 Modules

Modules provide a flexible way to manage the operating system image at runtime. Rather than rebooting the system to load a different operating system image, modules allow the operating system's functionality to be extended dynamically. This approach offers a convenient method for interacting with and modifying the kernel.

1.1.1 Paramters

Parameters for modules can be declared and initialized directly within the code:

```
    static int num = 5;
// S_IRUG0: everyone can read the sysfs entry
    module_param(num, int, S_IRUG0);
```

These parameters can also be specified when the module is instantiated:

```
    insmod yourmodule.ko num=10
```

1.1.2 Kernel crashes

Since your module has full access to kernel code and data, it can potentially compromise the kernel's state and lead to a crash. Common causes include:

- Memory access errors (e.g., NULL pointer dereferencing, out-of-bounds access).
- Explicitly triggering a kernel panic on error detection (using `panic`).
- Incorrect execution mode (e.g., sleeping in atomic context).
- Deadlocks detected by the kernel (e.g., soft lockup or locking issues).

Kernel oops When a kernel oops occurs, the CPU's state at the time of the fault is captured, along with:

- The contents of registers, which might offer clues about the crash.
- A backtrace of function calls that led to the crash.
- Stack content (the last few bytes).

In severe cases, the kernel may panic and halt all execution by stopping the scheduling of applications and entering a busy loop.

1.1.3 Printing messages

Messages can be printed from the kernel to the log buffer using the `pr_ast` family of functions, such as: `pr_emerg`, `pr_alert`, `pr_crit`, `pr_err`, `pr_warn`, `pr_notice`, `pr_info`, `pr_cont`, When printing pointers, consider the following formats:

- `%p` : displays the hashed value of a pointer by default.
- `%px` : always displays the pointer's address (use with caution for non-sensitive addresses).
- `%pK` : displays the hashed pointer value, zeros, or the actual address based on the `kptr_restrict sysctl` setting.

1.2 Kernel debugging

Kernel debugging often requires creative problem-solving rather than a one-size-fits-all approach. There are many possible strategies for arriving at the same solution, and the process usually involves experimentation and iteration.

1.2.1 GNU debugger

Using live GDB can be beneficial in the following cases:

- *Understanding code flow*: tracing how code is executed at runtime.
- *Dumping data structures and assembly*: inspecting kernel data and understanding how assembly relates to the higher-level code.
- *Debugging hangs*: identifying why the system is not responding.

However, there are scenarios where live GDB might not be the best option:

- *Issue is not reproducible*: the bug might not happen consistently, making it hard to trace in real time.
- *Unclear debugging path*: when you're unsure what to search for, live debugging can be cumbersome.

KGDB KGDB allows full control of kernel execution using GDB from another machine over a serial connection, enabled by `CONFIG_KGDB_SERIAL_CONSOLE`. This method supports inserting breakpoints even in interrupt handlers and comes with Python scripts that make debugging easier.

JTAG JTAG is useful for debugging low-level system issues when there's almost no functional software. It is commonly used during system bring-up when it's unclear if the kernel has reached a point where KGDB could be effective.

QEMU With QEMU, the system can be emulated, allowing you to debug as if you were using JTAG, even if KGDB isn't enabled. The GDB commands used remain the same.

GDB commands Here are some of the most commonly used GDB commands in kernel debugging:

```
// Inserts a breakpoint at a symbol
    break <sym>
// Continues execution until the next breakpoint
    continue
// Steps into subroutines and breaks at each line
    step
// Steps over subroutines without entering them
    next
// Prints a stack backtrace
    bt
// Lists loaded modules
    lx-lsmod
// Lists processes
    lx-ps
// Searches for a relevant command or symbol in GDB
    apropos lx
// Prints task struct of a process by address
    p *(struct task_struct *) <address of process from lx-ps>
// Gets a container struct from a list head pointer
    p $container_of(listheadp, "struct <type x>", "list head name <type x>")
// Prints the task struct of a process by PID
    p $lx_task_by_pid(<PID>)
// Sets a GDB variable to a task struct
    set $t = $lx_task_by_pid(<PID>)
// Prints the state of the process
    p $t -> __state
```