CS230/330: Project 2
Memory and Basic Process Management

Handed out: Friday, April 14, 2000
Due: Friday, April 28, 2000, 11:59 p.m.

1 Assignment

The purpose of this project is to extend your Yalnix kernel to support virtual memory so that you can load and execute multiple independent user programs. To do this, you will need to develop several critical components of your kernel. To make your life easier, we have outlined the basic stages below.

1.1 Physical Memory Management

When your `KernelStart()` function is invoked, the size of physical memory is passed as the `pmem_size` argument (see p. 36 of the Yalnix handout). Using this value, your first task is to implement physical memory management. To do this, you will need to do the following:

- Create a data structure that keeps track of free physical memory frames. The total number of memory frames is `pmem_size / PAGESIZE`. Therefore, one way to do this is to create an array of that size and record some kind of status for each physical memory frame (note: this is not the only way to do it).
- Write two functions for managing your physical memory management structure:
  - `int GetFreePage()`. Returns the frame number of a free physical memory page (if available). If no physical memory is available, this function should return -1 or some kind of error value.
  - `void ReleasePage(int fn)`. Releases a page of physical memory and returns it to the free list. `fn` is the frame number of the physical memory.
- Modify your `KernelStart()` function to place all of the memory frames not used by your kernel onto the free memory list you created in the first step. Hint: to do this, you should use the `sbrk(0)` function to find the highest memory address used by the kernel. Then, place all of the memory frames above that (up to the maximum physical memory) onto the free memory list using your `ReleasePage()` function.
- Test out your physical memory functions. To do this, you might want to stick a few print statements in your memory management functions to output some debugging information. Then, hack one of your interrupt handlers to grab physical memory frames and overwrite their values. A particularly good candidate for this is the `PEN_EVENT` handler. For example, you might stick this code fragment into it:

```
{
    int      pn, i;
    char     *c;
    pn = GetFreePage();
    if (pn < 0) {
      printf("Out of memory!\n");
    } else {
      /* Blow away the memory with 0's */
      c = (char *) (pn << PAGESHIFT);
      for (i = 0; i < PAGESIZE; i++) {
          c[i] = 0;
      }
    }
}
```

Now, with this in place, fire up your Yalnix and start moving the mouse around in the Yalnix window to generate events. You should see your kernel grabbing all kinds of physical memory. Continue doing this until you get the "Out of Memory" message. If you make it this far without crashing the simulator or causing your kernel to die, you have done everything correctly. Otherwise, there is a pretty good chance that you have placed some memory on the free list that you shouldn't have. After you have finished this, you may also want to test your kernel for proper release of memory (i.e., put all of the frames you grabbed back on the free list and see if things still work).

- Remove your testing code and read on. **DO NOT CONTINUE PAST THIS POINT UNLESS YOU ABSOLUTELY HAVE PHYSICAL MEMORY MANAGEMENT WORKING**.

1.2 Virtual Memory

- Remove the call to `BogusInitMemory()`. Replace it with your own boot-time memory initialization code that does the following:
  - Allocate memory for the kernel's page table. This should be some kind of array where the elements are of type `struct pte` (described on p. 9 of the handout).
  - Map the kernel's "text segment" so that it can be read and executed (protection mode `PROT_READ | PROT_EXEC`). You can use the `etext` symbol to determine the address of the top of the kernel's text segment.
  - Map the kernel's "data" and "bss" segments so that they can be read and written. You can use the `end` symbol to determine the address of the top of your kernel's bss segment. The top of the data segment can be obtained using the `sbrk(0)` system call. Note : you can assume that the bss/data segment starts at the page boundary immediately above the end of the text segment.

- Unmap the free pages so that they are invalid.
- Tell the hardware where your kernel page table is located by writing values to the `REG_PTBR0` and `REG_PTLR0` registers.
- Enable the virtual memory system.

Warning! Initially, you might want to avoid dynamic memory allocation in your kernel (i.e., `malloc()` or `new()`), since this will invoke your `SetKernelBrk()` function to allocate more memory on the kernel heap. Thus, you may want to start with large fixed-sized arrays for your page tables and other data structures. Once you have `SetKernelBrk()` working, you can replace these with more efficient data structures.

- Implement `SetKernelBrk()` so that your kernel can perform dynamic memory allocation. To do this, you will need to keep track of the top of the kernel's heap. Whenever `SetKernelBrk()` is called, you will have to map or unmap pages in the kernel's address space to satisfy the request. Once you have this working, it will be safe to use `malloc()`. Note : do not confuse `SetKernelBrk()` with the `Brk()` system call used to increase the memory used by user programs.

