CS230/330: Project 2
Memory and Basic Process Management

Handed out: Thursday, January 18, 2001
Due: Friday, February 2, 2001, 11:59 p.m.

NOTE: This part of the kernel project is difficult. DO NOT WAIT UNTIL THE LAST MINUTE TO START.

1 Assignment

In this part of the project, you will extend your Yalnix kernel to support virtual memory, processes, and context switching so that you can load and execute multiple independent user programs. To do this, you will need to develop a number of critical components of your kernel. In addition, you will implement a number of simple system calls. To make your life easier, we have outlined the basic stages below.

1.1 Physical Memory Management

When your kernel starts, it is loaded in physical memory starting at `PMEM_BASE` and extending to the end of the data segment (determined by the compiler and available as the value of `&end`). The total size of physical memory can be obtained by reading from the `REG_PHYSM` register. Using these values, your first task is to implement physical memory management. To do this, you need to create a data structure for keeping track of free physical memory pages. In addition, you need to figure out where to store this data structures in physical memory. Here's how to start:

- Create a data structure that keeps track of free physical memory frames. The total number of memory frames is `pmem_size / PAGESIZE` where `pmem_size` is the value of the `REG_PHYSM` register at kernel startup. Since you have not implemented dynamic memory management yet, you will probably want to use some kind of array. Furthermore, you will need to place this array in a well-known physical memory location. One possibility is to put the free page array in the page immediately following the kernel data segment.
- Write two functions similar to the following for managing your physical memory management structure:
  - `int GetFreePage()`. Returns the frame 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.

The names and parameters of these functions are up to you. Please note that in the current implementation of the hardware, physical page frame numbers are integers in the range of 2048-2175.

- Modify your kernel `main()` function to place all of the memory frames not used by your kernel onto the free memory list you created in the previous step. Hint: to do this, you need to look at the value of `&end` to figure out the last memory address being used by your kernel at startup. Then, place all of memory frames above that address (up to the maximum physical memory) onto the free memory list using your `ReleasePage()` function. Don't forget about the memory you might have used to store the free memory list itself—this should obviously **not** be put on the free list.

- Test out your physical memory functions. To do this, put a few kprint 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 `TRAP_PEN` handler. For example, add this code fragment in the pen handler interrupt:

```
{
    int      pn, i;
    char     *c;
    pn = GetFreePage();
    if (pn < 0) {
      kprints("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, fire up 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 a horrible death, you have probably done everything correctly. Otherwise, you have probably 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

Your next task is to bring up the virtual memory system for the kernel.

- Allocate some memory for the kernel's page table. This must be an array of size `MAX_PT_LEN` where the elements are of type `struct pte` (described on p. 11 of the handout). Hint: the page table exactly fits into a page of physical memory.
- Map the kernel's "text segment" so that it can be read and executed (protection mode `PROT_READ | PROT_EXEC`). You can use the value of `&etext` to determine the address of the top of the kernel's text segment.
- Map the kernel's "data" segment so that it can be read and written. You can use the `end` symbol to determine the address of the top of your kernel's data segment. You can also assume that the bss/data segment starts at the page boundary immediately above the end of the text segment.
- Map any other pages you might have allocated for the free page list and kernel page table so that they can also be read and written.
- Unmap all other pages so that they are invalid. At this stage, these pages should correspond to the pages on your free memory list.
- Tell the hardware where your kernel page table is located by writing the physical memory address of the page table to `REG_PTBR0` and the value `MAX_PT_LEN` to `REG_PTLR0`.
- Enable the virtual memory system by writing to the `REG_VM_ENABLE` register.
- Implement kernel memory management functions so that your kernel can perform dynamic memory allocation. You are free to do this any way that you wish. However, the easiest memory allocator to write is a page allocator in the form of two functions:
  - `void *pmalloc(int npages)`. Allocate npages of memory and return a pointer to their virtual memory address.
  - `void pfree(void *ptr, int npages)`. Release npages of memory.

  The main project handout has a few details on how you might implement these functions. The key point is that these functions return pointers to a virtual memory address in kernel space (Region 0). You should use the physical memory functions in the previous section to allocate and deallocate memory.
- Test your page allocator functions by modifying one of your trap handlers to allocate memory as before. For instance, in the `TRAP_PEN` handler, you could write code similar to the following:

