CS58, Dartmouth College

The Yalnix Project

Sean Smith, Dartmouth College

Revising and extending material by Dave Johnson, Rice University
Additional material from Adam Salem, Dartmouth College

Version of October 2, 2015

Contents

List of Figures

Chapter 1

Overview

1.1 Project Overview

Through this assignment, you will be able to learn how a real operating system kernel works, managing the hardware resources of the computer system and providing services to user processes running on the system. In the project, you will implement an operating system kernel for the Yalnix operating system, running on a fictional computer system known as the DCS 58.

Through the magic of the support software provided for your use in this project, a user-level program running in 32-bit mode on the Linux/x86 machines on the VirtualBox image we will release will be made to behave like an DCS 58 computer for you to run your Yalnix kernel on. That is, when you run your kernel, it will appear that you are running your own operating system on a real DCS 58 computer, but in reality, everything will be running in user mode in processes on the lab machines. Separate X terminal windows will also be used to simulate the terminals attached to the DCS 58 computer.

Yalnix supports multiple processes, each having their own virtual address space. Because of the magic of the simulation, your final system will run user-level programs at a fairly reasonable speed, and permit them to be linked to standard library functions.

This Document This document gathers, in one place, most of the information you need for the project.

Producing a document like this suffers from a tension:

- One wants to make it easily readable.
- But one also wants to make it as complete and precise as possible.

The manual's organization attempts to balance these constraints.

Chapter 1 gives an overview of the project and its basic logistics.

Chapter 2 discusses how to approach the project.

Chapter 3 describes the fictional computer hardware on which your OS must run.

Chapter 4 specifies the OS itself: the system calls and other behavior you must support.

Chapter 5 then gives more hints and suggestions for completing the project.

Resources for the Yalnix project live at **`/yalnix/`** in the vbox image. See the **`README`** there for a guide to what's present.

1.2 Specifics

1.2.1 Environment

- Your solutions must be implemented in C.
- Your solutions must run on our VirtualBox image, using the supplied DCS 58 computer system simulation environment, as described in Chapter 3.
- Your kernel must provide the interface and features described in Chapter 4.
- The project must be done in groups of two students.

  You will need to divide the load between the group members.

  However, make sure that you each know what the other is doing, and think ahead to design data structures and approaches that can be shared.

  In order to get the full benefit of the course, all partners in a project group should fully understand the group's solution to the project. *Be sure to test your code thoroughly, and make sure that your implementations adhere to the "API contract".*

  Be sure that all group members do their fair share of work.

  All group members are responsible for understanding how all parts of the kernel work.
- The course staff will set up code repositories for each team. (Besides making this project easier, using version control is a good habit for when you get into the real world.) This is also how you will submit your work.

1.2.2 Form

- Per the specifications in Section 5.7.3, your **`yalnix`** program should look to the command line for the name of the userland executable to run as the init process—and use the default "init" if no option is provided.

- You should use reasonable variable names and include comments in particularly tricky parts of the code, but you need not document code that does what it looks like.

- As noted above, the project requires some special libraries and other tools. The VirtualBox image has been appropriately configured and tested for this.

- Your project directory must include a **`Makefile`** that can be used to make your kernel and any Yalnix userland test programs you used in testing your kernel. A template **`Makefile`** is available as **`/yalnix/sample/Makefile`**. You should copy this file to your project directory and edit it as described in the comments in the file. *In particular, this template contains special rules for compiling and linking programs that must be used for the project to work.*

- Everything that you want us to grade should be in your team's repository.

- In the directory containing everything you want us to grade, the **`README`** file should give a quick guide to the various files.

  In the other directories (in the accounts *not* containing everything you want us to grade), your **`README`** file should just contain a simple statement saying where the real files are.

1.2.3 Etiquette

Your code may end up using lots of CPU cycles, and also (if it terminates nongracefully) generating lots of Linux processes called **`yalnixtty`**, **`yalnixnet`**, and **`yalnix.`**

Use **`make kill`** to check for and kill any spurious processes you may have created.

Use **`make no-core`** to remove any core files you may have lying around.

1.2.4 Grading

This project will be graded on correctness, efficiency, elegance, clarity, and general slickness.

We will read your source code. We will test basic functionality. We will also stress-test it. The directory **`/yalnix/sample/test/`** has a few user-level programs that will stress things. (We use these, and others, in grading.)

Throughout, a few opportunities may present themselves for features (e.g., sharable executable code) that go beyond the basic requirements of the project. Although we will look favorably on such innovations, you should not consider such extras until you have met all the basic requirements.

However, students taking the course for graduate credit will be expected to successfully implement some additional innovation.

1.2.5 Project Checkpoints

This is a large project, with a significant amount of time to work on it before the due date.

We offer the following advice to help you work your way successfully through the project:

- Start working on the project early—such as right now!
- Try to work on the project consistently throughout this time.

If you have any questions, ask the instructor or a TA.

As a further aid in helping you complete the project by the due date, the schedule for this project includes *project checkpoints* as intermediate milestone points within the project.

Note: the checkpoints are *minimum* levels of work you should achieve. (See also Section 5.11.)

- ***Checkpoint 1:*** By this point, you should have:
  - sketched all the kernel data structures
  - pseudo-coded all the traps, syscall, and major functions
  - (and gotten the VirtualBox image installed, and figured out how to move files back and forth between your host and the guestOS via shared folders).

  You need the data structures to start the real coding, and, from our experience, if you don't work through the flow of *all* the code, you won't have the right data structures. Fixing that later will be harder than fixing it now. These sketches should all be done in real source files—as comments, and potentially with data structures, function stubs, and prototypes.

- ***Checkpoint 2:*** `KernelStart` runs, initializing the kernel, the machine, the interrupt vector, and the page tables. An *idle* process should run.

  For this idle, you should a write simple idle function in the kernel text.

```
void
DoIdle(void) {
```

```
    while(1) {
      TracePrintf(1,"DoIdle\n");
      Pause();
    }

}
```

Cook things (e.g., the UserContext) so that when you return to user mode, you execute this code. This approach removes the need to get **`LoadProgram`** done first—thus making your life easier.)

Enough memory management code should be complete to enable the above to work: such as **`SetKernelBrk`**, enabling virtual memory, and a skeletal **`TRAP_MEMORY`**.

- ***Checkpoint 3:*** **`LoadProgram`** should work, enabling loading a simple *init* program. You should be able to context switch between *idle* and *init* processes. The **`GetPid`**, **`Delay`**, and **`Brk`** syscalls should be implemented.
- ***Checkpoint 4:*** The **`Fork`**, **`Exec`**, and **`Wait`**, syscalls should work. Some of the traps should be implemented.
- ***Checkpoint 5:*** The **`TtyRead`** and **`TtyWrite`** syscalls should be complete. The remaining traps and exceptions should be implemented.
- ***Checkpoint 6:*** The remaining syscalls—and all code—should be complete. Begin thorough testing.
- ***Checkpoint 7:*** Project complete.

If you are well ahead of schedule, you might think about porting your Ledyard Bridge code to run on Yalnix. This will require a few innovations, since you need to figure out how to share the bridge state among the processes.

If you want, you can specify and implement some new syscalls. The simulation code includes stubs for the **`Custom0`**, **`Custom1`**, and **`Custom2`** syscalls, just for this purpose. We've provided the **`calls.c`** file, as a reference for how the syscall wrappers are implemented. (If you inspect the code, you'll see that there also stubs for several unimplemented calls; you may use these too.)

1.3 Final Notes

1.3.1 Brief History

The original Yalnix support software was developed by Dave Johnson, for SunOS/SPARC, and is used at a number of universities.

I ported it to Linux/x86, so it could run on our Sudi Linux machines. (Evan Knop, Dave Johnson, and Tim Tregubov all provided assistance.) Other universities have started adopting our version.

The current version has new heap code (custom-written by me) and a new C library (which results in much smaller statically linked executables).

Yalnix (our version) differs from the original Yalnix in two ways:

- I have eliminated the suite of message-passing syscalls in the original Yalnix system. (The stubs still exist in the code.)
- But I have added standard synchronization primitives, to coordinate these processes. I also added pipes.

The original message-passing calls were tricky and not directly relevant to the course material; synchronization is less tricky and more relevant.

1.3.2 The Doughnut Challenge

The Yalnix simulation software is non-trivial, and uses sleight-of-hand that depends on details of the host OS and host CPU.

Although the code has been well-tested, there's always a chance that bugs may still be lurking. (For example, porting it to the Linux platform revealed a few bugs that had been there all along; more recently, a few bugs have been found in the new heap code).

Hence, a challenge:

- If, during this project, you find a bug in the underlying support software and build environment, then I will buy you a doughnut.
- But if the problem turns out to be in your code, then you must buy me a doughnut.

Discovery of significant bugs in the documentation will be similarly rewarded.

1.3.3 Honor Code

A growing number of universities are using Yalnix in their OS courses. It is possible that you might find Yalnix materials from students at these universities. It might also be possible find material and help from students who have previously taken CS58 at Dartmouth.

Using such resources would be considered a violation of the Honor Code.

To make it easier for future students to follow the Honor Code, please do not publicly distribute your code.

Of course, you are always free to discuss Yalnix with myself and the TAs.

Chapter 2

Approaching the Project

2.1 The Big Picture

This document contains a lot of material. To help you digest it, let's start by reviewing some of the concepts we've discussed in class, as applied to this project. (Chapter 5 will provide more detailed discussion of these concepts.)

Your kernel runs on the (somewhat simulated) hardware described in Chapter 3. The hardware provides standard features such as user mode, kernel mode, memory management (including TLB), interrupts and traps, and I/O.

Since your OS never actually runs as a freestanding program, your code will have no **`main()`**. Instead, it reacts:

- At boot time, the hardware invokes **`SetKernelData`** to tell your kernel some basic parameters about the data segment.
- Later during boot, the hardware invokes **`KernelStart`**. In this call, the kernel completes its initialization and sets up the first userland process.
- For interrupts, exceptions, and system calls, the hardware will suspend execution of the current userland process, switch to kernel mode, and invoke the appropriate handler function (from the interrupt vector) to see what it should run.
- At any time, the kernel's **`malloc`** library may turn around and call **`SetKernelBrk`** in order to adjust the kernel's heap break.

You will write all these call handlers as part of your kernel.

When calling **`KernelStart`** and the trap handlers, the hardware passes a pointer to a **`UserContext`**. For the trap handlers, this location contains a copy of the user context of the interrupted userland process. When the kernel returns from **`KernelStart`** or a trap handler:

- The hardware resumes running in user mode
- from whatever user context is now at that location
- and whatever address space mapping is currently provided by the MMU and TLB.

(**`SetKernelData`** quietly returns to the hardware start-up; **`SetKernelBrk`** returns to the kernel library.)

Figure 2.1 sketches this flow.

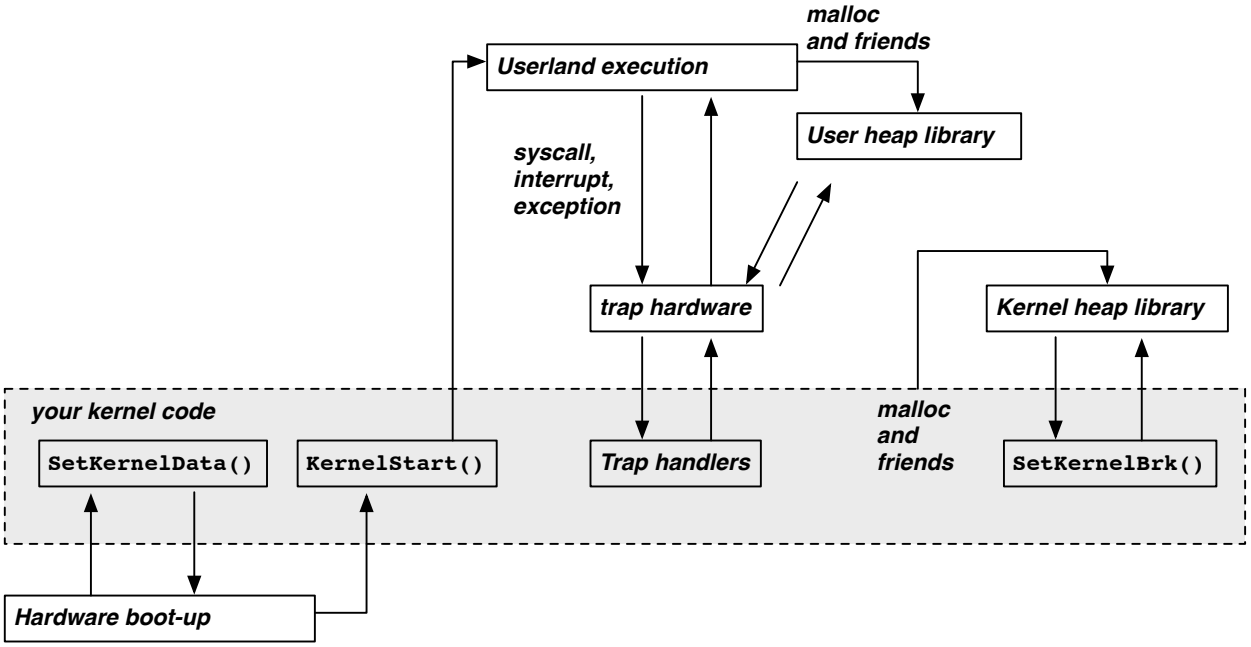

Figure 2.1: Your kernel consists of a series of calls handlers invoked under various circumstances. (Note also that the kernel may use **`KernelContextSwitch()`** to suspend one kernelland process and start/resume another).

What you need to do is think about the meaning of the system calls described in Chapter 4, in light of the traps described in Chapter 3. What data structures do you need to support these calls? What operations? How do you distribute these operations in response to the various system calls and other traps the hardware sends your way?

As we have mentioned in class, the hardware automatically disables all interrupts when the CPU is in kernel mode (thus making your life much easier).

2.2 Yalnix Virtual Memory Layout

The Yalnix operating system uses the virtual memory management hardware provided by the DCS 58 computer system to implement a separate virtual address space for each process, with a kernel shared by all processes (as we discussed in class).

