Team Emertxe

Kernel Memory Allocation

What is Kernel Memory Allocation?

- The Linux kernel provides several methods for dynamic memory allocation.
- The two primary functions used are kmalloc() and vmalloc().

Key Factors to Consider:

- Memory size requirements.
- Contiguity of physical memory.
- Performance implications.

Overview of kmalloc()

What is kmalloc()?

- Allocates physically **contiguous** memory in the kernel address space.
- Similar to malloc() in user-space but for kernel use.

Key Characteristics:

- Allocates memory from the **slab allocator**.
- Suitable for **small, frequent allocations**.
- Works well for DMA (Direct Memory Access) operations.

Use Cases for kmalloc()

Typical Scenarios:

1.Device Drivers:

- Allocating small buffers for data transfers.
- Allocating memory for hardware registers.

2.Interrupt Handlers:

- Needs fast allocation for handling real-time operations.

3.Short-lived Structures:

- Temporary allocations that are frequently created and destroyed.

Device Driver Buffer Allocation

Example:

```
char *buffer;
buffer = kmalloc(1024, GFP_KERNEL);
if (!buffer) {
printk(KERN_ERR "Memory allocation failed\n");
}
```

Why kmalloc?

- Physically contiguous memory needed for DMA.
- Low latency memory access.

Function Signatures and Details

kmalloc()

```
void *kmalloc(size_t size, gfp_t flags);
```

Arguments:

- size: Size of the allocation.
- flags: Allocation behaviour (e.g., GFP_KERNEL, GFP_ATOMIC).

Overview of vmalloc()

What is vmalloc()?

- Allocates virtually contiguous memory but physically scattered.
- Used when large memory allocations are required.

Key Characteristics:

- Memory is allocated via page tables, not slab caches.
- Suitable for large allocations (e.g., >4KB).
- Higher overhead due to page table mapping.

Use Cases for vmalloc()

Typical Scenarios:

1.Large Kernel Buffers:

- Allocating buffers for logs, packet processing, and cache systems.

2.Module Initialization:

- Modules requiring significant memory on load.

3.Sparse Data Structures:

- When physical contiguity is not critical but virtual continuity is needed.

Large Buffer Allocation for Logging

Example:

```
char *buffer;
buffer = vmalloc(10 * 1024 * 1024); // 10MB buffer
if (!buffer) {
    printk(KERN_ERR "Memory allocation failed\n");
}
```

Why vmalloc?

- Large buffer size beyond slab allocator limits.
- No strict need for physical contiguity.

Function Signatures and Details

vmalloc()

```
void *vmalloc(unsigned long size);
```

Arguments:

- size: Size of the allocation.

Advantages and Disadvantages

Allocator	Advantages	Disadvantages
kmalloc	Fast, suitable for DMA, lower fragmentation	Limited to physically contiguous memory
vmalloc	Allows larger allocations, virtually contiguous	Slower, higher overhead due to page mapping

Memory Allocation Best Practices

Choosing the Right Allocator:

1. Use **kmalloc** for smaller, performance-critical allocations.
2. Use **vmalloc** for larger memory needs where physical contiguity is not required.
3. Always check return values to avoid null pointer dereference.
4. Prefer kzalloc() for zeroed allocations to avoid garbage values.

Debugging Kernel Allocations

Common Tools for Debugging:

- **/proc/meminfo**: Check system memory usage.
- **kmemleak**: Detect memory leaks.
- **dmesg**: View kernel logs for allocation failures.

Example Debugging Command:

```
cat /proc/meminfo | grep Slab
```

kmemleak to Detect Memory Leaks

Linux provides a memory leak detector called kmemleak, which can help in tracking allocations and potential leaks.

Steps to use kmemleak:

Enable kmemleak at boot by adding the kernel parameter:

```
kmemleak=on
```

After booting, trigger a scan and check for leaks:

```
echo scan > /sys/kernel/debug/kmemleak
```

```
cat /sys/kernel/debug/kmemleak
```

If you see any suspicious entries related to your module, it may indicate memory leaks.

Inspect kernel memory allocations

1. Use /proc/meminfo to get an overview of memory usage.
2. Use /proc/slabinfo for kmalloc() tracking.
3. Use /proc/vmallocinfo for vmalloc() tracking.
4. Use dmesg for debugging memory allocation messages.
5. Use kmemleak and ftrace for tracking potential leaks and tracing.

Memory Mapping

Introduction to Memory Mapping

What is mmap?

- Memory mapping maps a file or device into the address space of a process.
- Enables direct access to memory without using read/write system calls.

Key Characteristics:

- Improves performance by avoiding frequent I/O operations.
- Provides shared memory between kernel and user space.

Why mmap?

Challenges Without mmap:

- Frequent system calls slow down performance.
- Requires complex buffer management for data exchange.

Advantages of mmap in Drivers:

- Efficient data transfer with minimal overhead.
- Direct access to device memory from user-space applications.
- Useful for high-speed hardware like graphics, network devices.

Use Cases in Device Drivers

Common Applications:

- **Frame Buffers:** Allow user-space apps to modify the display directly.
- **Shared Memory IPC:** Enables inter-process communication efficiently.
- **High-Speed Data Acquisition:** Useful in network and video processing devices.
- **Custom Hardware Access:** Provides direct access to registers/memory.

Implement mmap in Drivers

1. Allocate Kernel Memory (get_zeroed_page)
2. Map Kernel Memory to User Space (remap_pfn_range)
3. Provide mmap File Operation
4. Converts a virtual address to a physical address virt_to_phys
5. User-Space Application Calls mmap()

Advantages of Using mmap

- Reduces CPU usage by avoiding unnecessary copies.
- Faster device access compared to traditional I/O methods.
- Simplifies large data transfers.

mmap vs Traditional Read/Write

Feature	mmap	read/write
Performance	High	Medium
Complexity	Low	High
Buffering	Not needed	Needed
Use Case	High-speed devices	Standard file access

Common Issues and Debugging mmap

- **Issue:** mmap failure

Solution: Check permissions and memory limits.

- **Issue:** Incorrect data mapping

Solution: Verify PFN calculations.

- **Issue:** Segmentation faults

Solution: Ensure valid memory boundaries.

Thanks