```
{
    int       i;
    char     *c;
    c= pmalloc(1);       /* Allocate 1 page of memory */
    if (!c) {
      kprints("Out of memory!\n");
    } else {
      /* Overwrite the memory */
      for (i = 0; i < PAGESIZE: i++) {
          c[i] = 0;
      }
    }
}
```

Now, start your kernel and start generating `TRAP_PEN` events. You should see your kernel allocating memory and you should eventually see the "Out of memory" message. If your kernel crashes, your `pmalloc()` function is probably updating the kernel page table incorrectly or something else is broken.

- The `pmalloc()` function is extremely inefficient for small memory allocations because it consumes memory in increments of size PAGESIZE. If you plan to do lots of small memory allocations in your kernel, you might want to write the following memory allocation functions:

  - `void *kmalloc(int nbytes)`. Allocate nbytes of memory and return a pointer to their virtual memory address.
  - `void kfree(void *ptr)`. Release nbytes of memory previously allocated with kmalloc().

  Since writing these functions is somewhat complicated, you may want to alias kmalloc() to your page allocator when starting out. You can always improve the implementation later. Also, depending on how you have structured your kernel, you might be able to do everything using the simpler page allocator.

1.3 Your First System Call

Now, it's time to write the first system call for your kernel. To do this, you will implement the `ConsoleWrite()` system call so that user processes can write output to the kernel debugging console. Recall that `ConsoleWrite()` has the following prototype in user programs:

- `int ConsoleWrite(char *buffer, int len)`

Inside the kernel, this function may be mapped to a function such as the following (from project 1):

- `void SysConsoleWrite(cpu_context_frame *frame)`

To implement the system call, do the following:

- Read the system call parameters from `frame->regs[0]` and `frame->regs[1]` respectively.
- Use your `kputchar()` function implemented in project 1 to write the characters in `buffer` to the console.
- Put the syscall return value `len` in `frame->regs[0]` and return.

It is important to note that this system call is not directly callable from the kernel. Instead, it will be accessed by user programs through the `TRAP_KERNEL` handler. Because of this, it is a little difficult to test the system call until you have processes working. However, if you feel an overwhelming desire to test it before proceeding, here is one way to do it:

- Remove the -pedantic-errors compiler option from the kernel Makefile.
- Rewrite your kernel idle process so that it looks exactly like this:

```
void KernelIdle() {
  int ConsoleTest(char *buffer, int len);
  while (1) {
    kprints("Idling\n");
    ConsoleTest("Hello\n",6);   /* Execute ConsoleWrite syscall using */
    Pause();                    /* the hack below */
  }
  /* Never reached by the above code */
  asm("ConsoleTest:");
  asm("ld   [%sp], %g1");
  asm("mov  31, %g1");          /* sycall 31 = ConsoleWrite */
  asm("ta   16");               /* trap to kernel           */
  asm("retl");
  asm("nop");
}
```

- Recompile and run. You should see your `TRAP_KERNEL` handler being called, your `SysConsoleWrite()` function being called, and the message “Hello” being printed.
- Don’t forget to remove all of the above testing code after you have finished pondering the mysterious assembly code–you won’t need it any further.
- Please put the `-pedantic-errors` option back—it will help you find really nasty problems in your kernel.

1.4 Processes

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

- Define a process control block (PCB) data structure 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.

- Write a simple user test program similiar to the in-kernel idle process that idles and prints a message using the `ConsoleWrite()` system call. 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 kernel `main()`, 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:
  - During kernel boot, load `init1` and `init2` as you did for `init` above.
  - During the first timer interrupt, context switch from `init1` to `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). Note: the size of the user stack is limited to `MAX_STACK` bytes.

- 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.

1.5 Miscellaneous System Calls

If you have time, start working on the following system calls after you are done with everything else in this part of the project:

- `GetPid()`
- `Delay()`
- `Yield()`
- `DisplayWrite()`
- `SerialWrite()`

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:

- When setting up page tables, the hardware expects to receive physical memory addresses that point to the start of each page table. Since the page tables are accessed by the kernel in virtual memory, you will need to translate the virtual address of the page table to a physical address. Hint: all of this information should be available in the kernel page table.
- 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.

- 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 `/u5/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 many 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.