The layout of the virtual address space implemented by Yalnix is shown in Figure 2.2.

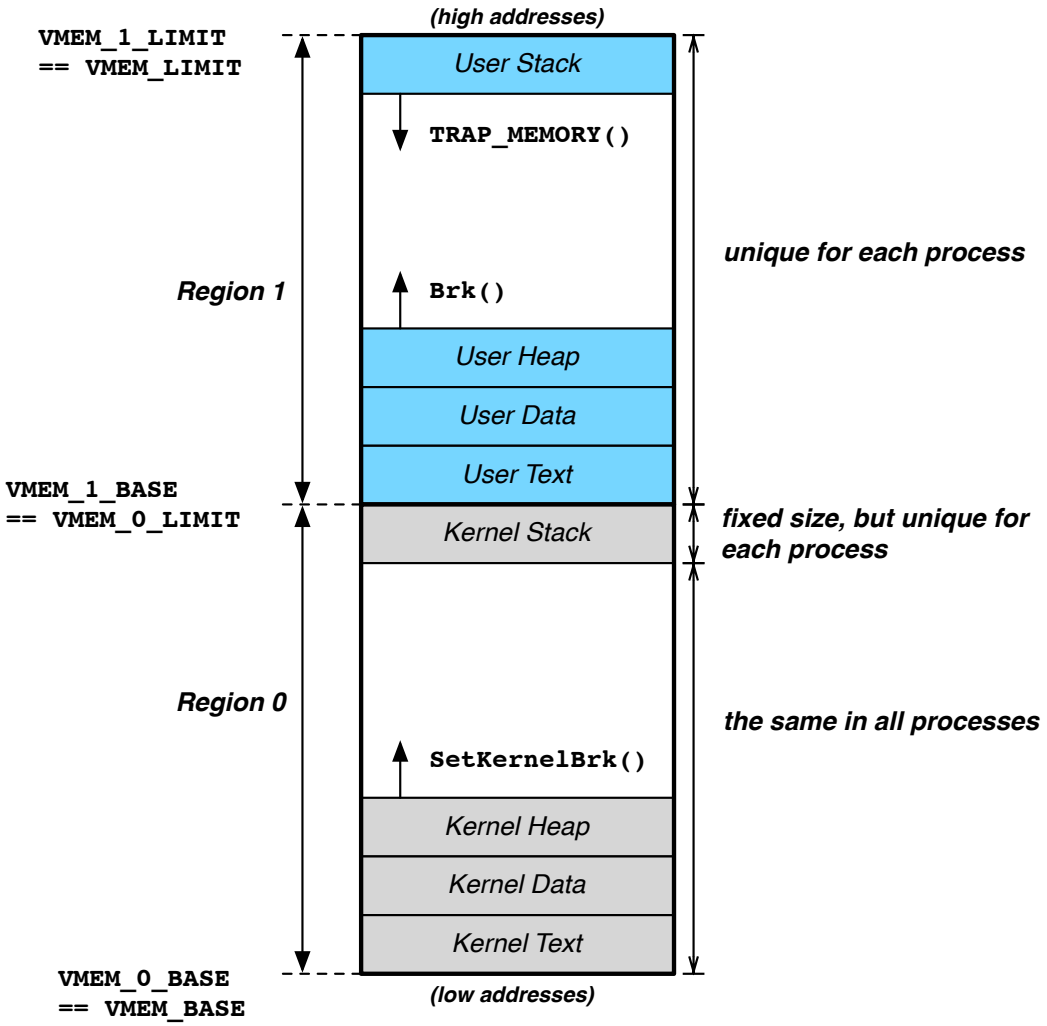

Figure 2.2: The virtual address space layout in Yalnix. Each process has its own Region 1. Each process also has its own kernel stack, constant size. However, there is only one kernel, shared by all processes.

As Figure 2.2 shows, the virtual address space of the machine is divided into two regions: *Region 0* is visible only in kernel mode; *Region 1* is the user space.

User Space The user space contains two contiguous allocated regions: the text, global data, and heap at the bottom, and the stack at the top The limit of the text/data/heap area (the lowest address above it that is not part of it) is called the process's *break* and is set from Yalnix user processes by the **`Brk`** kernel call, which you will implement in your kernel.

The break of user processes is initialized (when a program begins running) to an address defined in the executable program file from which the process was loaded. In **`template.c`**, we provide a template for you use to figure this all out.

After initialization, the process itself allocates more memory by calling the kernel's **`Brk`** kernel call.

In the library with which your Yalnix *user* programs are linked by the provided **`Makefile`**, we provide versions of the familiar **`malloc`** family of procedures. When they feel it's necessary, these library functions will in turn call your kernel's **`Brk`** kernel call. Thus, while your kernel must manage each user process's break via the calls by that process to **`Brk`**, the Yalnix user programs themselves can call **`malloc`**, **`realloc`**, **`free`**, etc., as if written for a normal Linux system. The memory that they use is controlled by your kernel, though, not by Linux.

Growing the stack area is something that your own Yalnix kernel will need to handle, when there's a **`TRAP_MEMORY`** exception resulting from an access to a page between the current break and the current stack pointer that has not yet been allocated. This happens during normal execution as the program makes procedure calls that automatically expand the stack downwards.

Kernel Space. The kernel also has space for a text, data, heap, and a fixed-size kernel stack. The text,data, and heap are shared by all processes when in kernel mode. However, each process has its own kernel stack.

As noted above, the kernel library uses **`SetKernelBrk()`** to increase its break.

For More Details. Chapter 4 contains more details on these operations.

2.3 Address Translation

As we will discuss in class, the hardware's MMU handles address translation and typically dictates the page table format. Yalnix on the DCS 58 is no exception.

Chapter 3 specifies how the page table should be formatted, and how the OS can govern memory management by writing to MMU control registers.

The DCS 58 MMU has a TLB. On this machine, the TLB will pin the entries corresponding to the common parts of Region 0.

2.4 Context Switching

As we also discussed in class, one of the initial mysteries of an OS is context switching. A syscall or interrupt will cause the machine to suspend execution of its user program and switch to kernel mode; in some scenarios, the OS then wants to suspend the current process (in kernel mode) and switch to another process (also in kernel mode).

To help with these mysteries, we provide some special data structures:

- The **`UserContext`** structure (defined in **`hardware.h`**) contains a copy of the hardware state of the currently running process, as saved by the hardware when the interrupt, exception, or trap occurred. Your kernel might need to work with the internals of this structure now and then.
- The **`KernelContext`** structure (also defined in **`hardware.h`**) is full of low-level simulation-specific data. You don't need to worry about what this data means, but your kernel code will need to keep track of the structure for bookkeeping.

Switching, during kernel mode, from one process (with its kernel stack and context) to another is somewhat like changing the tires on your bicycle while riding it: possible, but tricky. To make this easier, we provide a **`KernelContextSwitch`** function that provides a magic place, free of either process's stacks and contexts, to run a specified function on specified arguments.

Chapter 3 describes the user context; Chapter 4 provides some discussion of when to work with both kernel and user context. Chapter 5 describes how to use **`KernelContextSwitch`**.

2.5 I/O

As described in Chapter 3, the hardware interface to the terminals is through the **`TtyTransmit`** and **`TtyReceive`** hardware instructions. Only your kernel can execute these instructions.

As described in Chapter 4, your Yalnix kernel must provide kernel calls **`TtyWrite`** and **`TtyRead`** to allow user programs access to the terminals.

As a convenience, we also provide a library routine that can be used by Yalnix *user* processes:

```
int TtyPrintf(int, char *, ...)
```

that works similarly to the standard C library **printf** function, but does its output using the Yalnix **TtyWrite** kernel call intead. The first argument to **TtyPrintf** is the Yalnix terminal number to which to write. The next argument is a standard **printf**-style C format string, and any remaining arguments are the values for that format string to format. The return value from **TtyPrintf** is the value returned by your kernel from the underlying **TtyWrite** call performed by **TtyPrintf**. The maximum length of formatted output that can be written by a single call to **TtyPrintf** is **TERMINAL_MAX_LINE** bytes (defined in **hardware.h**). (If you try something bigger, the behavior is proverbially "undefined.")

Figure 2.3 gives a big-picture sketch of the I/O subsystem. Section 3.3.1 below will discuss the bottom-half interaction; Section 4.1 will discuss the top-half interaction.

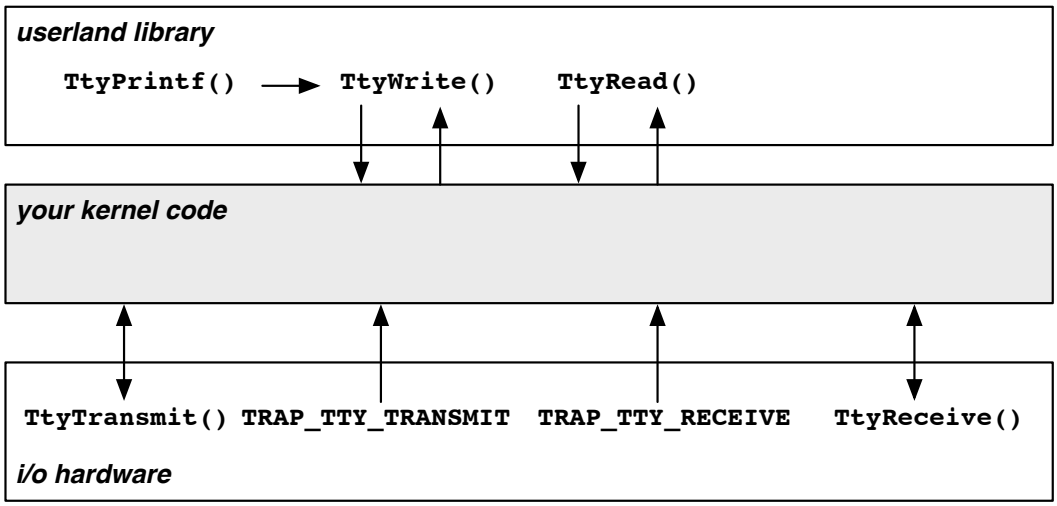

Figure 2.3: The userland library invokes I/O syscalls, which your kernel must implement via calls to the I/O hardware and handling of the traps the hardware throws.

Chapter 3

Basic Hardware Specification

This chapter describes the hardware architecture of the Dartmouth Computer Science DCS 58 computer, a fictional computer architecture on which you will implement an operating system kernel in this project. The DCS 58 computer system hardware is similar to the x86 computer architecture but is simplified somewhat to make the project more manageable.

This chapter is written generally in the style of a real computer system hardware architecture manual. Everything in this chapter is something that a real operating system designer and implementor on such a computer would need to know about the hardware. Certain features of the operating system running on top of this hardware are mentioned here only as guidance in the implementation of any operating system for this hardware.

3.1 Machine Registers

This section defines the general purpose registers and privileged registers of the DCS 58 computer system. All registers on the DCS 58 computer are 32 bits wide.

3.1.1 General Purpose Registers

The DCS 58 computer has a number of general purpose registers that may be accessed from either user mode or kernel mode. These general purpose registers include the following:

- **`PC`**, the *program counter*, contains the virtual address from which the currently executing instruction was fetched.
- **`SP`**, the *stack pointer*, contains the virtual address of the top of the current process's stack.

Table 3.1: DCS 58 Privileged Register Summary.

Name	Purpose	Readable	Details
REG_PTBR0	Page Table Base Register 0	Yes	Section 3.2.4
REG_PTLR0	Page Table Limit Register 0	Yes	Section 3.2.4
REG_PTBR1	Page Table Base Register 1	Yes	Section 3.2.4
REG_PTLR1	Page Table Limit Register 1	Yes	Section 3.2.4
REG_TLB_FLUSH	TLB Flush Register	No	Section 3.2.5
REG_VM_ENABLE	Virtual Memory Enable Register	Yes	Section 3.2.6
REG_VECTOR_BASE	Vector Base Register	Yes	Section 3.4.1

- **EBP**, the *base pointer*, contains the virtual address of the current process's stack frame.
- **R0** through **R7**, the eight *general registers*, used as accumulators or otherwise to hold temporary values during a computation.

In reality, the real hardware has quite a few additional general purpose registers, but since you will be writing your operating system in C, not in machine language or assembly language, only these registers are relevant to the project.

3.1.2 Privileged Registers

In addition to the general purpose CPU registers, the DCS 58 computer contains a number of privileged registers, accessible only from kernel mode. Table 3.1 summarizes these privileged registers.

Details for the use of each of these registers are described in the following sections, as indicated in Table 3.1, where the operation of that component of the DCS 58 computer system hardware architecture is defined. Most of the privileged registers are both writable and readable, except for the **REG_TLB_FLUSH** register, which is write-only.

The hardware provides two privileged instructions for reading and writing these privileged machine registers:

- **void WriteRegister(int which, unsigned int value)**

  Write **value** into the privileged machine register designated by **which**.

- **unsigned int ReadRegister(int which)**

  Read the register specified by **which** and return its current value.

Each privileged machine register is identified by a unique integer constant as noted in Table 3.1, passed as the **which** argument to the instructions. The file **hardware.h** defines the symbolic

names for these constants. In the rest of the document we will use only the symbolic names to refer to the privileged machine registers.

The values of these registers are represented by these two instructions as values of type **`unsigned int`** in C. You must use a C "cast" to convert other data types (such as addresses) to type **`unsigned int`** when calling **`WriteRegister`**, and must also use a "cast" to convert the value returned by **`ReadRegister`** to the desired type if you need to interpret the returned value as anything other than an **`unsigned int`**.

3.2 Memory Subsystem

3.2.1 Overview

The memory subsystem of the DCS 58 computer supports physical memory size that depends on the amount of hardware memory installed. The physical memory is divided into frames of size **`PAGESIZE`** bytes.

The individual frames of this physical memory may be mapped into the address space of a running process through virtual memory supported by the hardware and initialized and controlled by the operating system. As is standard, we have a *Memory Management Unit (MMU)* implementing the hardware control for this virtual memory mapping. As with the physical memory, the virtual memory is divided into pages of size **`PAGESIZE`**, and any page of virtual memory may be mapped to any frame of physical memory through the *page table* of each running process.