1.3 Processes

Next, you are going to start defining structures used to implement user processes.

- Define a process control block (PCB) data structure for your kernel that allows you to store all system state associated with a user process, including its machine context, memory management state, and any other state data you might need to manage. It is useful to spend some time thinking about what kinds of state you might need to store to implement various operations described in the Yalnix handout before proceeding. Don't forget that you will likely be placing PCBs on one or more queues.

- Modify your kernel to create a PCB for the in-kernel idle process from Project 1. In addition, your kernel should allocate a 1-page stack-frame for the idle process. To do this, grab a page of free memory and assign it to the idle process with read/write permissions. When creating the context for the idle process, be sure to set the stack pointer to this free page. The stack pointer should be set at least `INITIAL_STACK_FRAME_SIZE` bytes from the top of the page. After making these changes, test the idle process to make sure it still works.

- Copy the file **`/yalnix/sample/load.template`** to your directory and rename it to `load.c`. Modify the `LoadProgram()` function in this file to work with your PCB definition. You will also need to implement some user memory management functions to make this function work.

- Write a simple user test program similiar to the in-kernel idle process. Compile the program by adding it to the Makefile provided in Project 1 (look for rules related to `USER` programs). Call the program `init`

- Inside `KernelStart()`, use `LoadProgram()` to load the `init` process from the executable file `init`.
- Modify the timer interrupt handler to context switch between the idle process and the init process. At this point, you should see the idle and init process alternating. The idle loop should continue to print out a simple "idling" message. Once you have the basics of loading programs working, try writing some new `init` programs that stress various aspects of your kernel. Note that since most user-level system calls are still unimplemented, and in particular `Brk()`, you should not do any memory allocation in your init programs except to test the `Brk()` system call handler.
- Execute two user programs (`init1` and `init2`) as follows:
  - Set up `init1` while booting, as you did for `init` above.
  - During the first timer interrupt, swap out `init1` and load `init2`.
  - Context switch between `init1` and `init2` during each subsequent timer interrupt.
- Add support for dynamic process memory allocation and deallocation:
  - Implement the `Brk()` system call.
  - Allow for implicit memory allocation for the stack (i.e., automatically extend the stack size when a page fault occurs as a result of stack expansion).
- Implement the `Exit()` system call.
- Make sure to garbage collect system resources when a process exits.
- Remove all non-error messages from your kernel. For example, remove messages printed during benign interrupts or any debugging messages you may have added.

2 Specific Details and Suggestions

First and foremost, do some planning! Think about how you can perform iterative testing. This part of the project is fairly substantial, but if you attack it in small pieces, testing continually, it is not too bad. Here are some things to consider:

- If, and when both `init1` and `init2` complete, either due to calling `Exit()` or due to an exception, your kernel should exit quietly.
- Remember that `Brk()` can cause a process' heap to both grow and shrink.
- When handling a page fault that might be due to stack expansion, map as many pages as are needed to map the entire stack region from its current bottom to the address where the fault occurred (if possible).

- If you run out of system resources (e.g., memory), print an error message and return an error code to the user program if possible. If you cannot return cleanly because system resources ran out in response to a non-system call (e.g., a page fault caused by stack growth), print out an error message and terminate the offending process.
- The `end` and `etext` symbols described in the handout can be accessed using the following declarations:
  - `extern char end;`
  - `extern char etext;`

3 Logistics and Grading

This project must be handed in via CVS. Make sure that all of your source files, Makefile, and README files have been committed to the repository. If working in a group, the project should be committed to the repository located in `/export/home/groups/groupname`. Make sure your project name is "yalnix".

If you plan to start work on the next phase of the project before the due date, commit all of your files for Project 2 send us e-mail. We will collect the assignment and allow you to continue with your work.

To test correctness, we will run your `yalnix` executable on a number of different `init1` and `init2` programs. Some of these will be nice well-behaved Yalnix programs. Some of them will attempt to do nasty things (e.g., they may make system calls that you have not been asked to implement yet, they may be too big to fit in Yalnix's memory, they may generate illegal instructions, etc...). Your correctness score will be based on the extent to which you can load, run, and handle exceptions from these programs. Each test will be independent–in other words, your kernel need only load two programs, called `init1` and `init2`, and execute them.

To aid in your testing, we will place half of our test programs online in `/yalnix/test/project2`. Furthermore, a test script`/yalnix/bin/test2` will be made available to simplify the testing process (simply run this script to run a test). A separate collection of private tests will also be used in grading (and made available after the due date).

Do not wait until the last minute to start this part of the project.