Lab # 2
The Yalnix Operating System Kernel
COMP 421/ELEC 421, COMP 521/ELEC 552

Important Dates:

- Thursday, February 15, 2018: Lab 2 project begins.
- March 1–5, 2018: Midterm Exam. The midterm exam will be take-home and must be completed in a single contiguous period of no more than 4 hours. Copies of the exam will be available to be picked up beginning Thursday, March 1, and the completed exam is due by 5:00 PM on Monday, March 5.
- Friday, March 9, 2018: Optional but recommended Lab 2 project milestone checkpoint (see Section 4.4).
- March 10–18, 2018: Spring Break.
- 11:59 PM, Friday, March 30, 2018: Lab 2 project submission deadline.

1 Project Overview

Through this assignment, you will be able to learn how a real operating system kernel works, managing the hardware resources of the computer system and providing services to user processes running on the system. Although this is a long document, please read it through carefully in its entirety, as it contains many details necessary for the project, provides guidance on how to implement different aspects of the project, and will provide answers to many questions you may have as you work on the project. A Table of Contents appears at the end of this document, to help you in finding the sections addressing different topics.

In the project, you will implement an operating system kernel for the Yalnix operating system, running on mythical (simulated) computer system hardware known as the Rice Computer Systems RCS 421. Your solution must be implemented in the C programming language (e.g., not in C++ or other languages).

The Yalnix operating system is similar in many ways to the Unix operating system, with influences also from the V-System, a research operating system written at Stanford University. In particular, Yalnix supports multiple processes, each having their own virtual address space, plus interprocess communication between the processes based on message passing. The system also supports a system console and a set of terminals on which users can do input and output.

Through the magic of the support software provided for your use in this project, a single Linux process on CLEAR will be made to behave like an RCS 421 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 RCS 421 computer, but in reality, everything will be running in user mode in a single Linux process. As in the first project, separate X terminal windows will also be used to simulate the terminals attached to the RCS 421 computer.

The project will be done in groups of 2 students. You are free to choose your own partner for the project. The third project in the class will also be done in groups of 2 students; you may keep the same partner for the third project or change partners between these two projects if you like. If you are having trouble finding a partner for the project, please post a message on Piazza letting others in the class know you are looking for a partner and asking who else is looking for one.

In working on the project with your partner, make sure that you each know what the other is doing, and think ahead to design data structures and approaches that can be shared by both of you. In order to get the
full benefit of the course, both partners in a project group should fully understand your group’s solution to
the project.

*The project must be done on the CLEAR Linux system at Rice.* In particular, the hardware simulation
of the RCS 421 computer system requires certain libraries and executable files that are only installed on
CLEAR and are only compatible with CLEAR.

## 2 The RCS 421 Computer System Hardware

This section describes the hardware architecture of the Rice Computer Systems RCS 421 computer, a myth-
ical computer architecture on which you will implement an operating system kernel in this project. The oper-
ating system, known as Yalnix, is described in Section 3. The RCS 421 computer system has the features
of real hardware but is simplified somewhat to make the project more manageable.

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

### 2.1 Machine Registers

This section defines the general purpose registers and privileged registers of the RCS 421 computer system.

#### 2.1.1 General Purpose Registers

The RCS 421 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.
- **R0** through **R7**, the eight *general registers*, used as accumulators or otherwise to hold temporary
  values during a computation.

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

#### 2.1.2 Privileged Registers

In addition to the general purpose CPU registers, the RCS 421 computer contains a number of privileged
registers, accessible only from kernel mode. Table 1 summarizes these privileged registers.

Details for the use of each of these registers are described in the following sections, as indicated in
Table 1, where the operation of that component of the RCS 421 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.

