CS58/258, Dartmouth College

The Yalnix Project
Overhauled and Revised

Sean Smith, Dartmouth College

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

Draft of September 28, 2022

Contents

List of Figures

Preface for Instructors

During their periods of quarantine, Newton invented calculus and it is claimed that Shakespeare wrote *King Lear.* In my pandemic isolation, I overhauled the Yalnix OS project we use in Dartmouth's COSC58/258. (I also overhauled the CPU tools used in COCS51, but that's another story.)

If you've been through our Yalnix before this Spring 2020 overhaul, here are the principal differences:

- The support code has been updated to build and run on a modern Linux installation. Along the way, many bugs were discovered and fixed.
- The transitions between the "hardware support" and the student code have been clarified (see Figure 1.1).
- The hardware traceprints have been totally revamped in order to make it clear to the student what's happening with respect to these transitions (Section 6.1).
- My previous **`check_heap`** routines have been greatly expanded into a suite of "helper" tools designed to flag many common kernel bugs (Chapter 6).
- There's now a series of explicit examples (Chapter 7) of how to use these tools and gdb to debug yalnix kernels—including looking into userland. (The VirtualBox image has been also configured to allow code—even running in shared folders—to dump core in a known place.)
- The **`yalnix`** command line options have been overhauled to make it easier for students to do the right thing (Section 5.4).
- The Makefile and include structure have been simplified (Section 5.2 and Section 5.5).
- There's now an expanded checkpoint discussion (Chapter 8) with step-by-step advice. (The earlier manual's roundabout, repetitive presentation on how to build the project has been eliminated.)
- There's now also an expanded discussion of how to use **`KernelContextSwitch`** (Chapter 4).

To run an earlier kernel on the new framework, there are few things to note.

- See Section 8.2.2 for how to find out initial kernel addreses when first building the Region 0 page table. **`SetKernelData`** and the initial call to **`SetKernelBrk`** have been eliminated.)
- Use **`helper_new_pid`** to get a new pid, and **`helper_retire_pid`** to retire one. See Section 6.1.1.
- Command-line options have been simplified. To run **`my_init`** with default tracing and no Xterms, type **`./yalnix my_init`**. To run it with Xterms, type **`./yalnix -x my_init`**.

Since the Fall 2020 incarnation, there's been one significant change: the build process now requires an explicit **`YALNIX_FRAMEWORK`** environment variable saying where the **`yalnix_framework`** directory lives. (This means that if your project predates this, then you need to change the **`DDIR58`** line in your Yalnix **`Makefile`** to point to **`$(YALNIX_FRAMEWORK)`**.)

Also note that that old Virtual Box images don't always work on newer versions of Virtual Box.

Chapter 1

The Big Picture

Through this assignment, you will be able to learn how a real operating system kernel works—how it manages the hardware resources of the computer system and provides 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 a Linux OS on an Intel machine will be made to behave like a 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 virtual Linux machine. 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.

Machines The class Canvas page has information on how to find a machine to use for this project.

- Thayer maintains both Linux/Intel servers as well as rooms of Intel machines that boot into Linux.
- If your personal machine has an Intel architecture (that is, not an M1 Mac), then you can use the Virtual Box image we provide. (To minimize surprises, this image has the same Ubuntu, gcc, glibc, and gdb as the Thayer machines.)

The project build environment requires access to the **`yalnix_framework`** directory containing our libraries, headers, etc. The build also requires a **`YALNIX_FRAMEWORK`** environment variable pointing to this directory. On our Virtual Box image, **`yalnix_framework`** lives in the **`cs58`** home directory, and the environment variable is already set up. On the Thayer machines, **`yalnix_framework`** lives at **`/thayerfs/courses/22fall/cosc058/workspace/yalnix_framework`**. You should modify your **`.bashrc`** or such to set up the **`YALNIX_FRAMEWORK`** environment variable to point to that.

1.1 Overall Flow

Let's start with the punchline. Figure 1.1 shows the overall flow of system execution. You will write the kernel code (light gray boxes). You may also write application code (dark gray), although we provide some samples. Figure 1.1 also provides a framework for talking through the principal transitions.

- **Boot.** In **Transition 1**, the hardware invokes **`KernelStart()`** to boot the system. When this function returns (**Transition 2**), the machine begins running in user mode at a specified UserContext.

- **Kernel Heap.** Your kernel code will be linked to a kernel library (which we provide). This library includes helpful routines such as **`malloc()`** and friends, to manage dynamic allocation from the kernel heap. When the kernel library needs to adjust the top of the kernel heap, it will call **`SetKernelBrk()`** (**Transition 3**), which you will write.

- **Machine Instructions.** Your kernel code will need to call special machine instructions (**Transition 4**) to do things like change the page table pointers.

- **Syscalls, Interrupts, Exceptions.** When these happen, the hardware will throw a trap, switch to kernel mode, and invoke the trap handler you told it to invoke (**Transition 5**). (Note that while execution transitions from userland to kernelland, as the diagram shows, the *underlying cause* of that transition is the hardware, and may even be unrelated to what userland is currently doing.) When your handler returns, the hardware returns to user mode (**Transition 6**).

- **TLB Misses.** If userland or kernelland execution touch an address whose page translation is not in the TLB, the hardware will handle that miss (**Transition 7**) according to what it believes are the current page tables.

- **Kernel Context Switching.** Having the kernel change itself to a different execution context is tricky—one would need to change all the registers and PC and stack pointer while using those same registers. To make this easier for you, we provide a way for your kernel code to invoke kernel context-switching functions, which you will write, on a special context (**Transition 8,9**).

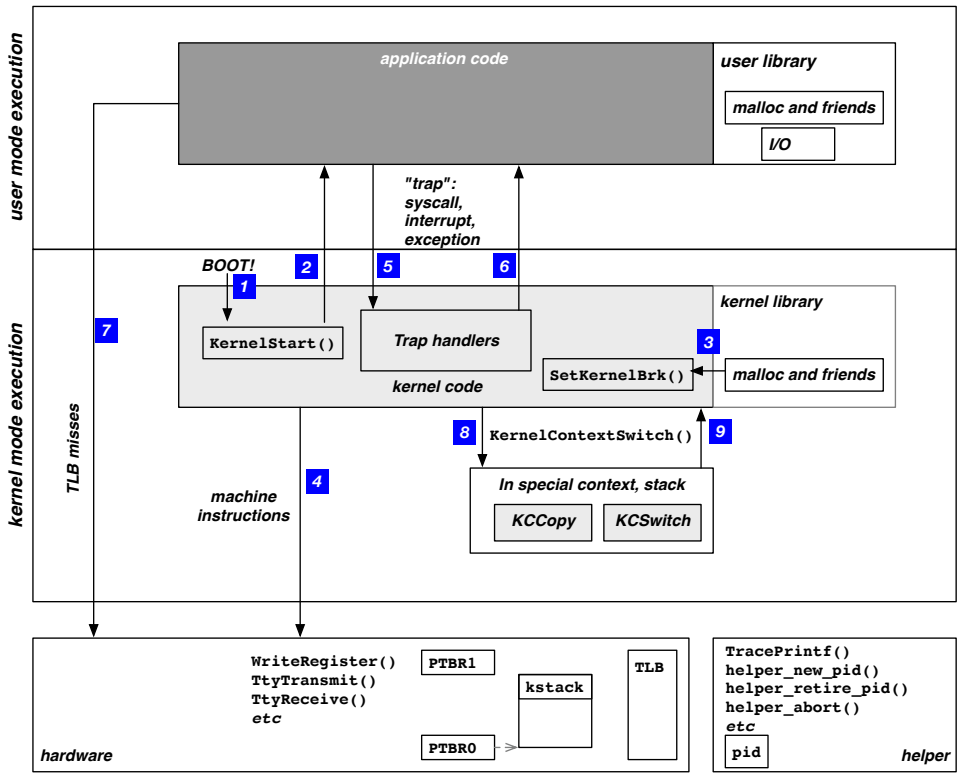

Figure 1.1: The overall flow of Yalnix.

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

Dartmouth's Yalnix 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.

As the preface notes, the 2020 version of Yalnix has been thoroughly revised.

1.3 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 other members of the course staff.

1.4 What's Coming Up

This manual starts by talking about the OS.

- Chapter 2 specifies the hardware of our fictional computer.
- Chapter 3 specifies the Yalnix OS, executing on top of this hardware.

We then give some advice on building the project.

- Chapter 4 goes into some detail about switching kernel contexts, a topic implementers can find tricky and for which our tools provide support.
- Chapter 5 gives advice on coding the project.
- Chapter 6 and Chapter 7 give advice on finding and eliminating bugs.

With the above background in place, we then spell out what you will do.

- Chapter 8 presents the sequence of project checkpoints and how to approach them.
- Chapter 9 gives some ideas for extra functionality.
- Chapter 10 discusses project grading.

Have fun!

Chapter 2

The Hardware

This chapter describes the hardware architecture of the fictional DCS 58 computer architecture on which you will implement an operating system kernel in this project. This 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.

2.1 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.

2.1.1 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.
- **`EBP`**, the *frame pointer* (aka *base pointer* contains the virtual address of the current process's stack frame. That is: this points to the highest end (that is, the bottom, since stacks grow toward zero).
- **`R0`** through **`R7`**, the eight *general registers*, used as accumulators or otherwise to hold temporary values during a computation.

2.1.2 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 2.1 summarizes these privileged registers.

Details for the use of each of these registers are described in the following sections, as indicated in Table 2.1, 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:

Table 2.1: DCS 58 Privileged Register Summary.