The constant **`PAGESIZE`** is defined in **`hardware.h`**. In addition, **`hardware.h`** defines a number of other constants and macros to make dealing with addresses and page numbers easier:

- **`PAGESIZE`**: The size in bytes of each physical memory frame and virtual memory page.
- **`PAGEOFFSET`**: A bit mask that can be used to extract the byte offset with a page, given an arbitrary physical or virtual address. For an address **`addr`**, the value **`(addr & PAGEOFFSET)`** is the byte offset within the page where **`addr`** points.
- **`PAGEMASK`**: A bit mask that can be used to extract the beginning address of a page, given an arbitrary physical or virtual address. For an address **`addr`**, the value **`(addr & PAGEMASK)`** is the beginning address of the page where **`addr`** points.
- **`PAGESHIFT`**: The log base 2 of **`PAGESIZE`**, which is thus also the number of bits in the offset in a physical or virtual address. You can use **`PAGESHIFT`** to turn page numbers into addresses, and vice-versa. E.g., **`r0page`**<<**`PAGESHIFT`** takes a Region 0 page number to the 0th address in that page.

- **`UP_TO_PAGE(addr)`**: A C preprocessor macro that rounds address **`addr`** up to the 0th address in the next highest page—in other words, the next highest page boundary. But If **`addr`** is already the 0th address of a page, it returns **`addr`**.

- **`DOWN_TO_PAGE(addr)`**: A C preprocessor macro that rounds address **`addr`** down to the next lowest page boundary (the 0th address of the next lowest page). But if **`addr`** is on a page boundary, it returns **`addr`**.

3.2.2 Physical Memory

The beginning address and size of physical memory in the DCS 58 computer system are defined as follows:

- **`PMEM_BASE`**: The physical memory address of the first byte of physical memory in the machine. This address is a constant and is determined by the machine's hardware design. The value **`PMEM_BASE`** is defined in **`hardware.h`**.

- The total size (number of bytes) of physical memory in the computer is determined by how much RAM is installed on the machine. At boot time, the computer's firmware tests the physical memory and determines the amount of memory installed, and passes this value to the initialization procedure of the operating system kernel.

As described in Section 3.2.1, the physical memory of the machine is divided into frames of size **`PAGESIZE`**. Frames numbers in physical memory addresses are often referred to as *page frame numbers*.

Except for also being a multiple of **`PAGESIZE`** bytes, ***The size of physical memory is unrelated to the size of the virtual address space of the machine***.

3.2.3 Virtual Address Space

The virtual address space of the machine is divided into two regions, called *Region 0* and *Region 1*. By convention, Region 0 is used by the operating system kernel, and Region 1 is used by user processes executing on the operating system. Your kernel will manage Region 0 to hold kernel state and a Region 1 mapping for each process.

This division of the virtual address space into two regions is illustrated in Figure 2.2, back in Chapter 2.

- The beginning (lowest) virtual address of Region 0 is defined as **`VMEM_0_BASE`**.

- The limit virtual address of Region 0 (that is: first byte *above* the end of Region 0) is defined as **VMEM_0_LIMIT**
- the size of Region 0 is defined as **VMEM_0_SIZE**.
- the beginning virtual address of Region 1 is defined as **VMEM_1_BASE**,
- the limit virtual address (first byte above the region) is **VMEM_1_LIMIT**,
- and the size is **VMEM_1_SIZE**.

By the definition of the hardware, the two regions are adjacent: **VMEM_1_BASE** equals **VMEM_0_LIMIT**. The overall beginning virtual address of virtual memory is **VMEM_BASE** (which equals **VMEM_0_BASE**), and the overall limit virtual address of virtual memory is **VMEM_LIMIT**. (**VMEM_LIMIT** equals **VMEM_1_LIMIT**). (Again, see Figure 2.2.)

The state of the kernel in Region 0 consists of two parts: the kernel code and global variables. This state is the same, independent of which user-level process is executing. The kernel stack however, holds the local kernel state associated with the user-level process that is currently executing.

On a context switch between processes, you need to change the mappings for the kernel stack and all of Region 1.

The hardware provides the two regions so that it can protect kernel data structures from illegal tampering by the user applications: when the CPU is executing in kernel mode, references to addresses in either region are allowed, but when the CPU is executing in user mode, references to addresses in Region 0 are not allowed. The hardware is specified to enforce this restriction.

3.2.4 Page Tables

As we discussed in class, the MMU hardware automatically translates each reference (use) of any virtual memory address into the corresponding physical memory address. This translation happens without intervention from the kernel, although the kernel does control the mapping that specifies the translation.

The hardware uses a direct, single-level page table to define the mapping from virtual to physical addresses. Such a page table is an array of page table entries, laid out contiguously in memory.[1] Each page table entry contains information relevant to the mapping of a single virtual page to a corresponding physical page.

The kernel allocates page tables in memory wherever it wants to, and it tells the MMU in the hardware where to find these page tables through the following privileged registers:

[1] In a real system, this would be contiguous physical memory—unless we were doing some clever multi-level scheme. In our simulation, this will be continuous virtual memory.

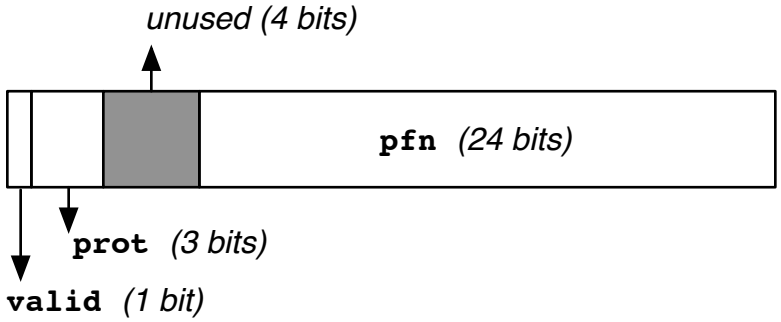

Figure 3.1: DCS 58 Hardware Page Table Entry (PTE) Format. Note that not all bits in the word are used. Unused bits may be used by your OS for its own nefarious purposes.

- **REG_PTBR0**: Contains the *virtual* memory base address of the page table for Region 0 of virtual memory.
- **REG_PTLR0**: Contains the number of entries in the page table for virtual memory Region 0, i.e., the number of virtual pages in Region 0.
- **REG_PTBR1**: Contains the *virtual* memory base address of the page table for Region 1 of virtual memory.
- **REG_PTLR1**: Contains the number of entries the page table for virtual memory Region 1, i.e., the number of virtual pages in Region 1.

Note that our simulation requires virtual addresses in these page table registers! (More discussion follows later.)

The kernel changes the virtual-to-physical address mapping by changing these registers (e.g., during a context switch). The kernel may also change any individual page table entry by modifying that entry in the corresponding page table in memory.

When the CPU makes a memory reference to a virtual page **vpn** (virtual page number), the MMU finds the corresponding page frame number **pfn** by looking in the page table for the appropriate region of virtual memory. The page tables are indexed by virtual page number, and they contain entries that specify the corresponding page frame number (among other things) of each page of virtual memory in that region.

For example, if the reference is to Region 0, and the first virtual page number of Region 0 is **vp0**, the MMU looks at the page table entry that is **vpn** - **vp0** page table entries above the address in **REG_PTBR0**. Likewise, if the reference is to Region 1, and the first virtual page number of Region 1 (i.e., **VMEM_1_BASE**>>**PAGESHIFT**) is **vp1**, the MMU looks at the page table entry that is **vpn** - **vp1** page table entries above the address in **REG_PTBR1**.

This lookup—to translate a virtual page number into its currently mapped corresponding physical page frame number—is carried out wholly in hardware. You will notice that in carrying out this

lookup, the hardware needs to examine page table entries in order to index into the page table (the size of a page table entry must be known to the hardware) and also to extract the page frame number from the entry (the format of a page table entry must also be known to the hardware).

The format of the page table entries is in fact dictated by the hardware. A page table entry is 32-bits wide (but does not use all 32 bits). It contains the following defined fields:

- **`valid`** (1 bit): If this bit is set, the page table entry is valid; otherwise, a memory exception is generated when/if this virtual memory page is accessed.
- **`prot`** (3 bits): These three bits define the memory protection applied by the hardware to this virtual memory page. The three protection bits are interpreted independently as follows:
  - **`PROT_READ`**: Memory within the page may be read.
  - **`PROT_WRITE`**: Memory within the page may be written.
  - **`PROT_EXEC`**: Memory within the page may be executed as machine instructions.

  As is standard programming practice, each of these constants consists of the integer obtained by setting a particular bit to one and the rest to zero. You can thus combine privileges by bitwise-OR'ing constants together.

  Execution of instructions requires that their pages be mapped with both **`PROT_READ`** and **`PROT_EXEC`** protection set.
- **`pfn`** (24 bits): This field contains the page frame number (the physical memory page number) of the page of physical memory to which this virtual memory page is mapped by this page table entry. This field is ignored if the valid bit is off.

This page table entry format is illustrated in Figure 3.1 and is defined in the file **`hardware.h`** as a C data structure as a **`struct pte`**. This C structure has the same memory layout as a hardware page table entry and can be used in your operating system kernel.

As we will discuss in class, if you need additional per-page bookkeeping not accommodated by this structure, you might consider using a shadow page table.

Virtual vs. Physical *The use of virtual addresses in the page table base registers is one way this simulation departs from reality. In a real system, you'd have to use physical addresses here, and then grapple with the challenge of, when creating a new address space, how to find sufficient physically contiguous space for its page table.*

You might ask yourself "How can this simulation possibly work?" How can the MMU look up an entry in the page table entry, if it must look in the page table to find out where the page table is? Read on!

3.2.5 Translation Lookaside Buffer (TLB)

The DCS 58, as with most architectures supporting virtual memory mapping, contains a *translation lookaside buffer*, or *TLB*, to speed up virtual-to-physical address translation. The TLB caches address translations so that subsequent references to the same virtual page do not have to retrieve the corresponding page table entry anew. This caching is important because it can otherwise take several memory accesses to retrieve a page table entry (and then yet another to touch the memory location itself).

Page table entries are loaded automatically, as needed, into the TLB by the hardware during virtual address to physical address translation. The operating system cannot directly load page table entries into the TLB and cannot examine the contents of the TLB to determine what page table entries are currently present in the TLB.

However, the operating system kernel must, on occasion, flush all or part of the TLB. In particular, after changing a page table entry in memory, the TLB may contain a stale copy of this page table entry, since this entry may have been previously loaded into the TLB by the hardware. Also, when carrying out a context switch from one process to another, the operating system must flush the current contents of the TLB, since the page table entries currently cached in the TLB correspond to page table entries for the process being context switched out, not to the new process being context switched in.

Failing to flush a TLB entry can be a really fun bug to track down!

The hardware provides the **`REG_TLB_FLUSH`** privileged machine register to allow the operating system kernel to control the flushing of all or part of the TLB. When writing a value to the **`REG_TLB_FLUSH`** register, the MMU interprets the value as follows:

- **`TLB_FLUSH_ALL`**: Flush all entries in the entire TLB.
- **`TLB_FLUSH_0`**: Flush all entries in the TLB that correspond to virtual addresses in Region 0.

  However, virtual addresses that correspond to addresses in the original kernel space—text, data, initial heap when virtual memory is enabled—are not flushed in this operation, so you can safely put the page table for Region 0 here!

  Also note: flushing a kernel stack page from the region 0 TLB causes the kernel stack to be silently remapped, according to the current page table. So, when changing Region 0 contexts, be sure to change the page table first, then flush.
- **`TLB_FLUSH_1`**: Flush all entries in the TLB that correspond to virtual addresses in Region 1.
- Otherwise, the value written into the **`REG_TLB_FLUSH`** register is interpreted by the hardware as a virtual address. If an entry exists in the TLB corresponding to the virtual memory page into which this address points, flush this single entry from the TLB; if no such entry exists in the TLB, this operation has no affect.

Symbolic constants for the **TLB_FLUSH_ALL**, **TLB_FLUSH_0**, and **TLB_FLUSH_1** values are defined in **hardware.h**.

3.2.6 Initializing Virtual Memory

When the machine is booted, boot ROM firmware loads the kernel into physical memory and begins executing it. The kernel is loaded into contiguous physical memory starting at addresses **PMEM_BASE**.

When the computer is first turned on, the machine's virtual memory subsystem is initially disabled and remains disabled until initialized and explicitly enabled by the operating system kernel. In order to initialize the virtual memory subsystem, the kernel must, for example, create an initial set of page table entries and must write the page table base and limit registers to tell the hardware where the page tables are located in memory. Until virtual memory is enabled, all addresses used are interpreted by the hardware as *physical addresses*; after virtual memory is enabled, all addresses used are interpreted by the hardware as *virtual addresses*.

The hardware provides a privileged machine register, the Virtual Memory Enable Register, that may be set by the kernel to enable the virtual memory subsystem in the hardware. To enable virtual memory, the kernel executes:

```
WriteRegister(REG_VM_ENABLE, 1)
```

After execution of this instruction, all addresses used are interpreted by the MMU hardware only as virtual memory addresses. Virtual memory cannot be disabled after it is enabled (that is, without rebooting the machine).

As noted above, virtual memory cannot be enabled until the kernel has run for a while and initialized the page table entries and page table base and limit registers. However, enabling virtual memory this late (after the kernel has already started running) creates a problem. Before virtual memory is enabled, all addresses are interpreted as physical addresses, but after virtual memory is enabled, all addresses are interpreted as virtual addresses.

This means that, at the point at which virtual memory is first enabled, suddenly all addresses are interpreted differently. *Without special care in the kernel, this can cause great confusion to the kernel,* since by this time, many addresses are in use in the running kernel: for example, in the program counter and stack pointer, perhaps in some of the general registers, and in any pointer variables declared and used in the kernel.