All values written into or read from these privileged registers must be of type **RCS421RegVal**. This
data type is defined in the file *comp421/hardware.h*. The *comp421/hardware.h* include file is
located in the directory `/clear/courses/comp421/pub/include/comp421`. You can include
the file *comp421/hardware.h* in your source file by using

```c
#include "comp421/hardware.h"
```
Table 1  RCS 421 Privileged Register Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Readable</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG_PTR0</td>
<td>Page Table Register 0</td>
<td>Yes</td>
<td>Section 2.2.4</td>
</tr>
<tr>
<td>REG_PTR1</td>
<td>Page Table Register 1</td>
<td>Yes</td>
<td>Section 2.2.4</td>
</tr>
<tr>
<td>REG_TLB_FLUSH</td>
<td>TLB Flush Register</td>
<td>No</td>
<td>Section 2.2.5</td>
</tr>
<tr>
<td>REG_VM_ENABLE</td>
<td>Virtual Memory Enable Register</td>
<td>Yes</td>
<td>Section 2.2.6</td>
</tr>
<tr>
<td>REG_VECTOR_BASE</td>
<td>Vector Base Register</td>
<td>Yes</td>
<td>Section 2.4.1</td>
</tr>
</tbody>
</table>

#include <comp421/hardware.h>

Do not copy this file into your own directory.

The RCS 421 hardware provides two privileged instructions for reading and writing these privileged machine registers:

- void WriteRegister(int which, RCS421RegVal value)
  Write value into the privileged machine register designated by which.

- RCS421RegVal 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 1, passed as the which argument to the instructions. The file comp421/hardware.h defines the symbolic names for these constants.

As noted above, the values of these registers are represented by the data type RCS421RegVal. You must use a C “cast” to convert other data types (such as addresses) to type RCS421RegVal 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 any type other than an RCS421RegVal.

2.2 Memory Subsystem

2.2.1 Overview

The memory subsystem of the RCS 421 computer supports physical memory size that depends on the amount of hardware memory installed. The physical memory is divided into pages of size PAGESIZE bytes.

The individual pages of this physical memory may be mapped into the address space of a running process through virtual memory supported by the RCS 421 hardware and initialized and controlled by your Yalnix operating system kernel. The hardware component that implements the hardware control for these virtual memory mappings is known as the Memory Management Unit, or MMU. As with the physical memory, the virtual memory is divided into pages of size PAGESIZE, and any page of the virtual address space may be mapped to any page of physical memory through the page table of each running process.

The constant PAGESIZE is defined in comp421/hardware.h. In addition, this include file also defines a number of other constants and macros to make dealing with addresses and page numbers easier:

- PAGESIZE: The size in bytes of each physical memory 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.

• **UP_TO_PAGE(addr)**: A C preprocessor macro that rounds address `addr` up to the next highest page boundary. If `addr` is already on a page boundary, it returns `addr`.

• **DOWN_TO_PAGE(addr)**: A C preprocessor macro that rounds address `addr` down to the next lowest page boundary. If `addr` is already on a page boundary, it returns `addr`.

Due to RCS 421 hardware restrictions (in reality, due to restrictions in the underlying Linux operating system), the bottom `MEM_INVALID_SIZE` bytes (`MEM_INVALID_PAGES` pages) of the address space are inaccessible; specifically, any address in this range cannot be used as an address, as the hardware disallows such addresses. Before virtual memory has been enabled, this means that any physical address in this range of addresses will cause an error. After virtual memory has been enabled, this means that all page table entries (`MEM_INVALID_PAGES` page table entries) corresponding to this address range must be all have a valid bit of 0. By enforcing this restriction, the hardware attempts to ensure that accidental accesses to memory using a NULL pointer are detectable.

Note that the first `MEM_INVALID_PAGES` pages of physical memory do exist and can be used the same as any other page of physical memory, once virtual memory has been enabled. That is, a valid virtual page can be mapped to any page of physical memory, including a physical page with a physical page number less than `MEM_INVALID_PAGES`, since in then accessing such a page, you would be using a virtual address, not directly a physical address, to access the page; as long as the virtual address you use for this is not less than `MEM_INVALID_PAGES`, the hardware will not consider this an error.