Name	Purpose	Readable	Details
REG_PTBR0	Page Table Base Register 0	Yes	Section 2.2.4
REG_PTLR0	Page Table Limit Register 0	Yes	Section 2.2.4
REG_PTBR1	Page Table Base Register 1	Yes	Section 2.2.4
REG_PTLR1	Page Table Limit Register 1	Yes	Section 2.2.4
REG_TLB_FLUSH	TLB Flush Register	No	Section 2.2.5
REG_VM_ENABLE	Virtual Memory Enable Register	Yes	Section 2.2.6
REG_VECTOR_BASE	Vector Base Register	Yes	Section 2.4

- **`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 2.1, 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`**.

2.2 Memory Subsystem

2.2.1 Overview

The memory subsystem of the DCS 58 computer supports a 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.

`hardware.h` defines a number of 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`**.

2.2.2 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 2.2.1, 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***.

2.2.3 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 2.1.

- 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 2.1.)

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. (See Figure 4.1 and Chapter 4).

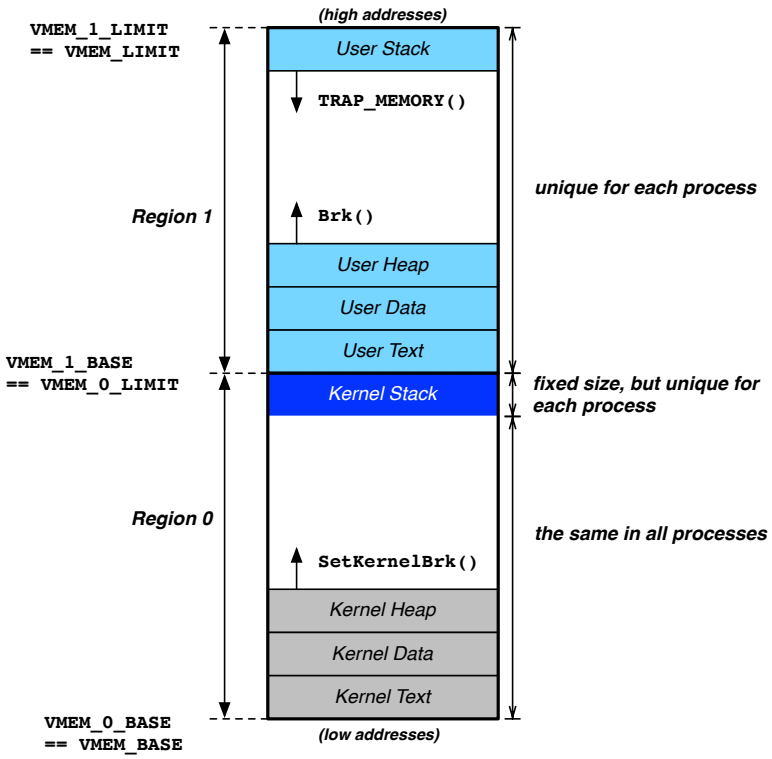

Figure 2.1: The virtual address space layout in Yalnix. Each process has its own Region 1. Each process also has its own kernel stack, constant size. However, there is only one kernel, shared by all processes.

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.

2.2.4 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 direct, single-level page tables to define the mapping from virtual to physical addresses. Each 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:

- **`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.

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 zeroth 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 zeroth 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`**.

> ***Warning:*** note above that page table are indexed by relative position of that page in that region. E.g., suppose each region has **`128`** pages and Region 0 starts with page **`0`**. If you want to write to the pte for page **`200`**, you would use the Region 1 page table (since it's a Region 1 page) and use **`200-128 = 72`** as the index. If instead you used **`200`** as the index, you would be writing off into mysterious space beyond the page table; such bugs can be hard to track down.

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:

[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.

- **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 2.2 and is defined in the file **hardware.h** as a C data structure as a **pte_t**. 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.

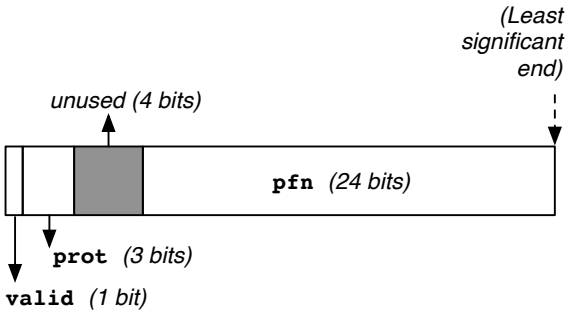

Figure 2.2: DCS 58 Hardware Page Table Entry (PTE) Format. Note that not all bits in the word are used. Unused bits may be used by your OS for its own nefarious purposes.

2.2.5 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 (in the near future) 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_1**: Flush all entries in the TLB that correspond to virtual addresses in Region 1.
- **TLB_FLUSH_KSTACK**: Flush all entries in the TLB that correspond to virtual addresses of the kernel stack pages in Region 0.
- **TLB_FLUSH_0**: Flush all entries in the TLB that correspond to virtual addresses in Region 0. However, note the two following details:
  - ***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!
  - 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.*** If you change-flush one page at a time, the kernel stack may get into an inconsistent state.
- 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**.

2.2.6 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. On the DCS58, ***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

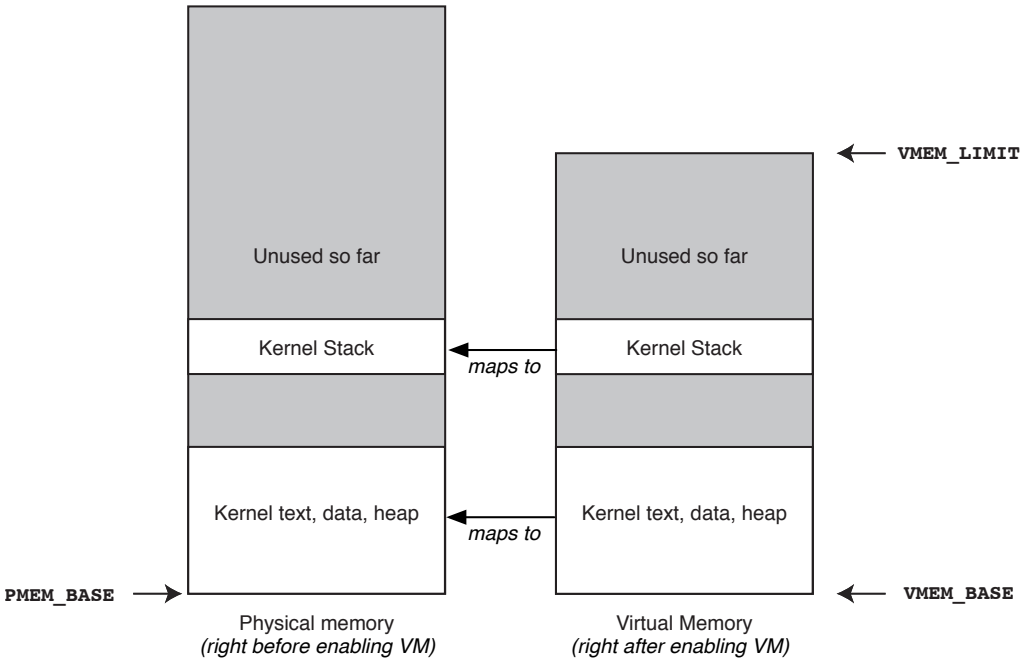

Figure 2.3: Initial memory layout before and after enabling virtual memory. If frame n is used used before VM is enabled, then page n needs to be valid and map to frame n—otherwise, surprising things may happen.

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 2.3. The hardware makes 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 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`**.

2.3 Hardware Devices

The DCS 58 computer system you will be using has been configured with three types of hardware devices: a hardware clock, a number of terminals, and a disk. (For the main project, you will only use the first two.)

2.3.1 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** (see Section 5.4.3).

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.

The operation of interrupts on the DCS 58 computer system is explained in more detail in Section 2.4 of this document.

2.3.2 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 our system, your kernel will interact with the terminals via

- a pair of machine instructions (Transition 4 in Figure 1.1)
  - **void TtyTransmit(int tty_id, void *buf, int len)**
  - **int TtyReceive(int tty_id, void *buf, int len)**
- and a pair of traps (Transition 5 in Figure 1.1)
  - **TRAP_TTY_TRANSMIT**
  - **TRAP_TTY_RECEIVE**

Output to the user. As Figure 2.4 shows, **void TtyTransmit(int tty_id, void *buf, int len)** 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, the memory at these addresses could become unavailable or change after a context switch—leading to user surprise and potentially kernel crashes.

When the data has been completely written out, the terminal controller will generate an interrupt: **TRAP_TTY_TRANSMIT**. 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.

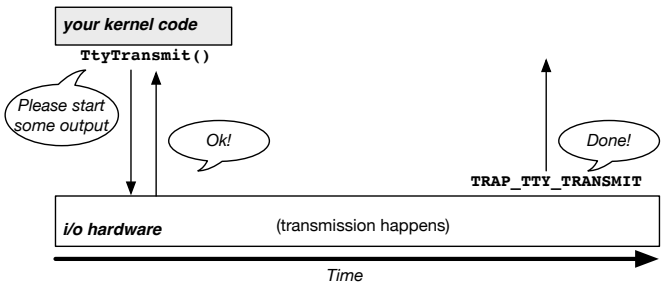

Figure 2.4: ***TtyTransmit*** *gets the hardware to start transmitting bytes to a terminal. When the hardware later completes the transmission, it throws a trap.*

Input from the user. As Figure 2.5 shows, 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 **int TtyReceive(int tty_id, void *buf, int len)** 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**.