To solve this problem, the simplest method is to arrange the initial page table entries in the kernel so that all addresses you've used so far actually still point to the same places after enabling virtual memory. This can be done by the kernel by arranging for the initial virtual address space layout to be the same as the physical address space layout, as illustrated in Figure 3.2. The hardware makes

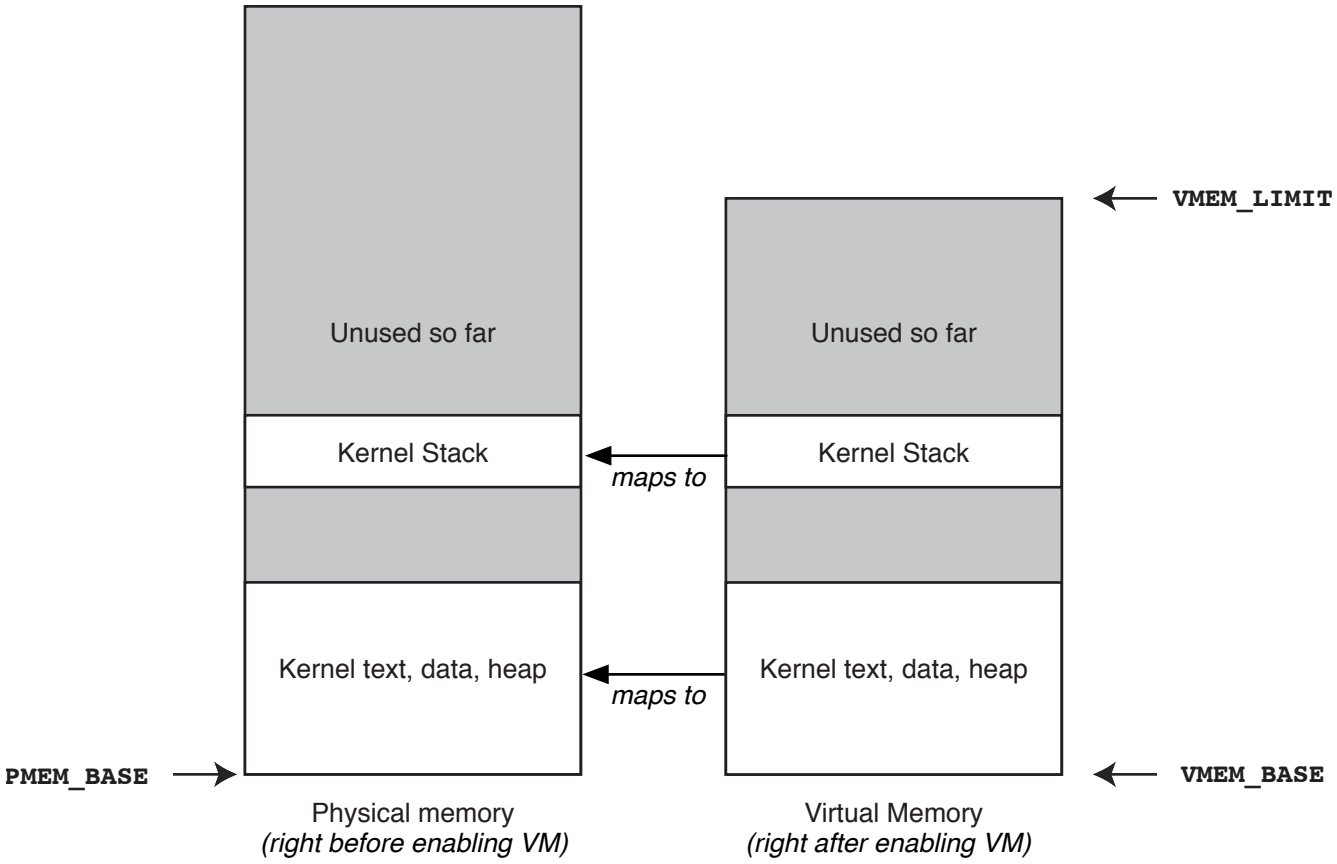

Figure 3.2: Initial memory layout before and after enabling virtual memory. If frame n is used used before VM is enabled, then page n needs to be valid and map to frame n—otherwise, surprising things may happen.

this possible by guaranteeing that **PMEM_BASE**, the beginning address of physical memory (the kernel is loaded at boot time starting at this physical address), is the same value as **VMEM_BASE**, the beginning address of virtual memory (which is thus the beginning address of Region 0 of virtual memory, where the kernel should be located once virtual memory is enabled). In building the initial page tables in the kernel before enabling virtual memory, the kernel should make sure, for every page of physical memory **pfn** then in use by your kernel, that a page table entry is built so that the new virtual page number **vpn** = **pfn** maps to physical page number **pfn**.

3.3 Hardware Devices

The DCS 58 computer system can be configured with a variety of I/O and other hardware devices. For purposes of this project, only two different types of devices will be available: a number of terminals and a hardware clock.

3.3.1 Terminals

The system is equipped with **NUM_TERMINALS** line-oriented terminals, numbered starting with 0, for use by the user processes. By convention, the zeroth terminal (terminal number 0) is considered to be the system console, while the other terminals are general-purpose terminal devices. However, this distinction is only a convention and does not imply an difference in behavior.

In a real system, the kernel would have to manipulate the terminal device hardware registers to read and write from the device. For simplicity, here we have abstracted the details into two C functions:

- **void TtyTransmit(int tty_id, void *buf, int len)**

  This function begins the transmission of **len** characters from memory, starting at address **buf**, to terminal **tty_id**.

  The kernel's invocation of **TtyTransmit** returns immediately—but the operation will take some time to actually complete. Consequently, the address **buf** must be in the kernel's memory, (i.e., it must be in virtual memory Region 0), since otherwise, this memory could become unavailable to the hardware terminal controller after a context switch.

  When the data has been completely written out, the terminal controller will generate a **TRAP_TTY_TRANSMIT** interrupt. When the **TRAP_TTY_TRANSMIT** fires, the terminal number of the terminal generating the interrupt will be made available to the kernel's interrupt handler for this type of interrupt. Until receiving a **TRAP_TTY_TRANSMIT** interrupt for this terminal after executing a **TtyTransmit** on it, no new **TtyTransmit** may be executed for this terminal.

- **int TtyReceive(int tty_id, void *buf, int len)** When the user completes entering an input line on a terminal by typing either newline (`'\n'`) or return (`'\r'`),

the hardware terminal controller will generate a **TRAP_TTY_RECEIVE** interrupt for this terminal (on input, both newline and return are translated to newline). The terminal number of the terminal generating the interrupt will be made available to the kernel's interrupt handler for this type of interrupt.

When this trap fires and indicates that data is ready to be read from the terminal, the kernel should then execute a **TtyReceive** operation for this terminal, in order to retrieve the new input line from the hardware. The new input line is copied from the hardware for terminal **tty_id**, into the kernel buffer at address **buf**, for maximum length to copy of **len** bytes. The buffer must be in the kernel's memory (i.e., it must be entirely within virtual memory Region 0).

The actual length of the input line, including the newline (`'\n'`), is returned as the return value of **TtyReceive**. Thus when a blank line is typed, **TtyReceive** will return a 1. When an end of file character (control-D) is typed, **TtyReceive** returns 0. End of file behaves just like any other line of input, however. In particular, you can continue to read more lines after an end of file. The data copied into your buffer is not terminated with a null character (as would be typical for a string in C); to determine the end of the characters returned in the buffer, you must use the length returned by **TtyReceive**.

The constant **TERMINAL_MAX_LINE**, defined in **hardware.h**, defines the maximum line length (maximum buffer size) supported for either input or output on the terminals. (Note that it is perfectly reasonable for user code to send more than this to the terminal; however, you, as the OS, need to break it up so that no more than **TERMINAL_MAX_LINE** bytes gets sent at one time.)

To repeat and summarize:

- In your kernel code, you use **TtyTransmit** to initiate an asynchronous transfer of the specified number (**len**) of bytes, to the specified terminal (**tty_id**), from the specified logical address (**buf**). The call returns immediately; but the hardware does the transfer and traps to you when the transfer is complete.
- In your kernel code, the hardware will trap to you when input data is ready at a terminal. To get that data, you need to have set up a buffer in your logical memory to receive it. This buffer can be characterized by a starting logical address (**buf**) and a bytelength (**len**). Call **TtyReceive** to receive that data: from the specified terminal (**tty_id**) and to be copied into that buffer.

The operation of interrupts on the DCS 58 computer system is explained in more detail in Section 3.4 of this document.

3.3.2 The Hardware Clock

The DCS 58 computer system is equiped with a hardware clock that generates periodic interrupts to the kernel running on the system. When a clock interrupt occurs, the hardware generates a

TRAP_CLOCK interrupt. The periodic frequency of clock interrupts is adjustable at the command line when you invoke **yalnix**.

The operating system must use these **TRAP_CLOCK** interrupts for any timing needs of the operating system. For example, these periodic interrupts might be used as the time source for counting down the quantum in CPU process scheduling.

(Section 5.7.3 will tell you how to control the clock interval from the command line.)

The operation of interrupts on the DCS 58 computer system is explained in more detail in Section 3.4 of this document.

3.4 Interrupts, Exceptions, and Traps

3.4.1 Interrupt Vector Table and Types

Interrupts, exceptions, and traps are all handled in a similar way by the hardware, switching into kernel mode and calling a handler procedure in the kernel by indexing into the *interrupt vector table* based on the type of trap. The following types of interrupts, exceptions, and traps are defined by the DCS 58 hardware:

- **TRAP_KERNEL**: This trap results from a "kernel call" (also called a "system call" or "syscall") trap instruction executed by the current user processes. Such a trap is used by user processes to request some type of service from the operating system kernel, such as creating a new process, allocating memory, or performing I/O. All different kernel call requests enter the kernel through this single type of trap.
- **TRAP_CLOCK**: This interrupt results from the machine's hardware clock, which generates periodic clock interrupts.
- **TRAP_ILLEGAL**: This exception results from the execution of an illegal instruction by the currently executing user process. An illegal instruction can be an undefined machine language opcode, an illegal addressing mode, or a privileged instruction when not in kernel mode.
- **TRAP_MEMORY**: This exception results from a disallowed memory access by the current user process. The access may be disallowed because the address is outside the virtual address range of the hardware (outside Region 0 and Region 1), because the address is not mapped in the current page tables, or because the access violates the page protection specified in the corresponding page table entry.

  Your handler should figure out what to do based on the faulting address.

  In case you're curious, the **code** field in the **UserContext** is derived from the code field from Linux, which ***allegedly*** has the following meanings:

  - **YALNIX_MAPERR**: "address not mapped"
  - **YALNIX_ACCERR**: "invalid permissions"

- **TRAP_MATH**: This exception results from any arithmetic error from an instruction executed by the current user process, such as division by zero or an arithmetic overflow.

- **TRAP_TTY_RECEIVE**: This interrupt is generated by the terminal device controller hardware, when a complete line of input is available from one of the terminals attached to the system.

- **TRAP_TTY_TRANSMIT**: This interrupt is generated by the terminal device controller hardware, when the current buffer of data previously given to the controller on a **TtyTransmit** instruction has been completely sent to the terminal.

The interrupt vector table is stored in memory as an array of pointers to functions, each of which handles the corresponding type of interrupt, exception, or trap. The name of each type listed above is defined in **hardware.h** as a symbolic constant. This number is used by the hardware to index into the interrupt vector table to find the pointer to the handler function for this trap.

The privileged machine register **REG_VECTOR_BASE** is used by the hardware as the base address in memory where the interrupt vector table is stored. After initializing the table in memory, the kernel must write the address of the table to the **REG_VECTOR_BASE** register. In real life, the **REG_VECTOR_BASE** register would contain the *physical address* of the interrupt vector table. However, to simplify the project, here it contains the *virtual address* of the table.

The interrupt vector table *must* have exactly **TRAP_VECTOR_SIZE** entries in it, even though the use for some of them is currently undefined by the hardware. (Note that if a trap happens, the hardware will try to go to the vector address; if you've left that **NULL**, then your kernel will segfault!)

As we have mentioned in class: ***to simplify the programming of your operating system kernel, the kernel is not interruptible.*** That is, while executing inside the kernel, the kernel cannot be interrupted by any interrupt. Thus, the kernel need not use any special synchronization procedures such as locks or cvars ***inside*** the kernel. Any interrupts that occur while already executing inside the kernel are held pending by the hardware and will be triggered only once the kernel returns from the currently interrupt, exception, or trap.

3.4.2 Hardware-Defined User Context Structure

Each of the functions pointed to by entries in the interrupt vector table must have the following function prototype:

void *name*(**UserContext ***)

The **UserContext** structure contains a copy of the hardware state of the currently running user process, as saved by the hardware when the interrupt, exception, or trap occurred.

In order to switch execution from one user process to another, the kernel must be able to save the hardware state of the running process, and to restore this state in the hardware sometime later when context switch this process back in. A copy of this state (including the value of the program counter, stack pointer, general registers) is passed to the kernel when an interrupt, exception, or trap is executed through the interrupt vector table; specifically, the handler function is passed a pointer to a User Context structure (of type **UserContext**). The copy of the hardware state in this User Context structure is automatically created by the hardware as a part of calling the handler function. Upon returning from the handler function, the current contents of this structure is automatically (re)loaded into the hardware state.

The format of the User Context structure is defined in **hardware.h**. The following fields are defined within a **UserContext**:

- **int vector**: The type of interrupt, exception, or trap. This is the index into the interrupt vector table used to call this handler function.
- **int code**: A code value giving more information on the particular interrupt, exception, or trap. The meaning of the **code** value varies depending on the type of interrupt, exception, or trap. The defined values of **code** are summarized in Table 3.2.
- **void *addr**: This field is only meaningful for a **TRAP_MEMORY** exception. It contains the memory address whose reference caused the exception.
- **void *pc**: The program counter value at the time of the interrupt, exception, or trap.
- **void *sp**: The stack pointer value at the time of the interrupt, exception, or trap.
- **void *ebp**: The frame pointer value at the time of the interrupt, exception, or trap.
- **u_long regs[8]**: The contents of eight general purpose CPU registers at the time of the interrupt, exception, or trap. For example, by convention, for a kernel call, these registers are loaded with the arguments for the call before executing the trap instruction, and thus for a **TRAP_KERNEL**, the arguments to the kernel call can be accessed using this **regs** array.

