Embedded Operating Systems - Part 5: Linux memory management and kernel memory

Tammy Noergaard - April 30, 2013

Editor's Note: Embedded Systems Architecture, 2nd Edition, is a practical and technical guide to understanding the components that make up an embedded system's architecture. Offering detailed explanations and numerous code examples, the book provides a comprehensive get-up-and-running reference for those new to the field and those updating their skills. This excerpt offers an introduction and review of embedded operating systems. In Part 1, the author defined the unique characteristics of an embedded OS and describes how an embedded OS works with processes. In Part 2, the author reviewed the different types of process schedulers found in embedded OSs. In Part 3, the author discussed how embedded OSs provide mechanisms that let tasks intercommunicate and synchronize their behavior. In Part 4, the author described the various techniques used for memory management in an embedded OS. In this installment, the author describes Linux memory management and the kernel memory space.

Adapted from "Embedded Systems Architecture, 2nd Edition" by Tammy Noergaard (Newnes)

Example 15: Linux Memory Management and Segmentation

Linux processes are made up of text, data, and BSS static segments; in addition, each process has its own stack (which is created with the fork system call). Heap space for Linux tasks are allocated via the C-language malloc/new system calls to dynamically allocate memory. There is no GC in Linux, so the programmer must deallocate memory manually via the free() system call.
void *mem_allocator(void *arg)
{
    int i;
    int thread_id = *(int *)arg;
    int start = POOL_SIZE * thread_id;
    int end = POOL_SIZE * (thread_id + 1);
    if (verbose_flag) {
        printf("Releaser %i works on memory pool %i to %i\n", thread_id, start, end);
        printf("Releaser %i started\n", thread_id);
    }
    while (!done_flag) {
        /* find first NULL slot */
        for (i = start; i < end; ++i) {
            if (NULL == mem_pool[i]) {
                mem_pool[i] = malloc(1024);
                if (debug_flag)
                    printf("Allocate %i: slot %i\n", thread_id, i);
                break;
            }
        }
        pthread_exit(0);
    }
}

void *mem_releaser(void *arg)
{
    int i;
    int loops = 0;
    int check_interval = 100;
    int thread_id = *(int *)arg;
    int start = POOL_SIZE * thread_id;
    int end = POOL_SIZE * (thread_id + 1);
    if (verbose_flag) {
        printf("Releaser %i works on memory pool %i to %i\n", thread_id, start, end);
        printf("Releaser %i started\n", thread_id);
    }
    while (!done_flag) {
        /* find non-NULL slot */
        for (i = start; i < end; ++i) {
            if (NULL != mem_pool[i]) {
                void *ptr = mem_pool[i];
                mem_pool[i] = NULL;
                free(ptr);
                ++counters[thread_id];
                if (debug_flag)
                    printf("Releaser %i: slot %i\n", thread_id, i);
                break;
            }
            loops++;
            if ((0 == loops % check_interval) &&
                (elapsed_time(&begin) > run_time)) {
                done_flag = 1;
                break;
            }
        }
        pthread_exit(0);
    }
}
Either with or without segmentation, some OSs divide logical memory into some number of fixed-size partitions, called blocks, frames, pages, or some combination of a few or all of these. For example, with OSs that divide memory into frames, the logical address is a compromise of a frame number and offset. The user memory space can then, also, be divided into pages, where page sizes are typically equal to frame sizes.

When a process is loaded in its entirety into memory (in the form of pages), its pages may not be located within a contiguous set of frames. Every process has an associated process table that tracks its pages, and each page’s corresponding frames in memory. The logical address spaces generated are unique for each process, even though multiple processes share the same physical memory space.

**Title-1**

Logical address spaces are typically made up of a page-frame number, which indicates the start of that page, and an offset of an actual memory location within that page. In essence, the logical address is the sum of the page number and the offset. (See Figure 9-37.)

![Figure 9-37. Paging.[3]](image)

An OS may start by prepaging, or loading the pages needed to get started, and then implementing the scheme of demand paging where processes have no pages in memory and pages are only loaded into RAM when a page fault (an error occurring when attempting to access a page not in RAM) occurs. When a page fault occurs, the OS takes over and loads the needed page into memory, updates page tables, and then the instruction that triggered the page fault in the first place is re-executed. This scheme is based upon Knuth’s Locality of Reference theory, which estimates that 90% of a system’s time is spent on processing just 10% of code.