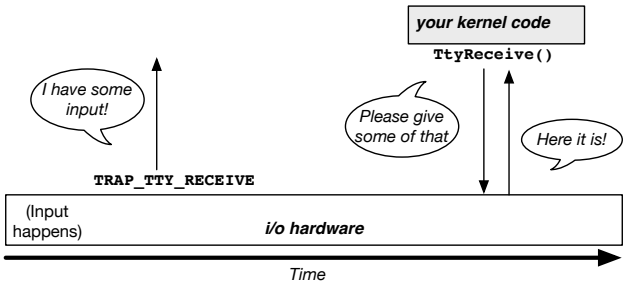

*Figure 2.5: When the terminal receives some bytes from the user, the hardware throws a trap. The kernel can then use **TtyReceive** to grap some of them.*

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 within one of these instructions. (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 after that, 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.

2.3.3 Disk

In the standard Yalnix project, you will not use this.

The machine has a hard disk of **NUMSECTORS**, each of **SECTORSIZE.** (See the end of **hardware.h**.)

To use it, your kernel code should issue a **DiskAccess** instruction:

- **void DiskAccess(int opcode, int sectornum, void * addr)** The opcode should be **DISK_READ**

or **DISK_WRITE**. It would be a good idea if the buffer at **addr** was real, mapped memory, in the kernel space, with appropriate permissions.

When the disk operation completes, a **TRAP_DISK** is thrown.

To make life interesting, we have tried to ensure that the disk operation duration is dominated by seek time.

The actual disk is a file that Yalnix creates in your directory called **DISK**. If there's one there already, it will use that.

2.4 Interrupts, Exceptions, and Traps

Interrupts, exceptions, and traps are all handled in a similar way by the hardware—it switches into kernel mode and calls 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 (as mentioned in Section 2.3.1 above).
- **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 (as mentioned in Section 2.3.2 above).
- **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 (as mentioned in Section 2.3.2 above).
- **TRAP_DISK**: This interrupt is generated by the disk when it completes an operation (as mentioned in Section 2.3.3 above).

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.

Each of these functions is given a single argument, a pointer to a **UserContext** (see Section 2.5 below), and returns **void**.

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.

2.5 User Context and Trap Handlers

The **UserContext** structure contains the hardware state of the currently running user process (including the value of the program counter, stack pointer, and general registers).

When a trap occurs, the hardware constructs a copy of the **UserContext** and passes its address to the trap handler. When the trap handler returns, the hardware restores whatever **UserContext** is at that address and resumes execution in user mode.

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 2.2.
- **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.

In order to switch contexts from one user process to another, the kernel must copy the running process's User Context (not just the address) 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 2.2: 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 2.4
TRAP_MATH	Type of math error
TRAP_TTY_TRANSMIT	Terminal number causing interrupt
TRAP_TTY_RECEIVE	Terminal number causing interrupt
TRAP_DISK	Undefined

2.6 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 3

The OS Spec

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.

3.1 Yalnix Syscalls

User processes request services from the kernel by executing a "syscall" (sometimes called a "system call" or "kernel call").

The include file **`yalnix.h`** defines function prototypes for the interface for all Yalnix syscalls. This file also defines a constant "syscall number" each type of syscall, used to indicate to the kernel the specific syscall 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 3.2.

3.1.1 Basic Process Coordination

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

  `Fork` is how 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 syscall 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

  ```
  int 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 longer a 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 will be 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.

  If the caller has an exited child whose information has not yet been collected via **Wait**, then this call will return immediately with that information.

  If the calling process has no remaining child processes (exited or running), then this call returns immediately, with **ERROR**.

  Otherwise, the calling process blocks until its next child calls exits or is aborted; then, the call returns with the exit information of that child.

  On success, the process ID of the child process is returned. If **status_ptr** is not null, the exit status of the child is copied to that address.

- **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 at least **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.

3.1.2 I/O Syscalls

- **`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.

  If there are sufficient unread bytes already waiting, the call will return right away, with those.

  Otherwise, 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.

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`** syscall 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.")

3.1.3 IPC Syscalls

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

  Create a new pipe; save its identifier at **`*pipe_idp`**. (See the header files for the length of the pipe's internal buffer.) 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`**, following the standard semantics:

  - If the pipe is empty, then block the caller.
  - If the pipe has **`plen`** $\leq$ **`len`** unread bytes, give all of them to the caller and return.
  - If the pipe has **`plen`** $>$ **`len`** unread bytes, give the first **`len`** bytes to caller and return. Retain the unread **`plen`** $-$ **`len`** bytes in the pipe.

  In case of any error, the value **`ERROR`** is returned. Otherwise, the return value is 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. (As the pipe is a FIFO buffer, these bytes should be appended to the sequence of unread bytes currently in the pipe.) Return as soon as you get the bytes into the buffer. In case of any error, the value **ERROR** is returned. Otherwise, return the number of bytes written.

3.1.4 Synchronization Syscalls

In Yalnix, the synchronization calls operate on *processes*, unlike the pthreads calls you used in Project 2 in class.

- **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)**

  The kernel-level process releases the lock identified by **lock_id** and waits on the condition variable indentified by **cvar_id**. When the kernel-level process wakes up (e.g., because the condition variable was signaled), it re-acquires the lock. (Use Mesa-style semantics.)

  When the lock is finally acquired, the call returns to userland.

  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.

3.2 Interrupt, Exception, and Trap Handling

As described in Chapter 2, 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.

This is what the OS should do for the following traps:

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

  The arguments to the syscall will be found in the registers, starting with **regs[0]**.

  The return value from the syscall 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.)

  If there are no runnable processes, dispatch idle.

- **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. 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. (For more discussion, see Section 3.5.2 below.)

  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** syscall by some user process.

- **TRAP_TTY_TRANSMIT**: This interrupt signifies that a 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** syscall, as necessary; also start the next terminal output on this terminal, if any.

- **TRAP_DISK**: Your OS can ignore these traps, unless you've decided to implement extra functionality involving the disk.

3.3 The Syscall Library

As described above, a **TRAP_KERNEL** occurs when a Yalnix user process requests a syscall 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 (unless you use inline assembly). 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 syscall in Section 3.1. The trap instruction generates a trap to the hardware, which invokes the kernel using the **TRAP_KERNEL** vector from the interrupt vector table.

On the way to the kernel trap, each syscall library wrapper makes sure all the arguments in the right **regs**; on the way back, if there's a return code, the wrapper fetches the return code from **regs[0]**.

(In case it's of interest, **yalnix_framework/etc/yuserlib/yuser/calls.c** is the actual code for these library wrappers.)

3.4 On Process Death

When the kernel aborts a process, your kernel should **TracePrintf** a message at level 0, 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** syscall (as described in Section 3.1) should be the value **ERROR**.

3.5 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.

3.5.1 Initializing Virtual Memory

As described in Chapter 2, 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.

Checkpoint 2 below (Section 8.2) walks through the steps required for how the kernel initiales virtual memory.

3.5.2 User Memory Mangement

Heap User code links to **malloc** and friends, which will use the **Brk** syscall to grow or shrink the heap, as needed. The kernel should not let the heap shrink below the original user data pages, or grow into the user stack.

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 3.2, 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.

Red Zone It's a good idea for the kernel to leave at at least one page unmapped (with the valid bit in its page table zeroed) between the heap 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.

3.5.3 Kernel Memory Management

Heap Just as the userland **`malloc`** library will invoke the **`Brk`** syscall when it needs to adjust the user heap, the kernelland **`malloc`** library will invoke the function **`SetKernelBrk`** when it needs to adjust the kernel heap. Checkpoint 2 (Section 8.2) below gives more information on **`SetKernelBrk`**.

Stack 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.

The kernel stack for each process is a fixed maximum size, **`KERNEL_STACK_MAXSIZE`** bytes. (This avoids the 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 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.

See Chapter 4 and Checkpoint 3 (Section 8.3) for more discussion on managing the kernel stack mappings.

Red Zone Leaving one or two pages below the kernel stack as unmapped can be helpful; as you will see in Checkpoint 3 (Section 8.3).

Chapter 4

Kernel Context Switching

Recall, from class, that we have two types of switching going on. (See Figure 4.1.)

- 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. The stack changes; the contents of the pc, sp, and all the other registers change.

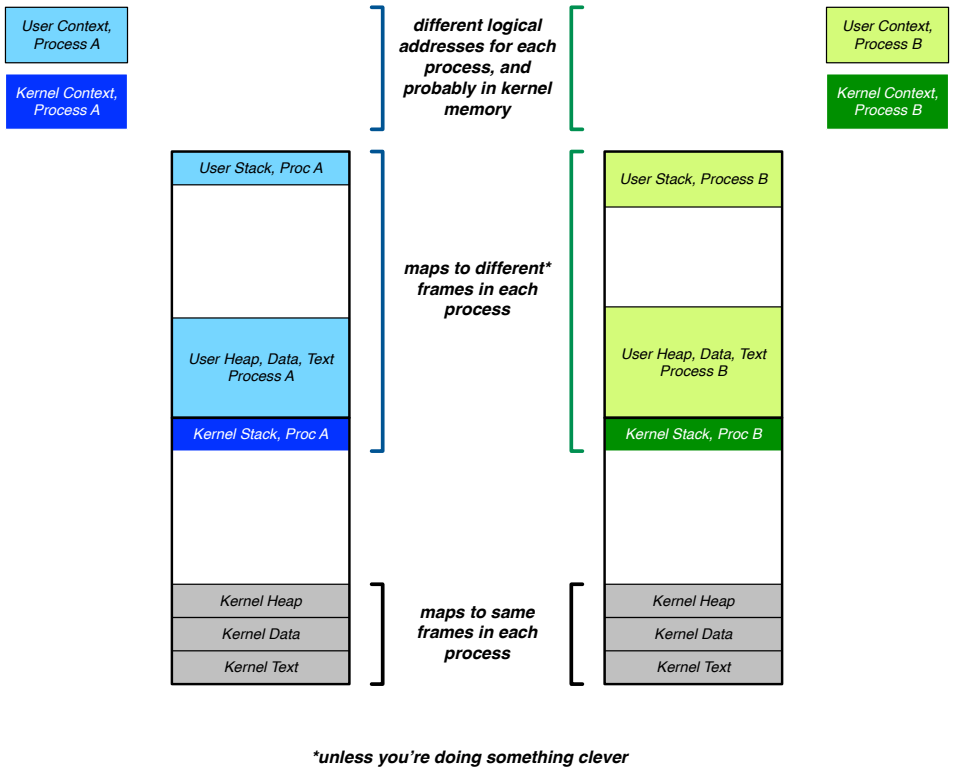

Figure 4.1: Each process has its own userland address space and user context, and its own kernel context and kernel stack. Traps, etc., switch from user to kernel; the kernel may then change from one process to another.

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 physical frame numbers 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.

4.1 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.

In your kernel, you will need **`KernelContextSwitch`** to perform two distinct tasks:

- switching (in kernel-level) from one process to another (Section 4.2 below)
- cloning a new (kernel-level) process in the first place, so there's something to switch to (Section 4.3 below).

For clarity, we recommend you write a different function for each of these two tasks.

4.2 Switching Between Processes

For example, to carry out context switches (see Figure 4.2) you might write a function **`KCSwitch`**, that takes three arguments:

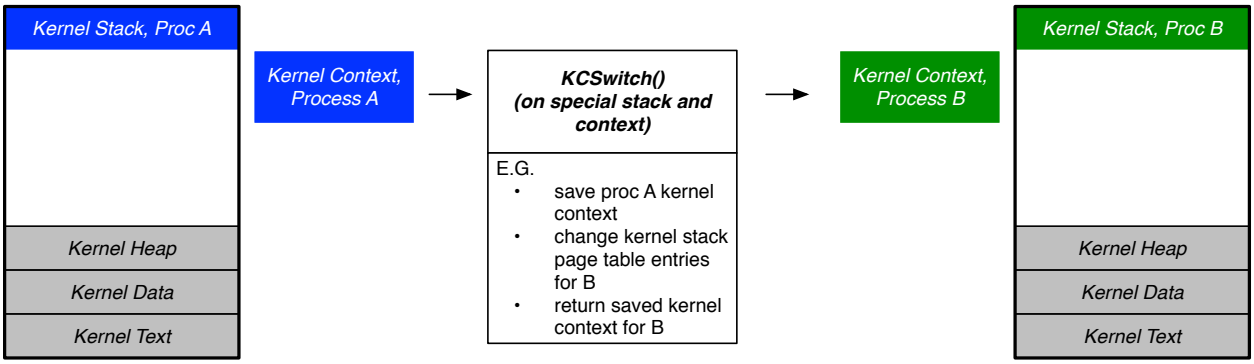

Figure 4.2: The kernel may use ***`KernelContextSwitch()`*** *to suspend one kernelland process and start/resume another.*

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

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

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

`KCSwitch` is then launched, fed with a pointer to a temporary copy of the current kernel context, and these two PCB pointers. Since **`KCSwitch`** is called while not using the normal kernel registers or stack, **`KCSwitch`** can easily and safely do what it needs to do to complete the context switch. As part of this work, it might copy the bytes of 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 **`KCSwitch`** 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:

1. A calls **`KernelContextSwitch`**
2. The support framework executes **`KCSwitch`** with A's arguments.
3. B magically starts running
4. B calls **`KernelContextSwitch`**
5. The support framework executes **`KCSwitch`** with B's arguments.
6. A then resumes running, as if just returned from its **`KernelContextSwitch`** call in Step 1.

If things are successful, the return value of **`KernelContextSwitch`** in is 0. If, instead, any error occurs, then the function 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.

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).

4.3 Cloning a Process

Of course, to switch for the first time to a process B, process B must have a kernel context already.

Hence, you will need to write:

```
KernelContext *KCCopy(KernelContext *kc_in,
                      void          *new_pcb_p,
                      void          *not_used)
```

See Figure 4.3.

KCCopy will simply copy the kernel context from ***kc_in** into the new pcb, and copy the contents of the current kernel stack into the frames that have been allocated for the new process's kernel stack. However, it will then return **kc_in**.

Process A, when it wants to clone B, might to do this:

```
TracePrintf(0,"About to clone A into B\n");
rc = KernelContextSwitch(KCCopy, B_PCB_p, NULL);
TracePrintf(0,"Back from the clone---am I A or B?\n");
```

After A returns from the call, it will execute the second traceprint. However, when B first wakes up, it will *also* execute that second traceprint.

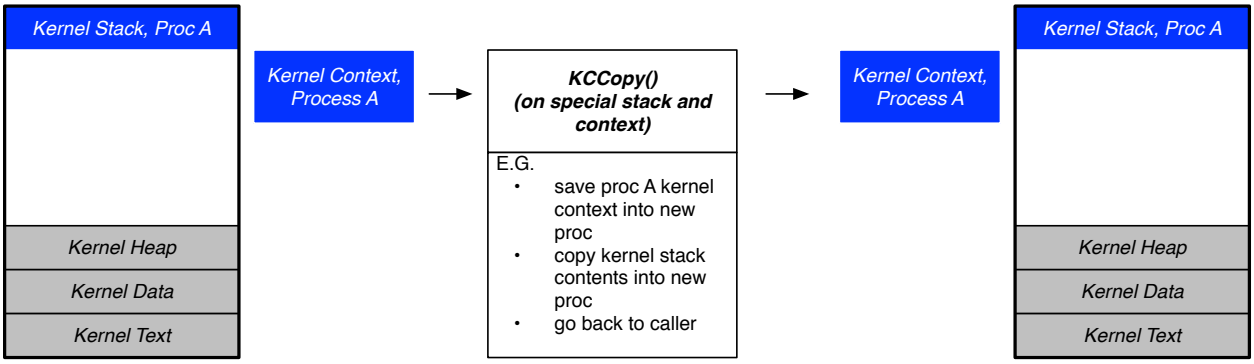

*Figure 4.3: The kernel may use **KernelContextSwitch()** to copy the current kernelland context into a new process.*

Chapter 5

Building and Running the Code

5.1 The Environment

As noted in Chapter 1, Yalnix—the kernel, the userland, and the hardware support code—is really just a userland process running on Linux on an Intel machine. Because "Linux" is a moving target, we've set up some properly configured and tested environments for you—see Chapter 1.

5.2 Compiling

The file **`yalnix_framwork/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.

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.

5.3 Traceprinting

All three components of the system—your kernel code, your user code, and the underlying hardware code—have access to a function:

```
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 (see below). 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`**.

Traceprints are sent to **`stderr`**, as well as to a file called **`TRACE`** (although the latter can be changed; see below).

Tracing in your code 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 (for an example, see **`ENTER`** and **`LEAVE`** in **`hardware.h`**), and put a higher-level **`TracePrintf`** when important actions occur.

Tracing in the hardware As of Spring 2020, the hardware support software, at trace level 1 or higher, will print whenever any of the principal transitions in Figure 1.1 occur. See Section 6.1 for more discussion.

5.4 Running Yalnix

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` *(yalnix options) (KernelStart options)*

5.4.1 KernelStart Options

If these are blank, then your kernel (from Checkpoint 3 onwards) will look for a executable called "init".

Otherwise, the KernelStart options would be passed into KernelStart as **`cmd_args`**. E.g., if you typed

```
./yalnix alt_init a b c
```

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

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

Your kernel would then look for **`alt_init`** to load as its initial process.

5.4.2 Common Yalnix Options

- **`-x`** : Use the X window system support for the simulated terminals attached to the Yalnix system. The default is not to bother. (You will probably not require **`-x`** until you get to Checkpoint 5.)
- **`-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. 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 default tracing level is 1.
- **`-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. The defult tracing level is 1.
- **`-W`**: This tells yalnix to stop and dump core if the hardware helper (see Chapter 6 below) encounters any significant problems. (Without **`-W`**, the helper will merely traceprint about them; with **`-W`**, you can then use gdb to examine the core and see what caused it.)

5.4.3 Esoteric Yalnix Options

- **`-t`** *tracefile*: Send traceprinnts to *tracefile* instead of **`TRACE`**.
- **`-C NNN`** : The default tick interval is allegedly[1] 400 ms. This option lets you change it to some other number of

[1] "Allegedly," due to the mysteriousness of Linux running on a VM running on your host machine

millisecons (e.g., to speed it up).

- **`-I`***n file*: feed terminal *n* with the data from *file*, as if it were typed there.
- **`-O`***n file*: By default, the output from terminal *n* goes to a file named **`TTYLOG.`***n*. This option lets you change that to *file*.

5.5 Coding

5.5.1 Headers

Your kernel code files should **`#include <ykernel.h>`**.

Your user code files should **`#include <yuser.h>`**.

(See **`yalnix_framework/include`**.)

5.5.2 Libraries

Your kernel and user code may use standard C library functions listed in **`ylib.h`**. 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, such as in **`LoadProgram`**) you should not use other library functions that make system calls. It breaks the simulation. (Hopefully, our build environment will complain if you try this.)

