Memory Management
CS421 – Dr. Shaffer
Due: November 9, 2011

---

1 Introduction

In this lab we will add support for dynamic memory allocation and for checking stack bounds at each context switch.

2 Stack bounds checking

Since we don't have a memory management unit on the AVR, there is nothing that prevents a task's stack from growing and bumping into another task's stack or into heap space. When a task is created, the expected maximum size of that task's stack is specified. The kernel "reserves" room for that task's stack just below the previously added task's stack (or below the kernel stack if this is the first task). When a context switch occurs we can verify that the active task does not **currently** exceed its allocated stack region. Since these context switches are asynchronous when we are using a preemption timer, there is *some chance* that we will catch a rogue task before it damages things so much that we can't evenly properly display a kernel fault. A "stack exceeded" kernel fault is much more useful than a system that seems to randomly reset or display other erratic behaviors.

Ultimately we want to modify os_yield so that it checks that a task's stack pointer is valid. Unfortunately os_yield is very sensitive to stack and register utilization in code generated by the C compiler. It is a "naked" function, leaving all register saving and restoring to the code in the function itself. We're better off not changing this code. The next subsection walks you through the process of renaming this original function and then calling this renamed function from a (dressed) os_yield. If you feel comfortable with trying this yourself, go to it, otherwise read on.

2.1 Coding

The current kernel uses the task's specified maximum stack size (from os_addTask()) to allocate stack space but it doesn't keep it for future reference. Modify the task control block to include a variable for the smallest address usable by the stack. Modify the os_addTask() function so that it stores each task's maximum stack size and bottom location in the newly created task control block.

Now rename os_yield to _os_basicYield (the leading underscore means that it is a private function). Move the function definition so that it is located with other private functions. Add a declaration for this function near the top of oc.c (look for the declarations of the other private functions) and be sure to mark it "naked" (you will see examples of other naked private already in this file). Edit os.h and remove the naked attribute from os_yield. Now, create a new os_yield in os.c that simply calls _os_basicYield. At this point your operating system should compile and run as before. You might try testing it with your weather station code or a few of the applications in the tests directory. Don't proceed without testing thoroughly because you want to make sure that you've "refactored" properly.

Now you will code the stack bounds checking. Modify os_yield so that, before it calls _os_basicYield it checks that the stack pointer, adjusted as discussed below, is within the bounds of the active task's stack. When performing your test

you must take into account that saving the context takes 33 bytes (32 GPR's and SREG). You're better off defining this value as a preprocessor symbol rather than just hardcoding it in this function. Add a comment to _os_basicYield expressing that any changes to the context save code must be reflected in this constant. In addition to saving the context, the call to _os_basicYield will require two more bytes of stack space (for call). With these adjustments taken into account, you should be able to validate the stack pointer. If, with these adjustments, it will not remain within the task's stack bounds, signal a kernel fault (add a new error code called OS_ERROR_STACK_OVERFLOW).

2.2 Testing

Add an application that creates a "stack-breaker" task that violates its stack bounds by recursing too deeply into a function. If you just code your stack-breaker to recurse infinitely you may find that the kernel doesn't catch it. This can happen if the stack grows down into the memory the kernel is using before the first context switch occurs. The kernel's memory becomes severely corrupted and your system becomes unresponsive. Of course this depends on your preemption timer and the details of your code. As I said, this isn't a perfect stack bounds checker but it is often better than nothing. Work a bit to come up with some stack-breakers that get caught by the OS.

Finally, verify that your weather station code still works with this new OS.

3 Dynamic memory allocation

In this section we will add facilities to our kernel that make it possible for tasks to request memory dynamically. We will also add a user-level library to manage the allocated memory frames. Frankly dynamic memory allocation in an embedded system should be kept to a minimum but it can be very useful as long as it is used carefully.

Just to ensure that there is no confusion: this section is not about verification, like the last section was, it is about providing a new facility to tasks. We can do nothing to ensure that tasks do not violate their memory constraints.