In order to switch contexts from one user process to another, the kernel must save the User Context of the running process into some kernel data structure (e.g., that process' Process Control Block), select another process to run, and restore that process' previously stored User Context by copying that data to the address (i.e., the User Context pointer) passed into the trap handler. The hardware takes care of extracting and restoring the actual hardware state into and from the trap handler's User Context argument.

The current values of the privileged machine registers (the **REG_FOO** registers) are *not* included in the User Context structure. These values are associated with the current process by the kernel, not by the hardware, and must be changed by your kernel on a context switch when/if needed.

Table 3.2: Meaning of Code Values in User Context Structure

Vector	Code
TRAP_KERNEL	Kernel call number
TRAP_CLOCK	Undefined
TRAP_ILLEGAL	Type of illegal instruction
TRAP_MEMORY	Type of disallowed memory access—see Section 3.4.1
TRAP_MATH	Type of math error
TRAP_TTY_TRANSMIT	Terminal number causing interrupt
TRAP_TTY_RECEIVE	Terminal number causing interrupt

3.5 Additional CPU Machine Instructions

In addition to the hardware features of the DCS 58 computer system already described, the CPU provides two additional instructions for special purposes:

- **`void Pause(void)`** Temporarily stops the CPU until the next interrupt occurs. Normally, the CPU continues to execute instructions even when there is no useful work available for it to do. In a real operating system on real hardware, this is all right since there is nothing else for the CPU to do. Operating Systems usually provide an idle process which is executed in this situation, and which is typically an empty infinite loop. However, in order to be nice to other users in the machines you will be using (the other user may be your emacs process), your idle process should not loop in this way. Instead, the idle process in your kernel should be a loop that executes the **`Pause`** instruction in each loop iteration.

- **`void Halt(void)`** Completely stops the CPU (end ends the execution of the simulated DCS 58 computer system). By executing the **`Halt`** hardware instruction, the CPU is completely halted and does not begin execution again until rebooted (i.e., until started again by running your kernel again from the shell on your UNIX terminal).

Chapter 4

Operating System Specification

This chapter describes the Yalnix operating system, a reasonably simple operating system that can be implemented on many types of computer systems including the DCS 58.

This chapter is written partially in the style of a real operating system architecture manual. Everything in this document is something that a real designer, implementor, or user of the operating system would need to know about the operating system. Certain features of the DCS 58 computer system on which your kernel will execute are mentioned here only because you will need them in the project; the Yalnix operating system could also be implemented on other hardware platforms.

4.1 Yalnix Kernel Calls

User processes request services from the kernel by executing a "kernel call" (sometimes called a "system call" or "syscall"). The following kernel calls are defined in Yalnix:

- **`int Fork(void)`**

  `Fork` is the only way new processes are created in Yalnix. The memory image of the new process (the child) is a copy of that of the process calling **`Fork`** (the parent). When the **`Fork`** call completes, *both* the parent process and the child process return (separately) from the kernel call as if they had been the one to call **`Fork`**, since the child is a copy of the parent. The only distinction is the fact that the return value in the calling (parent) process is the process ID of the new (child) process, while the value returned in the child is 0. If, for any reason, the new process cannot be created, this syscall instead returns the value **`ERROR`** to the calling process.

- **`int Exec(char *filename, char **argvec)`**

  Replace the currently running program in the calling process's memory with the program

stored in the file named by **filename**. The argument **argvec** points to a vector of arguments to pass to the new program as its argument list. The new program is called as

```
main(argc, argv)
int argc;
char *argv[];
```

where **argc** is the argument count and **argv** is an array of character pointers to the arguments themselves. The strings pointed to by the entries in **argv** are copied from the strings pointed to by the **argvec** array passed to **Exec**, and **argc** is a count of entries in this array before the first **NULL** entry, which terminates the argument list. When the new program begins running, its **argv[argc]** is **NULL**. By convention the first argument in the argument list passed to a new program (**argvec[0]**) is also the name of the new program to be run, but this is just a convention; the actual file name to run is determined only by the **filename** argument. On success, there is no return from this call in the calling program, and instead, the new program begins executing in this process at its entry point, and its **main(argc, argv)** routine is called as indicated above.

On failure, if the calling process has not been destroyed already, this call returns **ERROR** and does not run the new program. (However, if the kernel has already torn down the caller before encountering the error, then there is no process to return to!)

- **void Exit(int status)**

  Exit is the normal means of terminating a process. The current process is terminated, the integer **status** value is saved for possible later collection by the parent process on a call to **Wait**. All resources used by the calling process are freed, except for the saved **status** information. This call can never return. When a process exits or is aborted, if it has children, they should continue to run normally, but they will no longer have a parent. ***When the orphans later exit, you need not save or report their exit status since there is no longer anybody to care.***

  If the initial process exits, you should halt the system.

- **int Wait(int *status_ptr)**

  Collect the process ID and exit status returned by a child process of the calling program. When a child process **Exits**, its exit status information is added to a FIFO queue of child processes not yet collected by its specific parent. After the **Wait** call, this child process information is removed from the queue. If the calling process has no remaining child processes (exited or running), **ERROR** is returned. Otherwise, if there are no exited child processes waiting for collection by this calling process, the calling process is blocked until its next child calls exits or is aborted. On success, the process ID of the child process is returned and its exit status is copied to the integer referenced by the **status_ptr** argument. On any error, this call instead returns **ERROR**.

- **int GetPid(void)**

Returns the process ID of the calling process.

- **`int Brk(void *addr)`**

  `Brk` sets the operating system's idea of the lowest location not used by the program (called the "break") to **`addr`** (rounded up to the next multiple of **`PAGESIZE`** bytes). This call has the effect of allocating or deallocating enough memory to cover only up to the specified address. Locations not less than **`addr`** and below the stack pointer are not in the address space of the process and will thus cause an exception if accessed. The value 0 is returned on success. If any error is encountered (for example, if not enough memory is available or if the address **`addr`** is invalid), the value **`ERROR`** is returned.

- **`int Delay(int clock_ticks)`**

  The calling process is blocked until **`clock_ticks`** clock interrupts have occurred after the call. Upon completion of the delay, the value 0 is returned. If **`clock_ticks`** is 0, return is immediate. If **`clock_ticks`** is less than 0, time travel is not carried out, and **`ERROR`** is returned instead.

- **`int TtyRead(int tty_id, void *buf, int len)`**

  Read the next line of input from terminal **`tty_id`**, copying it into the buffer referenced by **`buf`**. The maximum length of the line to be returned is given by **`len`**. The line returned in the buffer is *not* null-terminated.

  The calling process is blocked until a line of input is available to be returned. If the length of the next available input line is longer than **`len`** bytes, only the first **`len`** bytes of the line are copied to the calling process, and the remaining bytes of the line are saved by the kernel for the next **`TtyRead`** (by this or another process). If the length of the next available input line is shorter than **`len`** bytes, only as many bytes are copied to the calling process as are available in the input line; On success, the number of bytes actually copied into the calling process's buffer is returned; in case of any error, the value **`ERROR`** is returned.

- **`int TtyWrite(int tty_id, void *buf, int len)`**

  Write the contents of the buffer referenced by **`buf`** to the terminal **`tty_id`**. The length of the buffer in bytes is given by **`len`**. The calling process is blocked until all characters from the buffer have been written on the terminal. On success, the number of bytes written (**`len`**) is returned; in case of any error, the value **`ERROR`** is returned.

  Calls to **`TtyWrite`** for more than **`TERMINAL_MAX_LINE`** bytes should be supported.

  (Recall that Figure 2.3 summarized how the I/O syscalls work with the lower-level hardware.)

- **`int PipeInit(int *pipe_idp)`**

  Create a new pipe; save its identifier at **`*pipe_idp`**. In case of any error, the value **`ERROR`** is returned.

- **`int PipeRead(int pipe_id, void *buf, int len)`**

  Read **`len`** consecutive bytes from the named pipe into the buffer starting at address **`buf`**. Block the caller until these bytes are available. In case of any error, the value **`ERROR`** is returned. Otherwise, return the number of bytes read.

- **`int PipeWrite(int pipe_id, void *buf, int len)`**

  Write the **`len`** bytes starting at **`buf`** to the named pipe. Return when this write is complete. In case of any error, the value **`ERROR`** is returned. Otherwise, return the number of bytes written.

- **`int LockInit(int *lock_idp)`**

  Create a new lock; save its identifier at **`*lock_idp`**. In case of any error, the value **`ERROR`** is returned.

- **`int Acquire(int lock_id)`**

  Acquire the lock identified by **`lock_id`**. In case of any error, the value **`ERROR`** is returned.

- **`int Release(int lock_id)`**

  Release the lock identified by **`lock_id`**. The caller must currently hold this lock. In case of any error, the value **`ERROR`** is returned.

- **`int CvarInit(int *cvar_idp)`**

  Create a new condition variable; save its identifier at **`*cvar_idp`**. In case of any error, the value **`ERROR`** is returned.

- **`int CvarSignal(int cvar_id)`**

  Signal the condition variable identified by **`cvar_id`**. (Use Mesa-style semantics.) In case of any error, the value **`ERROR`** is returned.

- **`int CvarBroadcast(int cvar_id)`**

  Broadcast the condition variable identified by **`cvar_id`**. (Use Mesa-style semantics.) In case of any error, the value **`ERROR`** is returned.

- **`int CvarWait(int cvar_id, int lock_id)`**

  Atomically release the lock identified by **`lock_id`** and wait on the condition variable indentified by **`cvar_id`**. Upon waking up, re-acquire the lock. (Use Mesa-style semantics.) In case of any error, the value **`ERROR`** is returned.

- **`int Reclaim(int id)`**

  Destroy the lock, condition variable, or pipe indentified by **`id`**, and release any associated resources. In case of any error, the value **`ERROR`** is returned.

  If you feel additional specification is necessary to handle unusual scenarios, then create and document it.

Note that the synchronization calls operate on *processes*, unlike the pthreads calls you used in Project 2.

The include file **`yalnix.h`** defines function prototypes for the interface for all Yalnix kernel calls. This file also defines a constant "kernel call number" each type of kernel call, used to indicate to the kernel the specific kernel call being invoked; this type value is visible to the kernel in the **`code`** field of the **`UserContext`** structure passed by the hardware for the trap, as described in Section 4.2.

Robustness Good system software—such as your kernel—must be like a concierge in an expensive hotel. ***Nothing the customers do or say should cause it to stop operating correctly.*** (In your case, the "customer" is the userland code.)

An important part of meeting this goal is *input validation.*

- Your kernel should check that all arguments passed to a kernel call are correct and sensible for the requested service.
- If there is any problem, you should return **`ERROR`**. If things are bad enough that returning to the caller is no longer possible, then you can terminate the caller.

  However, no matter what, if there is a problem you should politely describe it via a **`TracePrintf`**.
- Under no circumstances should you provide additional services than what is documented. (Meet the API contract!)

  (Providing extra, unintended functionality is a common cause of security trouble.)
- If you feel the specification of a service is unclear, then you should clarify it, document this clarification—and then continue as above. (This happens often in the real world!)

In particular, you need to verify that a pointer is valid before you use it. This means your kernel must look in the page table to make sure that the *entire area* (such as a pointer and a specified length) is readable and/or writable (as appropriate) before the kernel actually tries to read or write there. For C-style character strings (null-terminated) you will need to check the address of each byte (C strings like this are passed to **`Exec`**.)

For simplicity, you might write a common routine to check a buffer with a specified pointer and length for read, write and/or read/write access; and a separate routine to verify a string pointer for

read access. The string verify routine would check access to each byte, checking each until it found the `'\0'` at the end. Insert calls to these two routines as needed at the top of each kernel call to verify the pointer arguments before you use them.

Such checking of arguments is important for security and reliability. An unchecked **`TtyRead`**, for instance, might well overwrite crucial parts of the operating system, which might in some clever way gain an intruder access as a privileged user. Also, a pointer to memory that is not correctly mapped or not mapped at all would generate a **`TRAP_MEMORY`**, causing the kernel to crash, leading to a Big Green Screen of Death.

Custom Calls. As the header files show, our code framework also allows for a number of system calls that you can specify yourself (and then implement). This may prove useful if want to add additional features to your system, to support things like shared pages. As noted earlier, we've provided the **`calls.c`** file, as a reference for how the syscall wrappers are implemented.

If you want to add your own new inline assembly syscall wrappers, you may need to add **`-O1`** to your **`CPPFLAGS`**.

4.2 Interrupt, Exception, and Trap Handling

As described in Chapter 3, each type of interrupt, exception, or trap that can be generated by the hardware has a corresponding type value used by the hardware to index into the interrupt vector table in order to find the address of the handler function to call. The operating system kernel must create the interrupt vector table and initialize each corresponding entry in the table with a pointer to the relevant handler function within the kernel. The address of the interrupt vector table must also be written into the **`REG_VECTOR_BASE`** register by the kernel at boot time.

When an interrupt, exception, or trap occurs, the hardware automatically saves the current user mode hardware state on the kernel stack in a **`UserContext`** data structure. When calling the handler function, the hardware passes a pointer to this **`UserContext`** structure as the procedure argument to the handler.

The range of possible index values in the interrupt vector table range from 0 to one less than **`TRAP_VECTOR_SIZE`**, and the interrupt vector table must be exactly **`TRAP_VECTOR_SIZE`**. It's a good idea to initialize all entries, even the non-needed ones, to some type of error function—otherwise, what might happen should that trap actually occur?

The defined kernel call type values, together with an overview of the operation that should be performed by the kernel in response to each, are listed below:

- **`TRAP_KERNEL`**: Execute the requested kernel call, as indicated by the kernel call number in the **`code`** field of the **`UserContext`** passed by reference to this trap handler function.

The return value from the kernel call should be returned to the user process in the **regs[0]** field of the **UserContext**.

- **TRAP_CLOCK**: If there are other runnable processes on the ready queue, perform a context switch to the next runnable process. The Yalnix kernel should implement round-robin process scheduling with a CPU quantum per process of 1 clock tick.

- **TRAP_ILLEGAL**: Abort the currently running Yalnix user process but continue running other processes.