5.5.3 Syscall Handler Names

Note that the syscall prototypes given in Chapter 3 and in **`yalnix.h`** are the form in which Yalnix user-level programs use these syscalls. 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 syscall.

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, because that gets confusing—e.g., is **`Delay()`** the userland library wrapper or the kernel's handler? Instead, we recommend giving the handler a related but distinct name—e.g., **`KernelDelay()`**.

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.)

5.6 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`** at level 0 or 1.

- 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. If the user is passing the address of some userland buffer of a specific length, your kernel must look in the page table to make sure that the *entire area* is readable and/or writable (as appropriate) in that userland before the kernel actually tries to read or write there.

If a C-style character string (null-terminated) is passed to you, you will need to check the address of each byte—including the terminating null. (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 syscall 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 (such as an entry in a page table or the interrupt vector), which would allow a rogue userland program to take over kernel privileges. Alternately, 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.

Chapter 6

Helper Functionality

New to Yalnix 2020 is an expanded set of tools to help the implementer to see what's going on, and to help identify common bugs.

6.1 What the System is Doing

First, we discuss the newly enhanced hardware tracing functionality.

6.1.1 System State

The hardware support software already knows these things:

- the address of the current Region 0 page table (which likely should never change)
- the address of the current Region 1 page table
- the frame numbers of the kernel stack

The latter two items should change each time the kernel changes processes (recall Figure 4.1).

Each process will also have a PID. It would be helpful all around if the hardware support software knew the PID that matched each process. Thus, we ask you to tell the hardware about it:

- When building a new process, your kernel will **`malloc`** a Region 1 page table for it. Your kernel should ask tell the helper software about that and ask for a new PID by calling:

```
int helper_new_pid(struct pte *ptbr1)
```

- When exiting or killing a process (but before tearing down the Region 1 page table), your kernel should inform the helper by calling:

```
void helper_retire_pid(int pid)
```

6.1.2 Transitions

As noted earlier, the helper software will now print (at hardware trace level 1, the default) when each of the transitions in Figure 1.1 happens. Where relevant, at each these it will also print:

- The system state information, above
- What it believes the PID to be
- The pc and sp of where a trap came from (or where it's going back to)
- The memory region where these address live. This may be:
  - Region 0
  - Region 1
  - The special stack for **`KernelContextSwitch`** (Chapter 4)
  - One of the Linux software modules (way higher than Region 1) that make up the hardware support software

6.2 Warning of Common Bugs

The helper software will also help identify common Yalnix implementation bugs, and warn about them by printing "WARNING" messages, at hardware trace level 0. As noted in Section 5.4.2 above, if you invoke **`yalnix`** with command-line option **`-W`**, the helper will stop Yalnix and dump core at these points. (This functionality happens via **`helper_maybort`**, discussed in Section 6.3 below.)

The remainder of this section discusses some of these bugs the helper software warns about.

6.2.1 Heap Corruption

As you recall from class, the *heap* is the memory segment above the data, from which **`malloc`** allocates chunks of memory as needed; **`free`** deallocates these chunks.

It can be very easy to make a mistake when working with **`malloc`**'d space—e.g., writing past the end of the space, or using the space after you've already **`free`**'d it. Unfortunately, such bugs can be very hard to track down—module A might have committed the error, but it may a completely different module B (whose heap space just got corrupted) that manifests incorrect behavior. The developer may then scrutinize the code for module B—which isn't where the error is.

In Yalnix, both your kernel code and user code use a heap library I wrote (**`yalnix_framework/etc/new_heap.c`**), which includes a lot of sanity checking looking for signs of corruption (e.g., placing and then checking canaries between allocated chunks being changed). At principal transitions, the helper code will automatically invoke this sanity checking—and let you know if it detects evidence of heap corruption.

6.2.2 Stale TLB Mappings

As you recall from class, when an MMU translates a logical address to a physical address, it will first see if it has that page-to-frame mapping cached in the TLB. The MMU consults page tables only if the mapping is not already in the TLB.

This gives rise to a common bug scenario in OS implementations:

- the kernel changes a page-frame mapping
- but the TLB has cached an older mapping for that page
- and the kernel fails to flush that older mapping

As with heap corruption, this bug can be tricky to find because the code causing it (the kernel's changing of page-frame mappings) will not be the code manifesting the buggy behavior (the code who's using the new mapping and

finds things don't work right, or—what's worse—the code that was using the old mapping and finds, later, that its storage has changed while it wasn't looking).

Since our helper software can look into the TLB and into what the MMU has been told are the current page tables, it will warn of mismatches.

The solution here, as Section 2.2.5 discusses, is for the kernel to flush stale mappings.

(Note, however, that unnecessarily flushing away valid mappings reduces or eliminates the benefits of caching, so the course staff reserves the right to deduct points for it.)

6.2.3 Bugs in Free Frame Tracking

As part of memory management, the kernel needs to keep track of which frames are in use and which frames are free.

Bugs here can lead to the "action at a distance" phenomonon similar to described above. For a simple example, suppose the kernel inadvertantly assigns the same physical frame to the userland data regions for two different processes A and B. These userland processes may then exhibit buggy (and hard to reproduce behavior), even though the bug is not in their code. (For example, the programmer might see B's data getting corrupted and then spend time hunting for wild pointers in B.... except that's not where the problem is.)

Inadvertantly reusing a frame in other places (such as user text, or kernel space) may also cause mysterious problems even harder to explain.

Since the helper software knows about all the currently active page tables, it will check and warn if it sees a frame re-used in two places.

Note that if your kernel implementation is trying some advanced functionality, such as copy-on-write memory or shared memory (see Chapter 9) then the helper may give false positives here—since you may be intentionally sharing one or more frames across different userlands.

6.2.4 Re-Using Kernel Stack Frames

Each process has its own kernel stack (again, recall Figure 4.1).

A buggy kernel may inadvertantly do things such as change the kernel stack frames partway through a process's execution or inadvertantly use the wrong process's kernel stack, which could lead to strange behavior (such as failure to wake up correctly after going to sleep).

The helper software will keep of track what kernel stack frames belong to each process, and will warn you if it sees any funny business.

6.2.5 Mismatches between Kernel Context and Kernel Stack Contents

As Chapter 4 discussed, the kernel context and the kernel stack contents are a matched pair. Breaking this match can lead to vexing bugs.

Two examples of potential errors:

- When creating a new process, **`KCCopy`** may bungle the copying of the old kernel stack.
- When creating a new process, the kernel may correctly copy the current kernel stack, but do it outside of **`KCCopy`**, resulting in stack contents that do not match the kernel context established inside **`KCCopy`**.

The helper software will issue a warning if a **`KernelContextSwitch`** invocation returns a KernelContext not matching the current kernel stack contents. (In case you're curious: we do this by secretly saving a checksum of the kernel stack contents.)

6.3 Extra Debugging Calls

The helper software includes additional function calls your kernel may use to aid in debugging.

- **`void helper_abort(char *msg)`** will cause the kernel to abort and, if possible, dump core. If you find yourself adding a comment

  ```
  /* should not reach here */
  ```

  then you might consider adding a call to **`helper_abort`** so that, if it ever does each there, you will know about it and be able to identify the problem.

- **`void helper_maybort(char *msg)`** will cause the kernel to stop and dump core if the **`-W`** option was used on the command line when invoking **`yalnix`**.

- **`void helper_check_heap(char *msg)`**, if invoked in the kernel, will do sanity checking on the kernel heap and warn of any corruption; if invoked in userland, it will sanity check the user heap. In either case, if the **`-W`** option was used on the command line, then **`helper_check_heap`** will halt the system if it finds corruption.

Optional Message If any of the above three calls find a problem and the **`msg`** argument was not **`NULL`**, then the message there will be printed.

Last Known Return Address These helper calls may be invoked directly by your code—or may be invoked several layers down, inside the support software. To assist you—particularly in the latter case—the calls will also display a "return address" that's the last known point inside your code before it got triggered. See Section 7.4.2 for an example.

Chapter 7

Debugging Examples

This chapter shows some examples of using gdb and the helper tools of Chapter 6 to debug your code.

7.1 GDB and Yalnix

To run yalnix with gdb, you need a **`.gdbinit`** telling gdb about some signals yalnix uses.

On our Virtual Machine, the system has already been configured with the proper **`.gdbinit`**. If you're working with Yalnix on a Thayer Linux box be sure to copy **`yalnix_framework/sample/dot.gdbint`** into **`~/.gdbinit`**.

7.2 Coredumps

Our Virtual Machine has been configured to dump core at **`~/core`**, even when running inside a shared folder.

On the Thayer machines, make sure your account is configured to allow core dumps. Circa Fall 2022, your coredumps will live in **`/var/lib/apport/coredump/`** with a mysterious name that includes your userid. The invocation

```
cp -f `ls -t /var/lib/apport/coredump/*$USER* | head -1` ~/core
```

will grab the most recent one and leave it at **`~/core`**, so you can access it more easily with gdb. (So, you might want to set up a getcore alias in your shell profile.) (Also note that the support software will you tell you it dumped core at **`~/core`** even though Thayer leave it somewhere else.)

