
CMPSC 473

Operating Systems Design & Construction

CPU Virtualization
Dynamic Memory Allocation

Instructor: Ruslan Nikolaev

The Pennsylvania State University

January 22, 2026 – Lecture 4

CPU Virtualization

The Context Switch Challenge

[lecture4.pdf > page=2](#)

In a virtualized environment, the Operating System (OS) must manage the execution of multiple processes on a single physical CPU. This creates an illusion that each process has exclusive access to the processor.

Consider two processes, A and B. The OS must switch execution between them (Time Slicing). This raises four fundamental questions regarding system design and control:

1. **Safety:** What if a process executes undesirable instructions (e.g., halting the machine, accessing restricted memory)?
 2. **Control:** How does the OS regain control of the CPU once a user program starts running?
 3. **Restoration:** How does the OS ensure a process resumes exactly where it left off?
 4. **Scheduling:** How does the OS decide which process to run next?
-

Controlled Execution: Traps and Privileged Instructions

[lecture4.pdf > page=8](#)

Undesirable Instructions

If a user process were allowed to execute every instruction in the Instruction Set Architecture (ISA), it could compromise the entire system (e.g., executing a `HALT` instruction or directly accessing hardware I/O). To prevent this, instructions are categorized into:

1. **User ISA (Unprivileged):** Safe instructions available to all applications (e.g., `add`, `sub`, `jmp`, `load`, `store`).
2. **System ISA (Privileged):** Instructions reserved for the OS (e.g., `halt`, I/O operations, manipulating system registers).

Traps

[lecture4.pdf > page=11](#)

A **trap** is a mechanism that transfers control from the currently running process to the OS. The CPU enters a special state and jumps to a specific handler.

Triggers for Traps:

- **Involuntary:**
 - Execution of a privileged instruction by a user process.
 - Error conditions (e.g., Illegal Instruction, Divide by Zero, Memory Violation).
- **Voluntary:**
 - **System Calls:** A user program explicitly requests an OS service (e.g., reading a file). Historically, this was done using specific trap instructions (e.g., `int` on x86).

CPU Action upon Trap:

1. Save the current process state (Instruction Pointer, flags, etc.).
2. Look up the handler address in the **Trap Vector Table** using the trap number.

3. Jump to the handler code in the OS.
 4. Execute the handler.
 5. Restore process state and resume execution (Return-from-trap).
-

