Jackson Paull

JHP2539

27 March 2025

ECE 381L - RTOS

Lab 5 Report

1. **OBJECTIVES**

Lab 5 was focused on the implementation of a dynamic memory manager (the heap) and an integration with the ELF loader to be able to read and spawn processes. Unfortunately I didn’t have as much time to toy around with this lab, so my implementations are pretty bare bones.

1. **HARDWARE DESIGN**

None for this lab

1. **SOFTWARE DESIGN** - *Note: See GitHub, tag ‘lab5-release’ for source code files.*

Elf Loader Diagram

A diagram of a diagram

AI-generated content may be incorrect.

*Fig 1. ELF Loading Process*

SVC Handler

A screenshot of a computer program

AI-generated content may be incorrect.

*Fig 2. SVC Handler implementation*

*Note: This is the basic immediate solution I implemented for lab check-out. The commented part shows the idea of a better solution. R12 just needs to be bitshift by an appropriate amount corresponding to the length of the B instruction.*

The Heap

The heap is constructed of blocks with a header and footer (see fig 3.). The heap iteratively scans blocks on malloc until it finds one of suitable size and allocates (splitting if possible) the block. On free, blocks are joined with neighboring free blocks.

A diagram of a tall column

AI-generated content may be incorrect.

*Fig 3. The Heap*

Processes and the PCB

Processes are allocated with space for their text, data, and heap (see fig 4). These spaces are allocated in the context of the parent process (i.e. using the parent’s heap). In practice this means that processes are given a fixed image size upon creation, and processes cannot spawn other processes to “game” this system and hog memory resources from other processes. A process is responsible for all of its code, including any child processes, fitting within its heap space.

**A screen shot of a computer code

AI-generated content may be incorrect.**

*Figure 4. The PCB Struct*

On Kill, threads will decrement the numThreadsAlive member of their PCB. Once this reaches 0, the PCB will free all of its resources (from the context of its parent process).

New Interpreter Commands

The only new command added to the interpreter is the run command. It takes a single parameter, the path to the file to execute, and calls the elf loader.

run <path\_to\_axf>

1. **MEASUREMENT DATA**

*Fig 5. OS Kernel Task Execution*

*Fig 6. User Task Execution*

1. **ANALYSIS AND DISCUSSION**
2. *Briefly explain the dynamic memory allocation algorithm in your heap manager. Does this implementation have internal or external fragmentation?*
   1. I was originally going to implement Knuth’s buddy allocation algorithm, but I ended up falling back to the basic header/footer design in the interest of time.
   2. The header and footer are each one word long, representing the number of bytes in the block.
   3. A negative value is a free block, and a positive value is an allocated block.
   4. Using 1 word for the header and footer means that block sizes up to ~2GB are supported.
   5. My heap allocates pages aligned to a word (4-byte) level. Hence internal fragmentation is unavoidable. Assuming a uniform distribution of size requested, there is an average of 2 bytes of internal fragmentation. The same would be true for buddy allocation, though the buddy allocation would have more internal fragmentation depending upon the distribution of malloc size requests.
   6. External fragmentation is the real issue with this implementation, as small free blocks are often isolated between semi-permanently allocated blocks (like thread stacks for long living threads). Knuths buddy allocation would be better in this regard, as in practice blocks of similar size will usually be placed near each other so large blocks like stacks will not fragment smaller blocks like TCBs or anything else on the heap.
3. *How many simultaneously active processes can your system support? What factors limit this number, and how could it be increased?*
   1. My heap is initialized to hold 12288 (0x3000) bytes, and processes are allocated a 2048 (0x800) heap size within the entire system heap. Hence a maximum of 6 process heaps can fit inside the system heap, but we must consider the base OS processes which have their stacks on the base heap, hence 5 processes can be run simultaneously.
   2. This number can be increased by increasing the total heap size, or decreasing the heap size allocated to processes.