7.3 Kernel Crashes

If your kernel touches an illegal address, you will likely get a "Big Green Screen O' Death," such as the following:

```
| The Yalnix kernel has encountered a memory error at address 0x00000004 [[region 0]]
| SEGV fault---check the page tables
| We will now stop pretending to be real hardware
| and try to generate a core dump (in ~/core) of your kernel.
| It may be useful to look at this core file with a gdb

| (Rembember that the bt command will tell you how the kernel got here)
|


 BIG GREEN SCREEN O' DEATH!!!!!!

Segmentation fault (core dumped)
```

In a case like this, try invoking **`gdb yalnix`** and then looking at the core. For example:

```
(gdb) core ~/core
[New LWP 3782]
Core was generated by `./yalnix -n test/hello'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  HandleTrapClock (ucp=0xff728) at traps.c:717
717    ready_queue->next->prev = NULL;
```

What's wrong with that line of code? Let's look at the variables.

```
(gdb) print ready_queue
$1 = (node_t *) 0x2e86c
(gdb) print *ready_queue
$2 = {next = 0x0, prev = 0x0, payload = 0x0}
```

Aha! We were dereferencing a null pointer—we should have first checked if **`ready_queue->next`** had something.

7.4 Kernel Heap Corruption

Suppose the helper software detects a kernel heap corruption, and since we were running with **`-W`**, the kernel aborted at that point. We might see something like this:

```
User Prog hello from parent
Hardware  ----
Hardware  | Trap Alarm clock
Hardware  | From usermode  SP [region 1] 0x1ffda0 | PC [vdso] 0xf7f37169
Hardware  | pid 1 | PTBR1 0x02a0a4 | kstack 126 127
Hardware  | Calling Yalnix handler 0x23164 for trap TRAP_CLOCK
Hardware  ----
Hardware  | KERNEL HEAP CORRUPTION WARNING: heap block at 0x2ec80 has corrupted magic
number.
Hardware  | helper_maybort("TracePrintf"), ret addr = 0x231ae [region 0].
Abort (core dumped)
```

What do we do?

7.4.1 The Tricky Route

The standard approach with gdb would be to look at the core file.

```
(gdb) core ~/core
[New LWP 5833]
Core was generated by `./yalnix -W test/hello'.
Program terminated with signal SIGABRT, Aborted.
#0  0xf7f37169 in __kernel_vsyscall ()
```

But **`__kernel_vsyscall()`** is a function you haven't heard of!

We could use **`bt`** to get the full stack frame.

```
(gdb) bt
#0  0xf7f37169 in __kernel_vsyscall ()                                  support software
#1  0xf7d5a226 in __libc_signal_restore_set (set=0xff4fc)
    at ../sysdeps/unix/sysv/linux/internal-signals.h:84
#2  __GI_raise (sig=6) at ../sysdeps/unix/sysv/linux/raise.c:48
#3  0xf7edf9fd in helper_maybort (msg=0x264c0 "TracePrintf") at exception.c:208
#4  0xf7edca08 in sanity_check (bp=0x2ec80, msg=0x264c0 "TracePrintf")
    at new_heap.c:117
#5  0xf7edcb8a in helper_check_heap (msg=0x264c0 "TracePrintf")
    at new_heap.c:174

#6  0x0002423a in TracePrintf (level=0, fmt=0x25fd7 "ticks %d\n") at load.c:417

#7  0x000231ae in HandleTrapClock (ucp=0xff728) at traps.c:713

#8  0xf7ee12d0 in RealSignalHandler (sig=14, si=0xffb0c, context=0xffb8c)
    at exception.c:1276
#9  0xf7ee16f7 in SignalHandler (sig=14, si=0xffb0c, context=0xffb8c)
    at exception.c:1541
#10 <signal handler called>
#11 0xf7f37169 in __kernel_vsyscall ()                                  support software
#12 0xf7d5a5b8 in __GI___sigsuspend (set=0x1ffe50)
    at ../sysdeps/unix/sysv/linux/sigsuspend.c:26
#13 0xf7ee314a in Pause () at misc.c:87

#14 0x00102144 in ?? ()                                  Region 1 addresses...
#15 0x001002b0 in ?? ()                                  what might this be? :)
#16 0x001002d6 in ?? ()
```

With enough examination, this can start to make sense.

- In Stack Level 7, there's a function—**`HandleTrapClock`**—from the example kernel code we're using.
- Stack Level 6 shows **`TracePrintf`**. Our **`HandleTrapClock`** called that, which then called **`helper_check_heap`** (Stack Level 5), which found a problem—and went down several steps to acually stop the kernel.
- Stack Level 8 and beyond is mysterious.

So, after all this work, we could then use gdb's **up** to get up to the **HandleTrapClock** level, and then investigate.

7.4.2 The Easier Route

A quicker way to find the problem is to use the "last known return address" the helper tells us (recall Section 6.3).

```
User Prog hello from parent
Hardware  ----
Hardware  | Trap Alarm clock
Hardware  | From usermode  SP [region 1] 0x1ffda0 | PC [vdso] 0xf7f37169
Hardware  | pid 1 | PTBR1 0x02a0a4 | kstack 126 127
Hardware  | Calling Yalnix handler 0x23164 for trap TRAP_CLOCK
Hardware  ----
Hardware  | KERNEL HEAP CORRUPTION WARNING: heap block at 0x2ec80 has corrupted magic
number.
Hardware  | helper_maybort("TracePrintf"), ret addr = 0x231ae [region 0].
Abort (core dumped)
```

Aha—what's at **0x231ae**?

```
(gdb) list *0x231ae
0x231ae is in HandleTrapClock (traps.c:713).
707    ptr = malloc(10);
708
709    memset(ptr,0x00,100);
710
711    ticks++;
712
713    TracePrintf(0,"ticks %d\n", ticks);
```

It was when executing the **TracePrintf** call at line 713 that the helper found corruption!

What might be doing that? There's a **memset** shortly before writing to a **ptr** that we might happen to know was from **malloc**. Let's try bracketing that **memset** with **helper_check_heap** calls.

```
helper_check_heap("before");
memset(ptr,0x00,100);
helper_check_heap("after");
```

Re-running, we see the first call passes, but the second call fails.

```
User Prog hello from parent
Hardware  ----
Hardware  | Trap Alarm clock
Hardware  | From usermode  SP [region 1] 0x1ffda0 | PC [vdso] 0xf7f37169
Hardware  | pid 1 | PTBR1 0x02a0a4 | kstack 126 127
Hardware  | Calling Yalnix handler 0x23164 for trap TRAP_CLOCK
Hardware  ----
Hardware  | KERNEL HEAP CORRUPTION WARNING: heap block at 0x2ec80 has corrupted magic
number.
Hardware  | helper_maybort("TracePrintf"), ret addr = 0x231ae [region 0].
Abort (core dumped)
```

It must be that **`memset`**! (And, in this case, a quick look at its arguments will tell you why.)

7.5 Stale TLB Mappings

As Chapter 6 discussed, the helper sofware will also help detect stale TLB mappings. Suppose our kernel had a bug with that, and we were running with **`-W`** and got an abort.

```
Hardware      | WriteRegister(REG_PTBR1,0x02e870)
Hardware      | WriteRegister(REG_TLB_FLUSH,TLB_FLUSH_0)
Hardware  ----
Hardware  | On [kcs stack] invoking  MyKCS function 0x020fb6
Hardware  | From kernelmode SP [region 0] 0xff510 | PC [libhardware.so] 0xf7efe795
Hardware  | pid 2 | PTBR1 0x02e870 | kstack 126 127
Hardware  ----
Hardware      | WriteRegister(REG_TLB_FLUSH,TLB_FLUSH_KSTACK)
Hardware  ----
Hardware  | On [kcs stack] returning from MyKCS function 0x020fb6
Hardware  | Going to kernelmode SP [region 0] 0xff520 | PC [libhardware.so] 0xf7efe795
Hardware  | pid 2 | PTBR1 0x02e870 | kstack 048 049
Hardware  ----
Hardware
Hardware  | Back from Yalnix handler 0x233b2 for trap TRAP_KERNEL (Fork)
Hardware  | HELPER WARNING: stale TLB frames in Region 1
Hardware  | helper_maybort("stale TLB frames"), ret addr = 0xf7effbee [libhardware.so].
Abort (core dumped)
[cs58@localhost spring2020]$
```

Here, we don't even need to go to gdb—the first two lines in the above screenshot tell us what went wrong! Our kernel changed the Region 1 page table but mistakenly flushed Region 0 instead of Region 1.

7.6 Unreachable Code, in Theory

Chapter 6 also discussed the use of **`helper_abort`** to make sure that, if your kernel ever reaches some line of code it shouldn't, you know about it.

For example, the syscall handler for **`Exit`** should never return. Adding a call to **`helper_abort`** will make sure the kernel stops and dumps core if we ever actually reach that line.

```
void SysExit(UserContext *ucp) {

  TerminateCurrent(ucp->regs[0]);

  //  LaunchNewProcess(current,ucp);
  TracePrintf(0,"should not reach here!!!\n");
  helper_abort("SysExit failure");

}
```

In this instrumented example, we actually hit that line!

```
Hardware  ----
Hardware  | Syscall trap Exit
Hardware  |
...
Yalnix    should not reach here!!!
Hardware  | helper_abort("SysExit failure"), ret addr = 0x221c1 [region 0].
Abort (core dumped)
```

In this case, the cause is obvious: in the **`Exit`** handler, the code to launch a new process has been "inadvertently" commented out.

7.7 Kernel Breakpoints

Running yalnix under gdb lets us do standard gdb things, such as set breakpoints in the kernel.

For example, if **`HandleClockTrap`** is the name of the clock trap handler in our kernel, then if we tell gdb **`b HandleClockTrap`**, the kernel will stop each time it handles a clock trap, and we can do things like check that our various process queues all make sense.