In other words, once virtual memory has been enabled, you can use those bottom `MEM_INVALID_PAGES` number of pages. You can’t use as an address any address less than `MEM_INVALID_SIZE`, but once you have enabled virtual memory, you can access those first `MEM_INVALID_PAGES` number of physical pages, not by using their physical address, but instead by mapping some virtual address (greater than or equal to `MEM_INVALID_SIZE`) to one of those physical pages. The restriction is on what you can use as an address, not on what memory you can (in some other way) access.

### 2.2.2 Physical Memory

The beginning address and size of physical memory in the RCS 421 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 RCS 421 hardware design. The value `PMEM_BASE` is defined in `comp421/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. The size of physical memory is unrelated to the size of the virtual address space of the machine.
As described in Section 2.2.1, the physical memory of the machine is divided into pages of size \( \text{PAGESIZE} \). Page numbers in physical memory addresses are often referred to as page frame numbers.

### 2.2.3 Virtual Address Space

The virtual address space of the machine is divided into two regions, called *Region 0* and *Region 1*. Region 0 is used by user processes executing on the operating system, and Region 1 is used by the operating system kernel. In other words, Region 0 will contain the address space of each process, and Region 1 will be managed by your kernel to hold kernel state.

This division of the virtual address space into two regions is illustrated in Figure 1. The beginning virtual address of Region 0 is defined as \( \text{VMEM}_0\_\text{BASE} \), the limit virtual address of Region 0 is defined as \( \text{VMEM}_0\_\text{LIMIT} \), and the size of Region 0 is defined as \( \text{VMEM}_0\_\text{SIZE} \). Likewise, the beginning virtual address of Region 1 is defined as \( \text{VMEM}_1\_\text{BASE} \), the limit virtual address is \( \text{VMEM}_1\_\text{LIMIT} \), and the size is \( \text{VMEM}_1\_\text{SIZE} \). By the definition of the RCS 421 hardware, the two regions are adjacent, such that \( \text{VMEM}_1\_\text{BASE} \) equals \( \text{VMEM}_0\_\text{LIMIT} \). The overall beginning virtual address of virtual memory is \( \text{VMEM}\_\text{BASE} \) (which equals \( \text{VMEM}_0\_\text{BASE} \)), and the overall limit virtual address of virtual memory is \( \text{VMEM}\_\text{LIMIT} \) (which equals \( \text{VMEM}_1\_\text{LIMIT} \)).
The state of the kernel in Region 1 consists of two parts: the kernel’s text, and the kernel’s global variables (its data, bss, and heap). 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. All of Region 0 (which includes the kernel stack) are switched on a context switch, to map the context of the processes being switched. The RCS 421 hardware defines the separation between the two regions in order to simplify context switching: on a context switch, your kernel should switch to a new Region 0 but should leave Region 1 unchanged.

These regions are not “segments” (e.g., as defined in the class textbook in Section 8.4). Actual “segments” have an associated base address and size that are managed (and changeable) by the operating system, whereas the regions in the hardware in this project have locations and sizes that are fixed in the virtual address space by the RCS 421 MMU hardware.

2.2.4 Page Tables

The RCS 421 MMU hardware automatically translates each reference (use) of any virtual memory address into the corresponding physical memory addresses. This translation happens automatically in the hardware, without intervention from the kernel. The kernel does, however, control the mapping that specifies this translation, by building and initializing the contents of the page table. This division of labor makes sense because memory references occur very frequently as the CPU executes instructions, whereas the virtual-to-physical address mapping (the page table) changes relatively infrequently and in accord with the process abstraction and memory protection policy implemented by the kernel.

The RCS 421 hardware uses a direct, single-level page table to define the mapping from virtual to physical addresses. Such a page table is an array of page table entries, laid out contiguously in physical memory. Each page table entry (PTE) contains information relevant to the mapping of a single virtual page to a corresponding physical page.