Dual CPU Mode

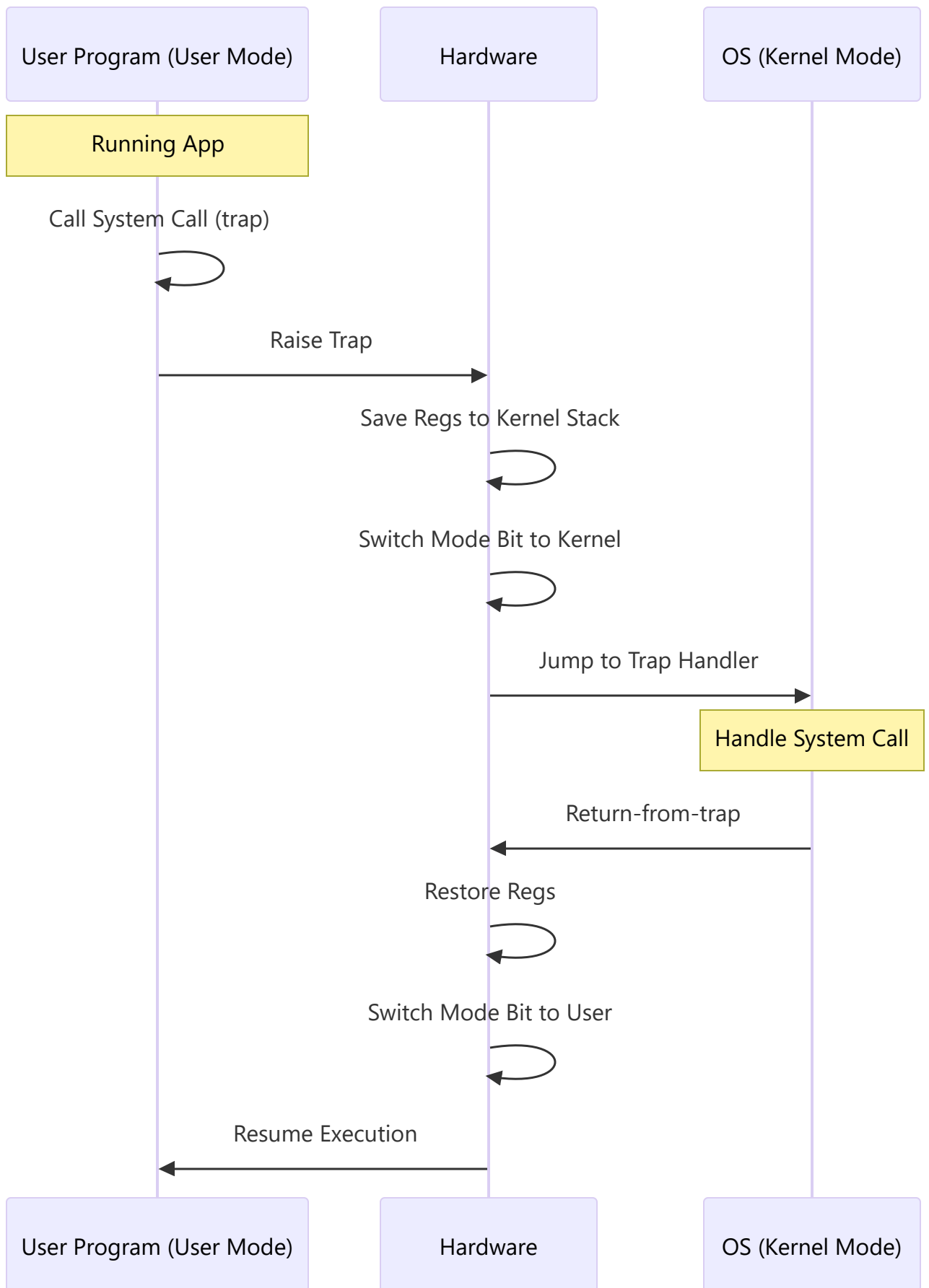
[lecture4.pdf > page=17](#)

To enforce the distinction between privileged and unprivileged execution, hardware provides at least two modes of operation, tracked by a **Mode Bit**:

1. **User Mode (Ring 3 on x86)**: Restricted access. Attempting privileged instructions causes a trap.
2. **Kernel Mode (Ring 0 on x86)**: Unrestricted access to full ISA and hardware.

Limited Direct Execution Protocol

The transition between modes is strictly controlled via "gates."



The Kernel Stack

[lecture4.pdf > page=21](#)

The OS maintains a separate **per-process kernel stack**.

- **Why?** Reliability and security. The user stack may be corrupted or too small.
 - **Function:** It stores the register context (PC, Stack Pointer, flags) of the user process when a trap occurs.
-

Regaining Control: Interrupts

[lecture4.pdf > page=24](#)

Question: If a process is running in User Mode and does not call a system call (voluntary trap) or cause an error (involuntary trap), how does the OS regain control to switch processes?

Mechanisms

1. **Cooperative (Yielding):** The process explicitly gives up the CPU (e.g., via `sched_yield`). This is unreliable; a malicious or buggy loop could freeze the system.
2. **Preemptive (Interrupts):** The OS uses a hardware **timer interrupt** that fires periodically (e.g., every few milliseconds).