7.8 Seeing into User Land

Continuing the example from Section 7.7 above, suppose we have hit a breakpoint at **`HandleClockTrap`**, our clock trap handler. Using gdb to look at the stack gives some mysterious results:

```
(gdb) bt
#0  HandleTrapClock (ucp=0xff728) at traps.c:725

#1  0xf7f7b2d0 in RealSignalHandler (sig=14, si=0xffb0c, context=0xffb8c)
    at exception.c:1276
#2  0xf7f7b6f7 in SignalHandler (sig=14, si=0xffb0c, context=0xffb8c)
    at exception.c:1541
#3  <signal handler called>
#4  0xf7fd1169 in __kernel_vsyscall ()
#5  0xf7df45b8 in __GI___sigsuspend (set=0x1ffe50)
    at ../sysdeps/unix/sysv/linux/sigsuspend.c:26
#6  0xf7f7d14a in Pause () at misc.c:87

#7  0x00102144 in ?? ()
#8  0x001002b0 in ?? ()
#9  0x001002d6 in ?? ()
#10 0x00100316 in ?? ()
#11 0x00100156 in ?? ()
```

At Stack Level 0, we see our clock trap handler. Stack Levels 1 through 6 show various levels of the support software. However, Stack Level 7 onwards show "??." What's going on?

If you've been working with the header files a lot, you might recognize that these "??" addresses belong to Region 1. If we know what the name of the Region 1 executable is, we can tell GDB about that via **`add-symbol-file`**—and gdb will tell us!

```
(gdb) add-symbol-file test/hello
add symbol table from file "test/hello"
(y or n) y
Reading symbols from test/hello...
(gdb) bt
#0  HandleTrapClock (ucp=0xff728) at traps.c:725
#1  0xf7f7b2d0 in RealSignalHandler (sig=14, si=0xffb0c, context=0xffb8c)
    at exception.c:1276
#2  0xf7f7b6f7 in SignalHandler (sig=14, si=0xffb0c, context=0xffb8c)
    at exception.c:1541
#3  <signal handler called>
#4  0xf7fd1169 in __kernel_vsyscall ()
#5  0xf7df45b8 in __GI___sigsuspend (set=0x1ffe50)
    at ../sysdeps/unix/sysv/linux/sigsuspend.c:26
#6  0xf7f7d14a in Pause () at misc.c:87

#7  0x00102144 in Pause () at yuser/libc.c:385

#8  0x001002b0 in sub () at test/hello.c:32
#9  0x001002d6 in main () at test/hello.c:53
#10 0x00100316 in __libc_start_main (main=0x1002cb <main>, argc=0,
    ubp_av=0x1fff74, init=0x1002de <__libc_csu_init>,
    fini=0x1002d8 <__libc_csu_fini>, rtld_fini=0x0, stack_end=0x1fff6c)
    at common/start.c:26
#11 0x00100156 in _start ()
```

And now we see what was going on in Region 1—all the way beyond **`main`** to the library starter code—when the breakpoint was triggered.

7.9 User Breakpoints

In Section 7.8 above, we used gdb to look into yalnix userland. A natural question is: can we set breakpoints there?

The answer, surprisingly, is yes, at least for now.[1]

To do so:

- You need to use gdb's **hb** ("hardware breakpoint") command rather than usual **b**.
- Gdb has to have already started running yalnix. This is already the case in Section 7.8 above. If you were starting from scratch, you would need to do something like **b KernelStart**, then start running, and then set up your userland breakpoints when the code is stopped in **KernelStart**.

For example, suppose this subroutine is in our userland code:

```
int sub(void)
{
  int rc;
  void *ptr;

  rc = Fork();

  if (rc) {
    while (1) {
      TracePrintf(0,"hello from parent\n");
      Pause();
    }
  } else {

    while(1) {
      TracePrintf(0,"hello from child\n");
      Pause();
    }

  }

}
```

If we put a **b** in our kernel clock trap handler, and then put **hb**s in each of the **while** loops in userland, we would see things bounce between the parent, the clock handler, the child, the clock handler, and so on.

7.10 User Heap Corruption

Section 7.4 above showed an example of detecting and diagnosing kernel heap corruption. Section 7.8 above showed how to convince gdb to look into yalnix userland.

Yes, we can use these techniques together to diagnose userland heap corruption. For example, we might running with **-W** and the following crash:

[1] I would be disappointed but not surprised if this stopped working.

```
Hardware  | Back from Yalnix handler 0x2339e for trap TRAP_KERNEL (Brk)
Hardware  | Off to usermode SP [region 1] 0x1ffe54 | PC [region 1] 0x10057b
Hardware  | pid 1 | PTBR1 0x02a044 | kstack 126 127
Hardware  ----
Hardware      ----
Hardware      | TLB miss p:129  -> pfn:029
Hardware      | In SP [region 1] 0x1ff7b8 | PC [libc-2.30.so] 0xf7daf7e3
Hardware      | pid 1 | PTBR1 0x02a044 | kstack 126 127
Hardware      ----
User Prog | USER HEAP CORRUPTION WARNING: heap block at 0x10779c has corrupted magic number.
Hardware  | userland helper_maybort("into sycall"), ret addr = 0x1002b2 [region 1].
Abort (core dumped)
```

On the way into a syscall, the helper has detected user heap corruption. If we invoke gdb and tell it about the userland symbols, we can then see where this was in the code.

```
(gdb) add-symbol-file test/hello
add symbol table from file "test/hello"
(y or n) y
Reading symbols from test/hello...
(gdb) list *0x1002b2
0x1002b2 is in sub (test/hello.c:19).
14     memset(ptr,0x00,100);
15
16
17
18
19     rc = Fork();
20
21     if (rc) {
22       while (1) {
23         TracePrintf(0,"hello from parent\n");
(gdb)
```

The **`Fork`** syscall handler triggered by line 19 found that the user heap was corrupted. What might have done that? Maybe it's that **`memset`** again.....

Chapter 8

The Checkpoint Sequence

This is a large project, with a significant amount of time to work on it before the due date. To help you complete the project by the due date, the schedule for this project includes *project checkpoints* as intermediate milestone points within the project.

8.1 Checkpoint 1: Pseudocode

By this point, you should have gotten comfortable with your build environment figured out how to move files back and forth between your personal machine and the enivornment. Additionally, you should have:

- examined **`yalnix_framework/include`**
- sketched all the kernel data structures
- pseudo-coded all the traps, syscall, and major functions

Clearly, this pseudo-code will not be final! However, from our experience, if you don't work through the flow of *all* the code first, your data structures will be farther from correct. 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.

8.2 Checkpoint 2: Idle

Your kernel should boot, which involves setting up virtual memory and the interrupt vector. Your kernel should then run idle in user mode.

For this checkpoint, you will probably run as **`./yalnix`**, maybe with **`-W`** as well.

8.2.1 Booting

You need to write:

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

- 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. (Recall Section 5.4.1.)
- 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.
- Thhe **uctxt** argument is a pointer to an initial **UserContext** structure.

8.2.2 Virtual Memory

During boot, you will need to do the following:

- Set up a way to track free frames.[1]
- Set up the initial Region 0 page table. To help you with this, the build process will tell your kernel about three addresses:
  - **void *_kernel_data_start**: the lowest address in the kernel data region.
  - **void *_kernel_data_end**: the lowest address not in use by the kernel's instructions and global data, at boot time.
  - **void *_kernel_orig_brk**: the address the kernel library believes is its brk, at boot time.

  See **yalnix.h**.
- Set up a Region 1 page table for idle. This should have one valid page, for idle's user stack.
- Write:

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

  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, **SetKernelBrk** only needs to track if and by how much the kernel brk is being raised beyond **_kernel_orig_brk**. After VM is enabled, **SetKernelBrk** acts like the standard **Brk**, but for uuserland.

  SetKernelBrk should return 0 if successful, and **ERROR** if not. (But be warned: that **ERROR** may lead to a kernel **malloc** call returning **NULL**.)
- Enable virtual memory. Note that if the kernel brk has been raised since you built your Region 0 page table, you need to adjust the page table appropriately *before* turning VM on—otherwise, things you malloc'd may mysteriously disappear.

8.2.3 Traps

You need to set up the interrupt vector (Section 2.4)—the table of addresses of the handlers the hardware should call for the various traps.

At this point, I recommend three handler functions:

[1] A bit vector is probably the most concise sane way of doing it. The slickest way is to track them as a linked list within the free frames themselves—hence, zero space overhead, effectively—but this is tricky. Bugs in tracking of free frames can be very hard to track down, so I recommend starting with something simple and easily correct.

- One for **TRAP_CLOCK**, that traceprints when there's a clock trap.
- One for **TRAP_KERNEL**, that traceprints (in hex) the code of the syscall. (See **yalnix.h**—these are much more readable in hex.)
- And a general "this trap is not yet handled" handler, in all the other entries. (If instead you leave an entry zeros but that trap occurs, there will be trouble.)

8.2.4 Idle

Your kernel is already running as its first process. Make that formal by creating an **idlePCB** that keeps track of this identity:

- its region 1 page table
- its kernel stack frames
- a UserContext (from the the **uctxt** argument to **KernelStart**))
- a pid (from **helper_new_pid()**, Section 6.3)

Write simple idle function in the kernel text.

```
void
DoIdle(void) {

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

}
```

(Recall from Section 2.6 that **Pause()** pauses the machine until the next clock trap.)

In the UserContext in **idlePCB**, set the pc to point to this code and the sp to point towards the top of the user stack you set up.

Cook things so that when you return to user mode at the end of **KernelStart**, you return to this modified UserContext.

I recommend a traceprint "leaving KernelStart" at the end of **KernelStart**.

8.3 Checkpoint 3: Init

For this checkpoint:

- Your kernel should load an **init** into userland
- For an init with a **while(1)** loop that traceprints and pauses, your kernel should bounce between idle and and init on each clock trap.
- Your kernel should also handle three syscalls: **Brk**, **GetPid**, and **Delay**.