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

- **REG_PTR0**: Contains the physical memory address of the beginning of the page table for Region 0 of virtual memory.
- **REG_PTR1**: Contains the physical memory address of the beginning of the page table for Region 1 of virtual memory.

The MMU hardware never writes to these registers. The kernel can change the virtual-to-physical address mappings by changing either both of these registers (e.g., during a context switch) and may change any individual page table entry by modifying that entry in the corresponding page table in memory. **Remember that the value written to REG_PTR0 and REG_PTR1 must be the physical address of the corresponding page table.** For your operating system kernel software to be able to access a page table, you will need to map it to some virtual address, but you must tell the hardware the physical address of the page table. Also, the entire page table must be contiguous in physical memory, beginning at the physical address given in this register.

And remember, the hardware, in looking for a page table entry, will always look only at the page table in physical memory that begins at the physical address where REG_PTR0 or REG_PTR1 points (for a virtual address in Region 0 or Region 1, respectively), regardless of what virtual address your kernel (or you) thinks the relevant page table is accessible at.

When a program, or the kernel itself, makes a memory reference to any address, the MMU hardware 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 (VPN) within the region and contain
page table entries (PTEs) that specify the corresponding page frame number (among other things) of each page of virtual memory in that region.

For example, suppose that \texttt{PAGESIZE} is defined to be \texttt{0x1000} (this is the actual value defined in \texttt{comp421/hardware.h}). If the CPU references virtual address \texttt{0x2468}, this virtual address is at offset \texttt{0x468} within virtual page number \texttt{0x2} within Region 0. The MMU hardware in this case looks at the page table entry at subscript \texttt{0x2} in the page table pointed to by the physical address currently in register \texttt{REG_PTR0}.

Suppose also that \texttt{VMEM_1_BASE} is defined to be \texttt{0x200000} (this is the actual value defined in \texttt{comp421/hardware.h}). If the CPU references virtual address \texttt{0x202468}, this virtual address is at offset \texttt{0x468} within virtual page number \texttt{0x2} within Region 1. The MMU hardware in this case looks at the page table entry at subscript \texttt{0x2} in the page table pointed to by the physical address currently in register \texttt{REG_PTR1}. Notice that the subscript used by the hardware within the relevant page table is the virtual page number \textit{within the region} in which the virtual address is located, so the subscript used by the hardware in this example is \texttt{0x2}, \textit{not} \texttt{0x202}. (You can also consider, in this example, the virtual address to be in virtual page number \texttt{0x202 \textit{of the entire address space}}, but the hardware always uses the virtual page number \textit{within the relevant Region} as the subscript within the relevant page table.)

This lookup to translate a virtual page number into its currently mapped corresponding physical page frame number is carried out automatically, wholly in hardware. 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 dictated by the hardware. A page table entry is 32-bits wide, and contains the following defined fields:

- \texttt{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. All other bits in a PTE are ignored if the \texttt{valid} bit is off.

- \texttt{kprot} (3 bits): These three bits define the memory protection applied by the hardware to this virtual memory page whenever operating in kernel mode. These three protection bits are interpreted independently as follows:
  - \texttt{PROT_READ}: Memory within the page may be read.
  - \texttt{PROT_WRITE}: Memory within the page may be written.
  - \texttt{PROT_EXEC}: Memory within the page may be executed as machine instructions.

Execution of instructions by the CPU requires that their virtual memory pages be mapped with both \texttt{PROT_READ} and \texttt{PROT_EXEC} protection set.
• uprot (3 bits): These three bits define the memory protection applied by the hardware to this virtual memory page whenever operating in user mode. These three protection bits are interpreted independently, in the same way as for the kernel mode protection bits (kprot) defined above.

• pfn (20 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 page table entry format is illustrated in Figure 2 and is defined in the file comp421/hardware.h as a C data structure as a struct pte. This C structure has the same memory layout as an RCS 421 hardware page table entry and can be used in your operating system kernel.