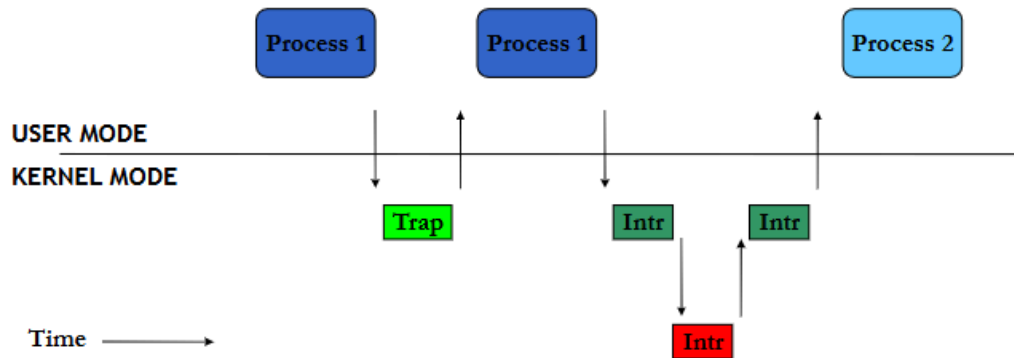
Interrupts vs. Traps

[lecture4.pdf > page=27](#)

While traps are triggered by the currently running instruction (synchronous), **Interrupts** are asynchronous events triggered by external hardware.

- **Examples:** Timer expiration, Disk I/O completion, Network packet arrival, Keystrokes.
- **Handling:** Similar to traps. The CPU detects the interrupt, saves state, switches to Kernel Mode, and jumps to the address defined in the **Interrupt Vector Table**.

Interrupts and Traps



- Only two ways to enter kernel mode from user mode
 - We previously already considered traps and trap handlers

Signals: Application-Level Interrupts

[lecture4.pdf > page=31](#)

Signals provide a mechanism for the OS to notify a process of system events. They effectively function as "interrupts" for user-space applications.

- **Creation:**
 - By the OS (e.g., detecting a segmentation fault).
 - By another process (e.g., sending a `kill` signal).
- **Handling:** A process can register a custom signal handler function. If triggered, the OS pauses the process's main execution, runs the handler on the user stack, and then resumes the main execution.

Signal vs. Interrupt Layout

- **Hardware Interrupt:** Hardware → OS Handler (Kernel Mode).
- **Signal:** OS → Process Handler (User Mode).