- **TRAP_MEMORY**: The kernel must determine if this exception represents an implicit request by the current process to enlarge the amount of memory allocated to the process's stack, as described in more detail in Section 4.3.4. If so, the kernel enlarges the process's stack to "cover" the address that was being referenced that caused the exception (the **addr** field in the **UserContext**) and then returns from the exception, allowing the process to continue execution with the larger stack. Otherwise, abort the currently running Yalnix user process but continue running other processes.

- **TRAP_MATH**: Abort the currently running Yalnix user process but continue running other processes.

- **TRAP_TTY_RECEIVE**: This interrupt signifies that a new line of input is available from the terminal indicated by the **code** field in the **UserContext** passed by reference to this interrupt handler function. The kernel should read the input from the terminal using a **TtyReceive** hardware operation and if necessary buffer the input line for a subsequent **TtyRead** kernel call by some user process.

- **TRAP_TTY_TRANSMIT**: This interrupt signifies that the previous **TtyTransmit** hardware operation on some terminal has completed. The specific terminal is indicated by the **code** field in the **UserContext** passed by reference to this interrupt handler function. The kernel should complete the blocked process that started this terminal output from a **TtyWrite** kernel call, as necessary; also start the next terminal output on this terminal, if any.

As described above, a **TRAP_KERNEL** occurs when a Yalnix user process requests a kernel call for some function provided by the kernel. A user processes calls the kernel by executing a trap instruction. However, since the C compiler cannot directly generate a trap instruction in the generated machine code for a program, we (like real systems) provide a library of assembly language routines that perform this trap from the user process. This library provides a standard C procedure call interface for the kernel, as indicated in the description of each kernel call in Section 4.1. The trap instruction generates a trap to the hardware, which invokes the kernel using the **TRAP_KERNEL** vector from the interrupt vector table.

Upon entry to your kernel, the **code** field of the **UserContext** structure indicates which kernel call is being invoked (as defined with symbolic constants in **yalnix.h**). The arguments to this call, supplied to the library procedure call interface in C, are available in the **regs** array in the

UserContext structure received by your **TRAP_KERNEL** handler. Each argument passed to the library procedure is available in a separate **regs** entry, in the order passed to the procedure, beginning with **regs[0]**.

Each kernel call returns a single integer value, indicated in Section 4.1, which becomes the return value from the C library procedure call interface for the kernel call. When returning from a **TRAP_KERNEL**, the value to be returned from this interface should be placed by the kernel in **regs[0]** in the **UserContext**.

When the kernel aborts a process, your kernel should **TracePrintf** a message, giving the process id of the process and some explanation of the problem. The exit status reported to the parent process of the aborted process when the parent calls the **Wait** kernel call (as described in Section 4.1) should be the value **ERROR**.

4.3 Memory Management

Your kernel is responsible for all aspects of memory management, both for user processes executing on the system and for the kernel's own use of memory.

Memory management for user processes is reasonably straightforward. To enlarge the the heap of the user process, the process calls the **Brk** kernel call. In general, processes don't do this explicitly themselves, however, since the standard C library **malloc** function will call **Brk** automatically if it needs more memory. To enlarge the stack area of the user process, the process simply tries to use the new stack pointer value, which causes a **TRAP_MEMORY** exception if not enough memory is currently allocated for the process's stack. Details of the procedure for growing the stack of a user process are given in Section 4.3.4.

Memory management for the kernel itself is a bit more complicated. The kernel may also use **malloc** for dynamic memory allocation for its own data structures, but unlike **malloc** from a user program, the kernel can't rely on someone else to actually allocate the memory for it. The kernel must allocate the physical memory into Region 0 for the kernel's own use, as needed by **malloc** calls made by the kernel. Details of this procedure are given in Section 4.3.2. In addition, the kernel must initialize and enable virtual memory at boot time; details of this procedure are given in Section 4.3.1.

4.3.1 Initializing Virtual Memory

As described in Chapter 3, the virtual memory subsystem of the computer is initially disabled at boot time, until it is explicitly enabled by the operating system kernel through the **REG_VM_ENABLE** privileged machine register.

Your kernel must first build the initial page tables, such that for every frame of physical memory

pfn then in use by your kernel, your kernel builds a page table entry so that the new virtual page number **vpn** = **pfn** maps to physical page frame number **pfn**. The hardware loads the kernel beginning at physical address **PMEM_BASE**, but in order to know the extent of pages of physical memory then in use by your kernel, you need some additional information.

To tell you information you need to now, the hardware, during bootstrap, will call a function

```
void SetKernelData(void * _KernelDataStart, void *_KernelDataEnd)
```

which you will write.

- **_KernelDataEnd** is the lowest address not in use by the kernel's instructions and global data, at boot time. As the kernel allocates more memory, the lowest address may change. You will need to keep track of this using a function you will write called **SetKernelBrk**, as described below. When building the initial page table entries for your kernel, the total amount of memory then used by your kernel is indicated by the value saved by **SetKernelBrk**. The initial value, before the first call to **SetKernelBrk**, should be **_KernelDataEnd**.
- **_KernelDataStart** is the lowest address used by your data segment. In building the initial page table entries for your kernel, the page table entries covering up to (but not including) this point should be built with protection **PROT_READ** and **PROT_EXEC** set. For the rest of the initial page table entries built for your kernel (covering the kernel's global data and dynamically allocated data), the page table entries should be built with protection **PROT_READ** and **PROT_WRITE** set.

The **SetKernelData** function gives you the values you need to determine the pages, that (along with the kernel stack) were initially in use by your kernel at boot time, when the kernel first began execution. After that time and before enabling virtual memory, the kernel is running in physical memory mode (all addresses are interpreted by the hardware as physical addresses), and the kernel has access to all of physical memory. If it needs to allocate additional memory dynamically, it can just use the next available physical memory without further ado. However, to help you write the kernel, we provide versions of **malloc** and friends that work correctly even before virtual memory is enabled. This avoids you having to manage physical memory while bootstrapping, and from having to make sure your kernel's dynamic storage remains accessible afterwards. Because your kernel needs to know how much memory your kernel's **malloc** allocates during this time, this version of **malloc** requests memory from and notifies your kernel by using the procedure

```
int SetKernelBrk(void * addr)
```

which you will write. The argument **addr** here is similar to that used by user processes in calls to **Brk** and indicates the lowest location not used (not yet needed by **malloc**) in your kernel.

In your kernel, you should keep a flag to indicate if you have yet enabled virtual memory. Before enabling virtual memory, the **SetKernelBrk** function that you must write need only keep track of the maximum extent (highest address) passed on any call to **SetKernelBrk**. This highest address then can be used to determine the total amount of memory then in use by your kernel at the time you enable virtual memory.

4.3.2 Kernel Memory Management

After you have enabled virtual memory, the job of the **SetKernelBrk** function that you will write, as described in Section 4.3.1, becomes more complicated. Once virtual memory is enabled, your implementation of **SetKernelBrk**, when called with an argument of **addr**, must ensure that all pages containing addresses from **VMEM_BASE** up to (but not including) **addr** are valid, and that no virtual memory pages in Region 0 outside this range (other than those for the kernel stack, explained below) are valid. This will require that **SetKernelBrk** allocate physical memory page frames and map or unmap virtual pages as necessary to make **addr** the new kernel break.

Should it run out of memory or otherwise be unable to satisfy the requirements of **SetKernelBrk**, your **SetKernelBrk** function can return -1. This will eventually cause **malloc** and friends in the kernel to correctly return a **NULL** pointer, as is defined for the standard behavior of **malloc**. If **SetKernelBrk** is successful (the likely case) it should return 0.

Note that the **malloc** implementation that you use inside your kernel (and its helping function **SetKernelBrk**, which you must write) is completely separate from the **malloc** implementation that each user process uses. User processes use the standard C library **malloc** implementation, which requests more memory from the operating system kernel when needed. As we discussed in class, you won't always see an OS syscall with each userland **malloc** call, because the **malloc** library also keeps a list of free blocks of memory and carves up existing free blocks as necessary; as a result, it only asks for more memory from the kernel when its existing available memory cannot satisfy the requested allocation. The kernel uses a slightly different implementation of the standard C library **malloc** implementation, which calls your **SetKernelBrk** function when it needs more total memory.

Each process running on Yalnix has its own break and is linked with its own copy of the **malloc** library routines. Likewise, your kernel has its own break and is linked with its own copy of the **malloc** library routines.

4.3.3 Stack Management

In addition to the stack in Region 1 of each user process, the kernel also needs a stack for each process executing on the system. The kernel is executable code and may need to store local variables and may need to call subroutines and save the return address and such on the stack. By separating the kernel stack for a process from the user stack for a process, the job of the kernel becomes

much less confusing, and also much more flexible; this separation also improves the security of the system, since values that may be left in the memory used for the kernel stack for a process can be made not visible to the user process once executing back in user mode, through use of the hardware memory protection configuration implied by each page table entry.

For example, as described in Section 4.3.4, the user stack of a process grows by trespassing into unmapped memory, which causes a **`TRAP_MEMORY`**, which in turns causes the CPU to enter kernel mode and the kernel to begin executing in its handler for this type of exception. When entering kernel mode, the CPU also switches automatically to the current kernel stack, and thus is no longer modifying the user stack for the process as the kernel runs. At this point, the kernel is executing its own instructions on its own stack and is thus free to manipulate the user stack as it sees fit, without risking interfering with the execution of the user process or without confusing its own execution as the kernel.

By having a separate kernel stack for each process, it is possible for the kernel to block the current process while inside the kernel. When performing a context switch, the kernel can then switch to the kernel stack of the newly selected process (being context switched in), thus leaving the state of the old process's kernel stack entirely unmodified until that process is selected to run again next and is context switched in. All of that process's local variables inside the kernel and all of the subroutine return addresses and such that represent the dynamic state of the process inside the kernel when it was last context switched out, are all still intact and again available when the process is next context switched in. The process can thus begin execution again after the context switch at exactly the point it was at inside the kernel when it last ran.

Unlike the user stack for a process, which can grow dynamically, the kernel stack for each process is a fixed maximum size. This avoids the extreme complexity (and contortions) that would be necessary for the kernel to handle its own **`TRAP_MEMORY`** exceptions and grow its own kernel stack while executing on its own kernel stack. The kernel stack for each process is thus a fixed **`KERNEL_STACK_MAXSIZE`** bytes in size.

The kernel stack is located at exactly the same memory address in virtual memory for all processes. The kernel stack always begins at virtual address **`KERNEL_STACK_BASE`**; the first byte beyond the stack has address **`KERNEL_STACK_LIMIT`**, which is at the extreme top of Region 0 of virtual memory. Each kernel stack is actually stored in different pages of physical memory, but at any given time, only one set of these physical memory pages is mapped into this range of virtual addresses. Your kernel must change this virtual memory mapping on each context switch in order to map in the new process's kernel stack as part of the context switch. Since the kernel stack for each process is located in Region 0 (not in Region 1), to change this virtual memory mapping to map in the new process's kernel stack, you must change the corresponding page table entries in Region 0's page table as part of the context switch operation. (Recall, however, the warning from Section 3.2.5: be sure to change the kernel stack page table entries before flushing the TLB.)

Section 5.1 contains more information on managing the kernel stack.

4.3.4 Growing a User Process's Stack

When a process pushes onto its user mode stack, it simply decrements the current stack pointer value and attempts to write into memory at the new address. Normally, this works with no complication, since this same memory was likely used to store other data earlier pushed onto and popped off of the stack. However, at times, the stack will grow larger than it has been before in this process, which will cause a **`TRAP_MEMORY`** to be generated by the hardware when the process tries to write to the memory pointed to by its new stack pointer value. As noted in Section 4.2, the **`TRAP_MEMORY`** in this case should be interpreted by your kernel as an implicit request to grow the process's user stack.

In your handler for a **`TRAP_MEMORY`** exception, if the virtual address being referenced that caused the exception (the **`addr`** value in the **`UserContext`**) is in Region 1, is below the currently allocated memory for the stack, and is above the current break for the executing process, your Yalnix kernel should attempt to grow the stack to "cover" this address, if possible. In all other cases, you should abort the currently running Yalnix user process, but should continue running any other user processes.

When growing the stack, you should leave at least one page unmapped (with the valid bit in its page table zeroed) between the program and stack areas in Region 1, so the stack will not silently grow into and overlap with the heap without triggering a **`TRAP_MEMORY`**. This unmapped page between the heap and the stack is known as a "red zone" into which the stack should not be allowed to grow. You must also check that you have enough physical memory to grow the stack.

Note that a user-level procedure call with many large local variables may grow the stack by more than one page in one step. The unmapped page that is used to form a red zone for the stack as described in the previous paragraph is therefore not a guaranteed reliable safety measure, but it does add some assurance that the stack will not grow into the program.

4.4 Bootstrapping and Kernel Initialization

As Section 2.1 discussed and Figure 2.1 sketched, your kernel is not a complete program—it is more like a library of functions that get called on interrupts, exceptions, and traps, because what an OS does is react, not cogitate. As a result, your kernel does not have a **`main`** procedure. Instead, your kernel begins executing at a procedure called **`KernelStart`**, which you must write in your kernel. The procedure **`KernelStart`** is automatically called by the bootstrap firmware in the computer, as follows:

```
void KernelStart(char *cmd_args[],
                 unsigned int pmem_size,
                 UserContext *uctxt)
```

The procedure arguments passed to **KernelStart** are built by the bootstrap firmware during initialization of the machine. The **cmd_args** argument is a vector of strings (in the same format as **argv** for normal Unix **main** programs), containing a pointer to each argument from the boot command line (what you typed at your Unix terminal) to start the machine and thus the kernel. The **cmd_args** vector is terminated by a **NULL** pointer. The **pmem_size** argument is the size of the physical memory of the machine you are running on, as determined dynamically by the bootstrap firmware. The size of physical memory is given in units of bytes. Finally, the **uctxt** argument is a pointer to an initial **UserContext** structure. This **UserContext** structure is built by the bootstrap firmware as part of the bootstrap process, rather than being built automatically by the hardware, since this initial **UserContext** is needed before the first interrupt, exception, or trap is executed. Your kernel should use this **UserContext** as the basis for "cloning" others, notably for the **UserContext** that starts the initial process loaded at boot time. Processes created from a **Fork**, instead, copy the **UserContext** from the parent process. (Note that in Yalnix, all processes except for the first process started at boot time are created by **Fork**.)