Note that init is going into userland, so you need to add **`init.c`** to the **`USER`** part of the makefile.

For this checkpoint, you will probably run as **`./yalnix`**, maybe with **`-W`** as well. If you write a userland test file **`test.c`**, then you might run as **`./yalnix test`**. If you've added fancier, deeper traceprinting in your kernel, you might specify a higher kernel trace level—e.g., **`-lk 5`**.

Getting to this goal requires several steps.

8.3.1 A New Process

Your **`KernelStart`** needs to create an **`initPCB`**.

You also need to decide:

- Is your **`KernelStart`** idle, cloning into init?
- Or is it init, cloning into idle?

The latter makes more sense in the long run, but requires changing around some things you did in Checkpoint 2. So, for pedagogical purposes, we'll now discuss the former.

Set up the identity for your new **`initPCB`**.

- Its region 1 page table (all invalid)
- new frames for its kernel stack frames
- a UserContext (from the the **`uctxt`** argument to **`KernelStart`**))
- a pid (from **`helper_new_pid()`**)

Then write your **`KCCopy()`** to:

- copy the current KernelContext into **`initPCB`**
- copy the contents of the current kernel stack into the new kernel stack frames in **`initPCB`**

(Recall Section 4.3.)

Copying a page into an unmapped frame? The second bullet above raises the question: "how can I copy the contents of a page into a frame not mapped into my address space—isn't that impossible?" The answer is yes, it is impossible. In order to do it, you need to temporarily map the destination frame into some page. I like to use the page right below the kernel stack.

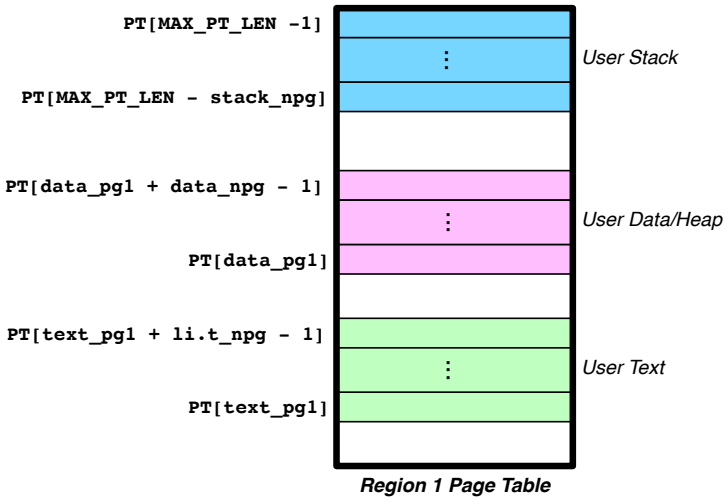

Figure 8.1: How **`LoadProgram`** *lays out the pieces of the userland executable in Region 1, relative to* indices *in the Region 1 page table.*

8.3.2 Loading a Program

Your kernel will need to open the init executable and set it up in Region 1.

In **`yalnix_framework/sample/template.c`**, 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. 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.

Figure 8.1 sketches how **`LoadProgram`** puts things into Region 1.

Note that **`LoadProgram`** loads into the *current* region 1, so you want that to be init's when you run this.

8.3.3 Specifying Init

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.

8.3.4 Changing Processes

In this checkpoint, your clock trap handler is going to need to change between init and idle (that is, unless init is delaying).

That means that (referring to Figure 1.1), your kernel will need to do the following:

- On the way into a handler (**Transition 5**), copy the current UserContext into the PCB of the current proceess.
- On the way back into user mode (**Transition 6**), make sure the hardware is using the region 1 page table for the current process, and copy the UserContext from the current PCB back to the **`uctxt`** address passed to the handler (so that we go back to the right place)
- Invoke your **`KCSwitch()`** function (**Transitions 8 and 9**) to change from the old process to the next process.

So, you now need write your **`KCSwitch()`** to:

- copy the current KernelContext into the old PCB
- change the Region 0 kernel stack mappings to those for the new PCB
- return a pointer to the KernelContext in the new PCB

(Recall Section 4.2.)

8.3.5 Does it really work?

Although the three syscalls for this checkpoint are relatively simple, you should test that they actually work. This probably entails writing some test userland programs, and explaining how they show correct behvavior.

You should also perform and document such testing for the remaining checkpoints.

8.4 Checkpoint 4: Fork, Exec, Wait

For this checkpoint, you need to write handlers for **`Fork()`**, **`Exec()`**, and **`Wait()`**.

You've already done most of the work for **`Fork()`** when you cloned a new process in Checkpoint 3; similarly, your invocation of **`LoadProgram`** in Checkpoint 3 did most of the work for **`Exec()`**.

For **`Wait()`**, you will need to figure out how to keep track of parents and children and zombies and such.

At this point, you should adjust your clock trap handler to implement the round robin scheduling required by the spec, and only dispatch idle if there are no other processes ready.

You should also begin to implement handlers for the remaining traps.

8.5 Checkpoint 5: Terminal I/O

The **`TtyRead`** and **`TtyWrite`** syscalls (Section 3.1.2) should be complete. The remaining traps and exceptions should be implemented.

You now want to start using the **`-x`** option (Section 5.4.2), since you need the terminals to show up.

Note that the functionality of the terminal I/O syscalls require a dance between the syscall handlers and the transmit/receive trap handlers. E.g., when a userland process calls **`TtyWrite`** with length less than **`TERMINAL_MAX_LINE`**, the kernel should:

- block the userland caller and dispatch someone else
- when the terminal is free to write to, invoke the **`TtyTransmit`** instruction
- when the hardware throws the **`TRAP_TTY_TRANSMIT`** to indicate this transmit is complete, the kernel moves the caller back to the ready queue. When the caller gets dispatched next, the userland process will return from **`TtyWrite`**.

Since the **`TtyTransmit`** operation may be going on while the caller is blocked and someone else is in Region 1, so the kernel better use a Region 0 buffer.

If userland calls **`TtyWrite`** with more than **`TERMINAL_MAX_LINE`** bytes, then the kernel may have to go through several **`TtyTransmit`** operations to complete it.

8.6 Final Submission

The remaining syscalls—and all code—should be complete. Do thorough testing.

Look at your work and wonder in amazement at the road you have traveled, and the enlightenment you have achieved.

Chapter 9

Extra Functionality

Students occasionally ask for a richer hardware environment for their Yalnix machine or additional implementation challenges. Students taking the course "for graduate credit" are required to do extra work, which includes some kind of "bonus functionality."

This chapter summarizes a few ideas.

9.1 Semaphores

The libraries contain wrappers for these semaphore syscalls:

- **`int SemInit(int *sem_idp, int val)`**

  Create a new semaphore; save its identifier at **`*sem_idp`**; and set its initial value to be the nonnegative number **`val`**. In case of any error, the value **`ERROR`** is returned.

- **`int SemUp(int sem_id)`**

  Perform an "up" operation on the semaphore with identifier **`sem_id`**. In case of any error, the value **`ERROR`** is returned.

- **`int SemDown(int sem_id)`**

  Perform a "down" operation on the semaphore with identifier **`sem_id`**. In case of any error, the value **`ERROR`** is returned.

If you implement semaphores, you should be sure to extend **`int Reclaim(int id)`** to garbage-collect semaphores as well.

9.2 Other Syscalls

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

`yalnix_framework/etc/yuserlib/yuser/calls.c`

as a reference for how the syscall wrappers are implemented.

If you inspect the headers, you'll see that there also stubs for several other unimplemented calls; you may use these too.

9.3 The Ledyard Bridge

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. One idea: implement a syscall that lets two processes explicitly share the same frame in some page, and store your common bridge data structures there.

9.4 Disk

As noted in Section 2.3.3 above, the hardware includes support for a disk. Maybe implement paging to disk? Or a filesystem? Or both?

Note that **`yalnix_framework/include/filesystem.h`** sketches a filesystem someone wrote once, for a version of Yalnix that used message passing. Feel free to use this header for inspiration.

9.5 Other Ideas

You might also consider:

- **Copy-on-Write** Extend your memory structure to allow sharing of frames via copy-on-write. (How many additional processes does this let you create?)
- **Threads** Support creation of multiple threads within a process. (How will this affect your other blocking syscalls?)
- **ps** Design and implement syscalls that let you interrogate the status of the machine. What processes are running? How many frames do they consume? What's the status of your synchronization primitives and pipes?
- **Or...** What else can you dream up?

Chapter 10

Grading

We will grade the project both by running it and by reading the source code.

10.1 Correctness

Your code should build with neither warning nor error.

Please remember the rules for init:

- Your kernel should first look on the command line for the name of the program to run as "init" and the arguments to feed it (Section 5.4.1, Section 8.3.3).
- If init exits, then the kernel should halt (Section 3.1.1).

These are needed for our testing. If we have to fix your code to comply with these rules, grumbling shall happen and points will be deducted.

We will test for correctness.

- basic functionality
- functionality and grace under stress
- robustness (Section 5.6).

The test programs we will use may include the ones in **`yalnix_framework/sample/test`**—particularly **`torture`**.

We will run your code with all tracelevels at 1. and with the **`-W`** option enabled. If you kernel dumps core, that is not good. (If **`-W`** will give you a false positive for some reason, please let us know beforehand.)

10.2 Software Engineering

We will read the code and evaluate against principles of good software engineering, including testing, efficiency, elegance, clarity, and general slickness.

10.3 My middle name is "Black Thumb"

Note: I usually can get every student project to crash at least once. But there have been exceptions...