Dividing up logical memory into pages aids the OS in more easily managing tasks being relocated in and out of various types of memory in the memory hierarchy, a process called swapping. Common page selection and replacement schemes to determine which pages are swapped include:

- **Optimal**: using future reference time, swapping out pages that won’t be used in the near future.
- **Least Recently Used (LRU)**: which swaps out pages that have been used the least recently. **FIFO**: which as its name implies, swaps out the pages that are the oldest (regardless of how often it is
accessed) in the system. While a simpler algorithm than LRU, FIFO is much less efficient.

- Not Recently Used (NRU): swaps out pages that were not used within a certain time period.
- Second Chance: FIFO scheme with a reference bit, if “0” will be swapped out (a reference bit is set to “1” when access occurs, and reset to “0” after the check).
- Clock Paging: pages replaced according to clock (how long they have been in memory), in clock order, if they haven’t been accessed (a reference bit is set to “1” when access occurs, and reset to “0” after the check).

While every OS has its own swap algorithm, all are trying to reduce the possibility of thrashing, a situation in which a system’s resources are drained by the OS constantly swapping in and out data from memory. To avoid thrashing, a kernel may implement a working set model, which keeps a fixed number of pages of a process in memory at all times. Which pages (and the number of pages) that comprise this working set depends on the OS, but typically it is the pages accessed most recently. A kernel that wants to prepage a process also needs to have a working set defined for that process before the process’s pages are swapped into memory.

Title-1

Virtual Memory
Virtual memory is typically implemented via demand segmentation (fragmentation of processes from within, as discussed in a previous section) and/or demand paging (fragmentation of logical user memory as a whole) memory fragmentation techniques. When virtual memory is implemented via these “demand” techniques, it means that only the pages and/or segments that are currently in use are loaded into RAM.

![Figure 9-38. Virtual memory.][3]

As shown in Figure 9-38, in a virtual memory system, the OS generates virtual addresses based on the logical addresses, and maintains tables for the sets of logical addresses into virtual addresses conversions (on some processors table entries are cached into translation lookaside buffers (TLBs); see Chapters 4 and 5 for more on MMUs and TLBs). The OS (along with the hardware) then can end up managing more than one different address space for each process (the physical, logical, and virtual). In short, the software being managed by the OS views memory as one continuous memory space, whereas the kernel actually manages memory as several fragmented pieces which can be segmented and paged, segmented and unpaged, unsegmented and paged, or unsegmented and unpaged.
9.3.2 Kernel Memory Space

The kernel’s memory space is the portion of memory in which the kernel code is located, some of which is accessed via system calls by higher-level software processes, and is where the CPU executes this code from. Code located in the kernel memory space includes required IPC mechanisms, such as those for message passing queues. Another example is when tasks are creating some type of fork/exec or spawn system calls. After the task creation system call, the OS gains control and creates the Task Control Block (TCB), also referred to as a Process Control Block (PCB) in some OSs, within the kernel’s memory space that contains OS control information and CPU context information for that particular task. Ultimately, what is managed in the kernel memory space, as opposed to the user space, is determined by the hardware, as well as the actual algorithms implemented within the OS kernel.

As previously mentioned, software running in user mode can only access anything running in kernel mode via system calls. System calls are the higher-level (user mode) interfaces to the kernel’s subroutines (running in kernel mode). Parameters associated with system calls that need to be passed between the OS and the system caller running in user mode are then passed via registers, a stack, or in the main memory heap. The types of system calls typically fall under the types of functions being supported by the OS, so they include file systems management (i.e., opening/modifying files), process management (i.e., starting/stopping processes), and I/O communications. In short, where an OS running in kernel mode views what is running in user mode as processes, software running in user mode views and defines an OS by its system calls.

© 2013 Elsevier, Inc. All rights reserved. Printed with permission from Newnes, a division of Elsevier. Copyright 2013. For more information on this title and other similar books, please visit www.newnespress.com.

If you liked this and would like to see a weekly collection of related products and features delivered directly to your inbox, click here to sign up for the EDN on Systems Design newsletter.