Before allowing the execution of user processes, the **KernelStart** routine should perform any initialization necessary for your kernel or required by the hardware. In particular, in addition to any initialization you may need to do for your own data structures, your kernel initialization should probably include the following steps.

- Initialize the interrupt vector table entries for each type of interrupt, exception, or trap, by making them point to the correct handler functions in your kernel.
- Initialize the **REG_VECTOR_BASE** privileged machine register to point to your interrupt vector table.
- Build a structure to keep track of what page frames in physical memory are free. For this purpose, you might be able to use a linked list of physical frames, implemented in the frames themselves. Or you can have a separate structure, which is probably easier, though slightly less efficient. This list of free pages should be based on the **pmem_size** argument passed to your **KernelStart**, but should of course not include any memory that is already in use by your kernel.
- Build the initial page tables for Region 0 and Region 1, and initialize the registers **REG_PTBR0**, **REG_PTLR0**, **REG_PTBR1**, and **REG_PTLR1** to define these initial page tables.
- Enable virtual memory.
- Create an "idle" process to be run by your kernel when there are no other runnable (ready) processes in the system. The idle process should be created based on the **UserContext** passed to your **KernelStart** routine. As mentioned in Chapter 3, the idle process should be a loop that executes the **Pause** machine instruction on each loop iteration.
- Create the first process and load the initial program into it. In this step, guide yourself by the file **template.c** that we provide, which shows a skeleton of the procedure necessary to load an executable program from a Unix file into memory as a Yalnix process. This initial

process will serve the role of the "init" process in Unix as the parent (or grandparent, etc.) of all processes running on the system.

(Keep in mind that there already is a kernel process running at boot time. Building *init* might simpler if you think about *becoming* that *init* rather than launching a new process.)

To run your initial program you should put the file name of the init program on your shell command line when you run your kernel. This program name will then be passed to your **`KernelStart`** as one of the **`cmd_args`** strings.

- Use the initial **`UserContext`** passed to **`KernelStart`** as the basis on which to construct the contexts of the init and idle processes, save these contexts somewhere in the kernel (so that these contexts can be switched in and out), and write the context of the init process into the currently active context (the one pointed to by the **`uctxt`** argument to **`KernelStart`**).

- Return from your **`KernelStart`** routine. The machine will begin running the program defined by the current page tables and by the values returned in the **`UserContext`** structure (values which you have presumably modified to point to the initial context of the initial process).

Chapter 5

Additional Hints

To review, in order to complete the project, you must implement a Yalnix kernel that runs on the provided DCS 58 computer simulation. Specifically, you must implement:

- a **`SetKernelData`** routine to catch basic data segment parameters at boot.
- a **`KernelStart`** routine to perform kernel initialization,
- a **`SetKernelBrk`** routine to change the size of kernel memory,
- a procedure to handle each defined type of interrupt, exception, or trap, and
- a procedure (called by your **`TRAP_KERNEL`** handler) to implement each defined Yalnix kernel call.

Note that the kernel call prototypes given in Chapter 4 and in **`yalnix.h`** are the form in which Yalnix user-level programs use these kernel calls. The function names and prototypes of the procedures that implement them inside your kernel need not be the same. The code in your **`TRAP_KERNEL`** handler calls whatever functions inside your kernel it wants to, in order to provide the effect of each type of kernel call.

You will likely end up having a kernel routine for each syscall. We highly recommend that you don't the exact same name for both. For example, you could name the function you use to implement the **`Delay`** inside your kernel "**`KernelDelay`**" It will thus be harder to confuse it with the **`Delay`** library wrapper called by user processes.

How you choose to get arguments and return values between your **`TRAP_KERNEL`** handler and functions like **`KernelDelay`** and its cousins is an internal matter of kernel design. (That is—it's up to you! User code does not see this.)

5.1 Context Switching in the Kernel

5.1.1 Quick Review

Recall, from class, that we have two types of switching going on.

- A trap or such causes a switch from user mode to kernel mode *within the same process.* We start running from the trap handler, in kernel mode, and storing local variables and (if necessary) subroutine context within the kernel stack. But we're still in the same process, with the same address space.
- However, when we need to switch between processes, we do that in kernel mode. Process A, running in kernel model, switches to process B, also in kernel mode.

Switching to a new process takes several steps. Process A first picks some other process B to run next. Process B, kernel mode, is currently in suspended animation. Stored away in a data structure somewhere is the data describing this suspended state:

- the numbers of the physical frames containing process B's kernel stack,
- and the kernel context that should be restored when B starts running again.

To switch to B, process A needs to restore these two items. However, because process A (kernel mode) is itself going to go into suspended animation, it also needs to save its current state: the frame numbers, and the kernel context.

To help with this process of saving your current kernel context and changing to a new one, we provide a magic function to force a context switch from the current process to another process while inside the kernel—and then later, to resume execution of this blocked kernel-mode process.

If the kernel ever switches back to process A, process A will wake up, in kernel mode, thinking it just returned from this call.

5.1.2 The Details

To accomplish this context switching inside the kernel, we provide the function:

```
int KernelContextSwitch(KCSFunc_t *, void *, void *)
```

The type **`KCSFunc_t`** (kernel context switch function type) is a C typedef of a special kind of function:

- It takes a **KernelContext** pointer and two void pointers as arguments.
- It returns a **KernelContext** pointer.

(**KernelContextSwitch** and the type **KCSFunc_t** are defined in **hardware.h**.)

Your kernel code would use **KernelContextSwitch** by passing it a function of this type, and two void pointers. The **KernelContextSwitch** function temporarily stops using the standard kernel context (registers and stack), goes to a special magic stack (independent of all this Yalnix stuff), and calls this function, feeding it:

- a pointer to the kernel context of the caller
- the two void pointers you gave it.

When your function returns, we resume running within the Yalnix kernel, using the kernel context you returned.

For example, to carry out context switches, you might write a function **MyKCS**, that takes three arguments:

```
KernelContext *MyKCS(KernelContext *kc_in,
                     void          *curr_pcb_p,
                     void          *next_pcb_p)
```

When it's time to switch, process A might call:

```
rc = KernelContextSwitch(           MyKCS,
                         (void *)   &current_pcb,
                         (void *)   &next_pcb);
```

MyKCS is then launched, fed with a pointer to a temporary copy of the current kernel context, and these two PCB pointers. Since **MyKCS** is called while not using the normal kernel registers or stack, **MyKCS** can easily and safely do what it needs to do to complete the context switch. As part of this work, it might copy the kernel context into the current process's PCB and return a pointer to a kernel context it had earlier saved in the next process's PCB.

When it calls **KernelContextSwitch**, process A be blocked inside the kernel exactly where it called **KernelContextSwitch**, with exactly the state (register contents and stack contents) that it had at that time. (Note that, except for this state, the rest of the system may have changed!) **KernelContextSwitch** and **MyKCS** do the context switch. If some process B later calls **KernelContextSwitch** to switch back to A, A will resume execution by returning from its earlier call. The real-time sequence of operation may be something like:

- A calls **KernelContextSwitch**
- We execute **MyKCS** with A's arguments, and return.
- B magically starts running
- B calls **KernelContextSwitch**
- We execute **MyKCS** with B's arguments, and return.
- A then resumes running, as if just returned from its **KernelContextSwitch** call.

If things are successful, the return value of **KernelContextSwitch** in is 0. If, instead, any error occurs, **KernelContextSwitch** does not switch contexts and instead returns -1; in this case, you should print an error message and exit, as this generally indicates a programming error in your kernel.

Note that **KernelContextSwitch** doesn't do anything to move a PCB from one queue to another in your kernel, or to do any bookkeeping as to why that process was context switched out or when it can be context switched in again. Nor will it automatically manipulate the page tables. **KernelContextSwitch** only helps you with actually saving the state of the process and later restoring it. Your kernel has to take care of the rest itself.

Also, remember that each process has *its own kernel stack* in the same place in virtual memory (as shown back in Figure 2.2). You don't need to save or restore the actual contents of the kernel stack. However, you do need to *change the page table entries for the kernel stack,* so that these virtual addresses now map instead to the new process's kernel stack in physical memory. The kernel stack for a process begins at virtual address **KERNEL_STACK_BASE** with a constant size of **KERNEL_STACK_MAXSIZE** bytes (this is a multiple of **PAGESIZE**). The limit of the kernel stack is at virtual address **KERNEL_STACK_LIMIT**. These constants are all defined in **hardware.h**. (Recall the flushing warning from Section 3.2.5: change the kernel stack page table entries before flushing the TLB.)

Depending on what you make your **MyKCS** (or any other name you choose) function do, you can also use **KernelContextSwitch** to just get a copy of the current kernel context, in the format of a **KernelContext** structure, without necessarily switching to a new process. You might find this useful in your implementation of **KernelStart** in creating the *idle* or *init* process, and in your implementation of the **Fork** kernel call in creating the new child process.

The actual **KernelContext** data structure definition is provided by **hardware.h**. You don't need to worry what is in a **KernelContext**; just copy the whole thing into the PCB. On a real machine, this structure would be full of low-level hardware-specific stuff; on the DCS 58 simulation, it is full of low-level Linux and x86-specific stuff (as a consequence of our simulation).

5.2 Loading Yalnix User Programs from Disk

When creating the *init* process in **`KernelStart`** and when loading a new program in your implementation of **`Exec`**, you will need to open and read the new program from a Linux file from inside your kernel. Although a bit unrealistic (a real kernel would use that system's own file system), reading the program from a Linux file allows you to use the normal Linux file I/O calls in this project, such as **`open`**, **`read`**, and **`close`** (or **`fopen`**, **`fread`**, and **`fclose`**, if you prefer).

Although this part of loading a new program can be simplified for this project, you still need to be able to understand what you read from the program file and must know how to set up the memory of a new program from the file. This can't be simplified very much and still do the necessary job. So, we provide you a template for the code necessary to do this.

Specifically, we provide a template for a function called **`LoadProgram`**, which can load a program from a Linux executable file into a Yalnix process address space. This function template is in the file **`/yalnix/sample/template.c`**. You should make a copy of that file in your own directory and edit it to fit with your kernel. There are a number of places in that file that you must modify, and there are instructions in the file on how to do these modifications. Look for places marked with the symbol ==>> at the beginning of each line and follow the instructions there.

This file will not compile correctly until you make all the necessary modifications and remove or comment out the ==>> instructions there.

5.3 Addresses

You will need to think carefully about *addresses*. Much of what you'll be doing involves juggling details of address spaces, and leaving data structures around in places where you can find them later. (This is one reason why we've been using C so far in the course.)

Some common address errors (that will cause the simulation to print out a long message and halt) include:

- having your kernel access a page in a manner inconsistent with its protections (e.g., trying to write to a read-only page)
- having your kernel access an illegal address
- making a memory reference when the page-frame mapping for the page table for that reference is no longer in the TLB
- leaving uninitialized data in the page table

5.4 Process IDs, and other Identifiers

Each process in the Yalnix system must have a unique integer process ID. Since a signed integer allows over 2 billion processes to be created before overflowing its range, you may (for this project) simply assign sequential positive integers to each new process created and not worry about integer overflow (real operating system kernels must worry about this, though). The process ID *must* be implemented as an integer, *not* as a pointer.

You may end up needing to implement a mechanism inside your kernel to find a process's PCB (for example) given only its process ID. (For example, if we had a suite of message-passing syscalls, this might be necessary.) Within a Yalnix implementation, you may find other ways to get the PCB you need. Once you've mastered the basic functionality of the project, you might want to start considering efficiency and slickness. The challenge of how to map process ID to PCB is one place to start.

A similar discussion applies to the identifiers for the synchronization objects.

5.5 Fork

On a **`Fork`**, a new process ID is generated and a new PCB is allocated. The kernel stack (which contains the saved user context from when the user executed the **`TRAP_KERNEL`** to call **`Fork`**) and kernel context for the child process are created as copies of the parent (calling) process. New physical memory is allocated for the child process, into which you copy the parent's memory space contents.

Since the CPU can only access memory by virtual addresses (using the page tables), you must map both the source and the destination of this copy into the virtual address space at the same time. That is: the virtual address of the source page in the parent is the same as the virtual address of the destination page in the child. But somehow, you need to copy the parent's frame to the child's frame. This means that you need to temporarily arrange for both frames to be mapped into your address space at the same time.

You need not map all of both address spaces at the same time, however: the copy may be done piecemeal, since the address spaces are already naturally divided into pages.

5.6 Exec

On an **`Exec`**, you must load the new program from the specified (Linux) file into the program region of the calling process, which will in general also require changing the process' page tables. The program counter in the **`pc`** field of the user context must also be initialized to the new pro-

gram's entry point. The template **LoadProgram** function described in Section 5.2 shows how to do this and takes care of most of the messy details for you.

The template **LoadProgram** also shows how to initialize the stack for the new program. The stack for a new program starts with only one page of physical memory allocated, which should be mapped to virtual address **VMEM_1_LIMIT** - **PAGESIZE**. As the process executes, it may require more stack space, which can then be allocated as usual for the normal case of a running program.

To complete initialization of the stack for a new program, the argument list from the **Exec** must first be copied onto the stack. The template **LoadProgram**function also takes care of most of this for you.

To accommodate constraints of various underlying hardware architecture, the template code does two additional tasks:

- The SPARC hardware architecture (on which some Yalnix versions run) also requires that an additional number of bytes (defined in **hardware.h** as **INITIAL_STACK_FRAME_SIZE**) be reserved immediately below the argument list.

- Some Linux/x86 systems get confused if there are insufficient null bytes after the argument vector. We allocate **POST_ARGV_NULL_SPACE** additional 4-byte words, and zero them.

The stack pointer in the **sp** field of the user context structure should then be initialized to the lowest address of this space reserved for the initial stack frame. Like all addresses that you use, this must be a virtual address.