As discussed in class, most operating systems track physical memory utilization in fixed-size chucks called "page frames." The page frame is the unit of memory allocation/deallocation at the process level. Processes may dynamically (at run-time as opposed to load-time) request frames from the operating system. It is up to the process to manage use of the memory in the frame(s) given to it. If a task does not use an entire memory frame, the unused portion of this frame is wasted space. The frame size on these operating systems is selected to ensure that little such waste will occur under the typical uses expected for that OS. Once a user process has a page frame, it normally uses the memory within that frame as a "memory pool." Using the library functions in the malloc family to allocate and deallocate memory within that pool. If malloc cannot find the memory it needs, it asks the OS for another (or more than one) frame.

Unlike traditional operating systems, our OS does not need to be concerned with loading processes (tasks) into RAM. Our processes execute directly from Flash. The static memory utilized by our processes has already been compiled into the data and bss segments and, when our program starts, initialization of this RAM is performed by the "CRT" startup procedure. The RAM that we're concerned with here is above this statically-allocated space but below the last allocated stack block. This memory region is available for all tasks to use but we need to coordinate access to it.

The C language memory allocation library includes the primitive functions malloc and free. One simple dynamic memory allocation scheme would be to have tasks call these functions directly. Unfortunately these functions are not re-entrant and so separate tasks calling them will ultimately lead to programs with difficult-to-find race conditions. To avoid this problem we could disable interrupts while in these calls but these calls can be very time consuming and disabling interrupts for a long time is a bad idea. Finally, as an alternative to disabling interrupts, we can protect these calls with a global mutex but

then every memory allocation performed by a task will require locking and unlocking a mutex. This will be inefficient for applications with tasks that allocate and free small segments of memory frequently.

With these concerns in mind we will implement a two-level system:

- The kernel provides mutex-protected dynamic memory allocation functions. Tasks may use these to allocate with the understanding that there is significant overhead.
- Tasks can optionally use a user-level library to manage their dynamically allocated memory. This library allocates memory using the kernel tools but allocates in increments of pages and then manages memory allocation within these pages using faster unprotected library calls similar to malloc and free.

This scheme gives us the best of both worlds. Applications, and even individual tasks, can choose the allocator that best meets their needs. Performance and fragmentation problems can be dealt with by tuning the parameters associated with allocation (page size, fragmentation thresholds etc).

3.1 Kernel memory allocation system calls

As discussed in the previous sub-section, the kernel will provide mutex-protected access to the libc malloc routine. Begin by reading: `http://www.nongnu.org/avr-libc/user-manual/malloc.html` to see how to configure malloc. Note specifically that malloc allocates memory between the addresses contained in the two static global variables __malloc_heap_start and __malloc_heap_end. The former is set correctly for us automatically. The latter must be set **before** any calls to malloc but **after** the last task's stack is allocated. Complete the functions in kernel/memory.c so that they behave as documented.

3.2 Userland memory allocation

In this sub-section we provide a library for memory allocation for user processes. From the client's point of view this library is similar to the malloc-related tools provided by libc. The basic idea behind these tools is that they manage (allocate, deallocate) memory within regions, collectively referred to as the task memory "pool", that were previously given to the task by the kernel (though calls to osMem_alloc()). When the user-space allocator cannot find space to satisfy an allocation request, it must ask the kernel for more memory, adding the resulting region to its memory pool. Providing an efficient and useful implementation of this allocator is challenging. Rapid allocating and freeing of memory, typical for tasks using dynamic memory, can result in a fragmented memory pool, leading to poor utilization of memory. Compacting the memory pool on each allocation/deallocation request can result in poor performance.

Complete the functions in util/memory.c so that they behave as documented.

4 Testing

I have included some test cases in the tests directory. Please write your own additional tests. There are lots of "edge cases" in these allocators and they need to be tested thoroughly.