Dynamic Memory Allocation

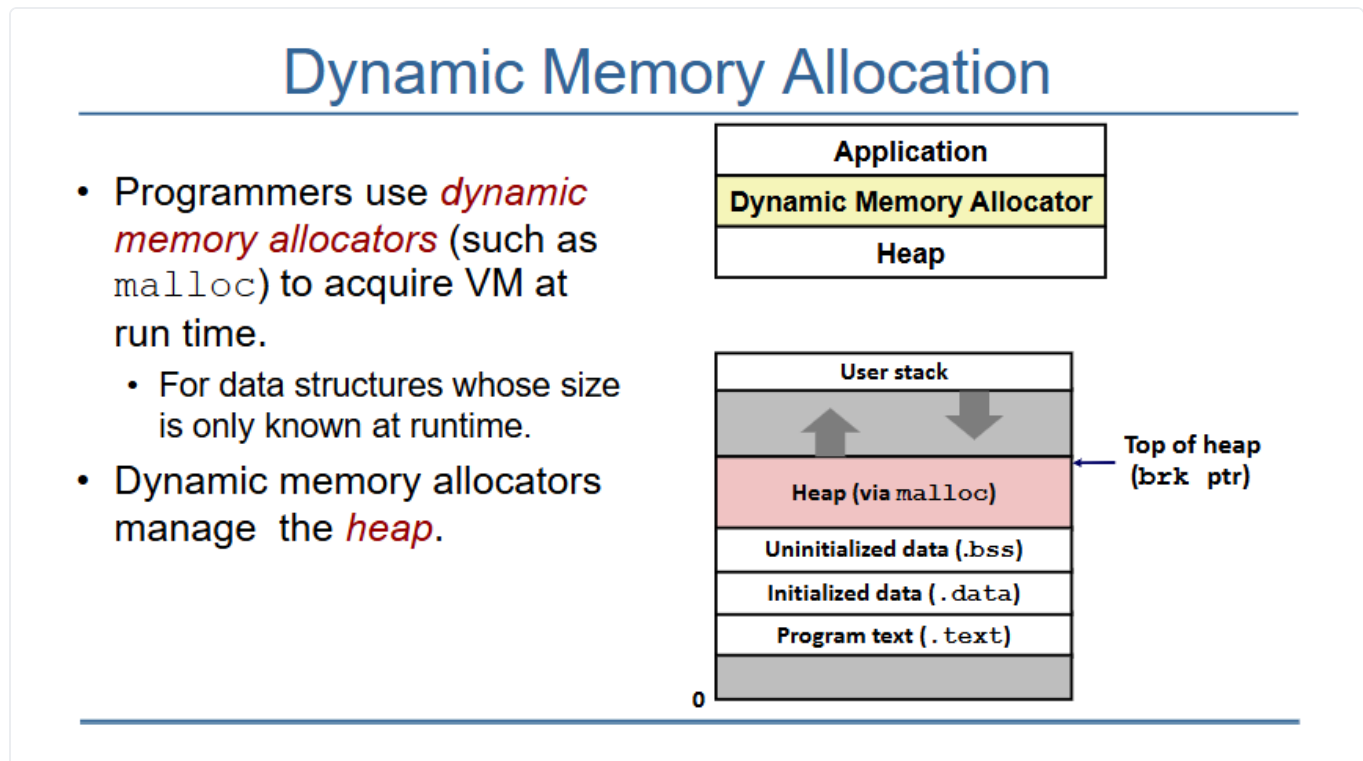
Memory Management Context

[lecture4.pdf > page=37](#)

Memory management operates at two granularities:

1. **OS Level (Virtual Memory):** Manages coarse-grained pages.
2. **User-Space Level (Dynamic Allocation):** Manages fine-grained words within the heap.

The heap is a region of memory used for data structures whose size is known only at runtime. It grows upward (towards higher addresses), while the stack grows downward.



Types of Allocators

1. **Explicit Allocator:** The application must explicitly allocate and free memory (e.g., C `malloc / free`, C++ `new / delete`).
2. **Implicit Allocator:** The application allocates, but the system handles freeing via Garbage Collection (e.g., Java, Python).

The `malloc` Package (C Standard Library)

API Overview

- `void *malloc(size_t size)`
 - Allocates a block of at least `size` bytes.
 - Returns a pointer to the block, aligned to system requirements (8-byte on x86, 16-byte on x86-64).
 - Returns `NULL` if allocation fails.
 - **Note:** memory is *uninitialized*.
- `void free(void *p)`
 - Returns the block pointed to by `p` to the free pool.
 - `p` must be a pointer returned by a previous `malloc`, `calloc`, or `realloc`.
- `void *calloc(size_t nmemb, size_t size)`
 - Allocates memory for an array and **initializes to zero**.
- `void *realloc(void *ptr, size_t size)`
 - Resizes a previously allocated block.
- `void *sbrk(intptr_t increment)`
 - Low-level system call used internally by allocators to grow/shrink the heap size by moving the program break.

Technical Details

- **`void *` (Void Pointer):** A generic pointer type with no associated data type. It cannot be dereferenced directly; it must be typecast first. It allows `malloc` to return a pointer usable for any data type.
- **`size_t`:** An unsigned integer type used for object sizes.
- **Alignment:** Blocks must be aligned to specific boundaries (double-word alignment) to satisfy hardware requirements for accessing specific data types (e.g., `double`).

Example Usage

[lecture4.pdf > page=45](#)

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

void foo(int n) {
    int i, *p;

    // Allocate a block of n integers
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
    }
}
```

```
    exit(0);
}

// Initialize
for (i = 0; i < n; i++) {
    p[i] = i;
}

// Explicitly free memory
free(p);
}
```

Allocator Constraints

[lecture4.pdf > page=52](#)

Designing an allocator is difficult because it must operate under strict constraints while balancing performance (throughput) and memory utilization (minimizing fragmentation).

Application Constraints:

- Can issue arbitrary sequences of allocate/free requests.
- Can only issue `free` on blocks previously returned by `malloc`.

Allocator Constraints:

1. **Immediate Response:** Must respond to `malloc` immediately (cannot buffer or reorder requests to optimize placement).
2. **Free Memory Only:** Can only manipulate the free list; cannot modify allocated blocks.
3. **No Compaction:** Allocated blocks generally cannot be moved once allocated (pointers in the application would break).
4. **Alignment:** Must meet hardware alignment requirements.