Again, all these details are presented in the form of a C procedure **LoadProgram** in the file **template.c**.

5.7 Compiling and Running Yalnix

5.7.1 Compiling Your Kernel

The file **/yalnix/sample/Makefile** contains a basis for the **Makefile** we recommend you use for this project. The comments within it should be self-explanatory. There is some magic involved how this **Makefile** sets up compiling and linking with **gcc**, but you may safely ignore most of the details.

The arguments passed to the compiler and linker to make the kernel and Yalnix user programs should not be changed.

5.7.2 Library Calls

Your kernel and user code may use standard C library functions listed in

/yalnix/etc/yuserlib/common/.

Your kernel and user code may also use the **malloc** family of library functions—but we've modified them so their syscalls go into the right parts of the Yalnix software, instead of to the Linux host.

However (except as noted elsewhere) you should not use other library functions that make system calls. It breaks the simulation.

As noted earlier, I replaced the uncooperative glibc heap code with new heap code I wrote on the airplane to Stanford. This code includes a new library function:

```
void check_heap( void )
```

This function does some sanity checking on the heap, and reports warnings and usage statistics as **TracePrintf**s, level 2. When called from your Yalnix code, it checks the kernel heap; when called from a user process, it checks the user heap of that process.

5.7.3 Running Your Kernel

Your kernel will be compiled and linked as an ordinary Linux executable program, and can thus be run as a command from the Linux shell prompt. When you run your kernel, you can put a number of Linux-style switches on the command line to control some aspects of execution. The file name for your executable Yalnix kernel should be **yalnix**. You can then run your kernel as:

yalnix -t *tracefile* **-lk** *level* **-lh** *level* **-lu** *level* **-s** *initfile initargs...*

All of the switches and arguments shown here on the command line are optional. The meaning of each of the command-line switches is as follows:

- **-t** *tracefile*: This turns on "tracing" within the kernel and machine support code, and optionally specifies the name of the file to which the tracing output should be written. If no *tracefile* argument follows the **-t** switch, the default *tracefile* name is "**TRACE**" in the current directory.

  To generate trace output from your kernel, you may call

```
TracePrintf(int level, char *fmt, args...)
```

where **level** is an integer tracing level, **fmt** is a format specification in the style of **printf**, and **args...** are the arguments needed by **fmt**. You can run your kernel with the "tracing" level set to any integer. If the current tracing level is greater than or equal to the **level** argument to **TracePrintf**, the output generated by **fmt** and **args...** will be added to the trace. Otherwise, no trace output is generated from this call to **TracePrintf**.

We cannot understate the importance of TracePrintf. If used properly and consistently while developing the project, it can help solve a lot of problems. For example, you might put a low-level **TracePrintf** at the beginning and end of each function, and put a higher-level **TracePrintf** when important actions occur.

- **-lk** *level*: Set the tracing level for this run of the kernel. The default tracing level is -1—which is how we will test your code—if you enable tracing with the **-t** or **-s** switches. You can specify any level of tracing.

- **-lh** *level*: Like **-lk**, but sets the trace level to be applied to internal **TracePrintf** calls made by the hardware simulation. The higher the number, the more verbose, complete, and incomprehensible the hardware trace. Tracing the hardware simulation may sometimes be useful to you in your own debugging; if you encounter any really strange problems, this tracing may be very useful to us in helping you find your problem or in diagnosing any internal simulator problems (although no such problems are expected).

- **-lu** *level*: Like **-lk** and **-lh**, but sets the tracing level applied to **TracePrintf** calls made by user-level processes running on top of Yalnix.

  Again, we will test your code with user level tracing set at 1.

- **-s** : Send the tracing output to the Linux **stderr** file (this is usually your screen) in addition to sending it to the tracefile. This switch enables tracing, even if the **-t** switch is not specified.

In your initial testing, you might also want to include the **-n** flag, which suppresses use of the X window system for the simulated terminals.

These switches are automatically parsed, interpreted, and deleted for you before **KernelStart** is called. The remaining arguments from the command line are passed to **KernelStart** in its **cmd_args** argument. For example, if you run your kernel with the command line

```
yalnix -t -lk 5 -lh 3 init a b c
```

then your **KernelStart** would be called with **cmd_args** as follows:

```
cmd_args[0] = "init"
cmd_args[1] = "a"
cmd_args[2] = "b"
cmd_args[3] = "c"
cmd_args[4] = NULL
```

Inside **KernelStart**, you should use **cmd_args[0]** as the file name of the *init* program, and you should pass the whole **cmd_args** array as the arguments for the new process. You should use a default *init* program name of "**init**" (with no additional arguments) if no **cmd_args** are specified when Yalnix is run from the Linux shell.

You can also use a command-line option to change the clock speed. The default tick interval is allegedly 400 ms; you can now change to some other number by **-C NNN**.

5.8 Testing and Debugging Your Kernel

5.8.1 Testing

Keep in mind that, when grading your project, we will be running your kernel on all sorts of test cases—some looking at basic aspects of functionality, and some to see if your code can gracefully handle scenarios that are intentionally stressful. (To give you an idea of the latter, one is based on some student-written code called **torture**, which we provide in our sample test code. Most kernels usually crack within about 20 seconds; the rest endure indefinitely. Aspire to be in the latter category.)

As a consequence, it would be a good idea for you to thoroughly test your kernel first.

To assist in testing and debugging, I have set up some additional options for driving the terminals. If you have a file **infoo1** full of data that you'd like to feed to **YTERM1**, include **-I1 infoo1** on the command line when you invoke Yalnix. If you would like to specify some file other than **TTYLOG.1**, use **-O1 outbar**. You can specify redirection for the other terminals as well.

5.8.2 Debugging

You may use **TracePrintf** within your kernel to generate any necessary debugging output you want to see. Please be aware that the insertion of such print statements may alter the behavior of your program, since they can change the timing of when things happen relative to when clock interrupts or terminal interrupts happen, for example.

You should use **gdb**.

Because of our emulator's heavy use of Linux signals internally, it is necessary to tell the debugger to ignore certain events. One way is to type the following to the **gdb** command line each time you run it:

```
handle SIGILL SIGFPE SIGBUS SIGSEGV pass nostop noprint
handle SIGALRM SIGUSR1 SIGCHLD SIGPOLL pass nostop noprint
```

Other options:

- Use the **-x** option to **gdb** to specify a file containing these commands.
- Put these commands in your **.gdbinit** file in your home directory (which will cause them to be invoked *every* time you run **gdb**.
- Put them in a **.gdbinit** file in your Yalnix working directory.

(A sample lives at **/yalnix/sample/dot.gdbinit**.)

Our VirtualBox images will have **.gdbinit** installed in the home directory of the default account.

It is rumored that **dbx** also works, but I have not tested it.

When debugging "Big Green Screens," it can be very useful to run gdb on the dumped core—that will often tell you exactly which data structure in your kernel went bad.

This means that your Linux shell and account need to be set up so that you can actually dump core.

- If you use csh or tcsh, put this in your **.cshrc**:

  ```
  limit core unlimited
  ```

- If you use bash, put this in your **.bashrc**:

  ```
  ulimit -c unlimited
  ```

- If the above doesn't work, check with the sysadmins.

(Remember, use **make no-core** to clean up old core files.)

Here are a couple of debugging tactics that have proven useful:

- **Looking for heap corruption.** Down in the kernel, you build a lot of tables, and (since you're using C and living close to the metal) there's nothing to stop you from writing to any valid writable address you want to. As a result, it can be easy to overrun the ends of malloc'd

tables, and corrupt the heap. This kind of bug can be hard to find, since the consequence can appear to be far disconnected from the cause.

However, the heap code we're using this term has lots of sanity checking, and will warn you if it sees some corruption happening. Run yalnix with hardware tracing at least 2 to see these warnings.

- **Flushing the TLB.** If a page is currently valid, but you change its page table entry, the *old* entry will persist in the TLB until you flush.

  Not flushing correctly can also be the source of maddening bugs. For one example, **`LoadProgram`** might establish the new program load into the old region1's frames; for another example, you might end up copying both frames of the kernel stack into the same frame.

5.9 Controlling Your Yalnix Terminals

By default (unless you specify the option **`-n`** option on the command line or unless you are not running X windows) each Yalnix terminal is emulated as an **`xterm`** on your X windows display.

The current DCS 58 machine configuration supports four terminals, numbered from 0 to 3, with terminal 0 serving as the Yalnix system console, and the other 3 serving as regular terminals. In fact, though, these uses are only a convention, and all four terminals actually behave in exactly the same way. The constant **`NUM_TERMINALS`** in **`hardware.h`** defines the total number of terminals supported. The constant **`TERMINAL_MAX_LINE`** defines the maximum length of line supported for either input or output on the terminals.

When you run your kernel, the X window system will create four new windows on your display, called

```
Yalnix Console
Yalnix Terminal #1
Yalnix Terminal #2
Yalnix Terminal #3
```

to act as the four terminals that the hardware configuration supports.

Each of these windows is actually running Linux **`xterm`** and can be controlled as such. For example, you can resize or inconify any of these windows, and can use the scroll bar to review output in the window.

For those who like to play with configuring X windows, you can define the shell environment variable **`YALNIX_XTERM`** to be **`xterm`** command line arguments to be used when starting the Yalnix terminal windows. You can also define the shell environment variables **`YALNIX_XTERM0`** ,

YALNIX_XTERM1 , **YALNIX_XTERM2** , and **YALNIX_XTERM3** to be further command line arguments to be used only when starting the corresponding terminal window. For example, you could define these four environment variables to contain appropriate "**-geometry**" options to automatically place each window on the screen for you. You can also include **xterm** options to change the font size in each window, change the foreground or background colors, etc.

When your kernel program ends, these four X windows automatically go away. However, to allow you to look at them before they disappear, the **Halt** instruction pauses before exiting from your kernel program, and prints

```
Press RETURN to end simulation...
```

Once you hit the **RETURN** key, the machine simulation will exit and the console and terminal windows will disappear.

The terminal windows keep log files of all input and output operations on each terminal. The files **TTYLOG.0** , **TTYLOG.1** , **TTYLOG.2** , and **TTYLOG.3** record all input and output from each terminal separately, and the file **TTYLOG** collectively records all input and output from the four terminals together. The terminal logs show the terminal number of the terminal, either "<" for input or ">" for output, and the line input or output. You can type an end of file on a terminal with control-D as in Unix, which appears in the terminal log files as "`(EOF)`".

5.10 Cleaning Up

Sometimes, termination of your kernel may leave lots of **yalnixtty** processes running on your (Linux) machine. You can clean those up by typing

```
killall yalnixtty
```

You may also end up with a **yalnixnet** or **yalnix** process still running, if yalnix ended non-gracefully.

Some students also find it useful to write a "startup script" that does things like:

- **make kill**
- **make clean**
- **make**
- run their code with the correct parameters.

5.11 Overall Plan of Attack

You may choose to implement the various parts of your kernel in any order you see fit. If you are having difficulties putting together a coherent plan, we suggest you start with the following sketch of a plan. Please remember to thoroughly test you code using your own stubs or test procedures, and as part of the whole system, at each and every opportunity. The plan of attack below is not necessarily complete, but it is suggestive of a full plan that can allow you to complete the project in some order. (See also Section 1.2.5.)

- Read the handout for the project carefully.

  Try to understand it and perhaps read it again. Ask the instructor or a TA if you don't understand parts. Although there are a lot of pages here to read, the handout is intended to be very complete and to guide you through the project.

- Think through and sketch out some high-level pseudo-code for each type of kernel call, interrupt, and exception. Then decide on the things you need to put in what data structures (notably in the Process Control Block) to make it all work. Iterate until the pseudo-code and the main prototype data structures mesh well together.

- Take a cursory look at the template **`LoadProgram`** function in the file **`template.c`**. You need not understand all the details, but make sure you understand the comments that are preceded by " ==>>" markers.

- Write an initial version of your **`KernelStart`** function to bootstrap the kernel. You need to initialize the interrupt vector table here and point the **`REG_VECTOR_BASE`** register at it. You also need to build the initial page tables and enable virtual memory by writing a 1 into the **`REG_VM_ENABLE`** register. The support code that makes the DCS 58 simulation work does a lot of error checking at this point, so if you get through this far with no errors, you are probably doing OK (although we can't check for all possible errors at this point).

- Write an *idle* routine in your kernel text (a single infinite loop that just executes a **`Pause`** instruction). Try running your kernel with just an *idle* process to see if that much works.

- Write a simple *init* program. The simplest *init* program would just loop forever. Make sure you can get your **`Makefile`** to compile this as a Yalnix user program, since you will need this skill in order to write other test programs later. Modify your **`KernelStart`** to start this *init* program (or one passed in the Unix shell command line) in addition to the *idle* program.

- Make sure your *init* and *idle* processes are context switching. At this point you will be able to pass the second assignment checkpoint.

- Implement the **`GetPid`** kernel call and call it from the *init* process. At this point your kernel call interface is working correctly.

- Implement **`SetKernelBrk`** to allow your kernel to allocate substantial chunks of memory. It is likely that you haven't needed it up to this point, but you may have (in this case implement it earlier).
- Implement the **`Brk`** kernel call and call it from the *init* process. At this point you have a substantial part of the memory management code working.
- Implement the **`Delay`** kernel call and call it from the *init* process. Make sure your *idle* process then runs for several clock ticks uninterrupted, until the delay period expires. This will be the first proof that blocking of processes works.
- Implement the **`Fork`** kernel call. If you get this to work you are almost done with the memory system.
- Implement the **`Exec`** kernel call. You have already done something similar by initially loading *init*.
- Write another small program that does not do much. Call **`Fork`** and **`Exec`** from your *init* process, to get this third program running as a child of *init*. Watch for context switches.
- Implement and test the **`Exit`** and **`Wait`** kernel calls.
- Implement the kernel calls related to terminal I/O handling. These should be easy at this point, if you pay attention to the process address space into which your input needs to go.
- Implement the synchronization syscalls.
- If time permits, consider trying to port your Project 2 solution onto Yalnix.
- ***Look at your work and wonder in amazement at the road you have traveled, and the enlightenment you have achieved.***