2.2.5 Translation Lookaside Buffer (TLB)

The RCS 421 contains a translation lookaside buffer, or TLB, to speed up virtual-to-physical address translation. The TLB caches page table entries so that subsequent references to the same virtual page do not have to retrieve the corresponding page table entry anew from physical memory.

Page table entries are loaded automatically by the hardware, as needed, into the TLB 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 at times 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 (the value of the PTE before it was changed), since this entry may have been previously used and thus loaded into the TLB by the hardware. Also, when performing a context switch from one process to another, the operating system must flush all entries from the TLB corresponding to the virtual address space of the outgoing (old) process, since outgoing and incoming process use different page tables.

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

• TLB_FLUSH_ALL: Flush all entries in the entire TLB.
• TLB_FLUSH_0: Flush all entries in the TLB that correspond to virtual addresses in Region 0.
• TLB_FLUSH_1: Flush all entries in the TLB that correspond to virtual addresses in Region 1.
• Otherwise, the value written into the REG_TLB_FLUSH register is interpreted by the hardware as a virtual address. If an entry exists in the TLB corresponding to the virtual memory page into which this address points, flush this single entry from the TLB; if no such entry exists in the TLB, this operation has no effect.

Symbolic constants for the TLB_FLUSH_ALL, TLB_FLUSH_0, and TLB_FLUSH_1 values are defined in comp421/hardware.h.

2.2.6 Enabling Virtual Memory

When the machine is booted, a boot ROM firmware loads the kernel into physical memory and begins executing it. Specifically, the kernel is loaded into contiguous physical memory starting at a physical addresses equal to the number VMEM_1_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 RCS 421 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:

\[
\text{WriteRegister}(\text{REG\_VM\_ENABLE}, 1)
\]

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

Before enabling virtual memory as described above, your operating system kernel must properly initialize the page table for Region 1, which maps the kernel's own memory. The details on this process are given in Section 3.4.3.

2.3 Hardware Devices

The RCS 421 computer system can be configured with a variety of I/O and other hardware devices. For purposes of this project, only two different types of devices will be available: a number of terminals and a hardware clock.

2.3.1 Terminals

The system is equipped with \texttt{NUM\_TERMINALS} terminals, numbered 0 through \texttt{NUM\_TERMINALS} - 1, for use by user processes. By convention, the first 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 any difference in behavior. Like in the first project, the terminals are line-oriented, but here, this behavior is implemented for you; imagine that this line-oriented behavior is implemented in the RCS 421 hardware (although this is not entirely realistic, it does simplify the project).

When reading from a terminal, an interrupt is generated by the hardware only after a complete line has been typed at the terminal. For writing to a terminal, a buffer of output characters (which may actually be only part of a line or may be several lines of output) is given to the hardware terminal controller, and an interrupt is not generated until that entire buffer of output has been sent to the terminal. The constant \texttt{TERMINAL\_MAX\_LINE}, defined in \texttt{comp421/hardware.h}, defines the maximum line length (maximum buffer size) supported for either input or output on the terminals.

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

- \texttt{void TtyTransmit(int tty_id, void *buf, int len)}

The \texttt{TtyTransmit} operation begins the transmission of \texttt{len} characters from memory, starting at \texttt{virtual} address \texttt{buf}, to terminal \texttt{tty_id}. The address \texttt{buf} must be in the kernel's virtual memory, (i.e., it must be in virtual memory Region 1), since otherwise, this memory could become unavailable to the hardware terminal controller after a context switch. When the data has been completely written out, the terminal controller will generate a \texttt{TRAP\_TTY\_TRANSMIT} interrupt. When the \texttt{TRAP\_TTY\_TRANSMIT} interrupt handler in the operating system kernel is called, the terminal number of the terminal generating the interrupt will be made available to the handler. Until receiving a \texttt{TRAP\_TTY\_TRANSMIT} interrupt for this terminal after executing a \texttt{TtyTransmit} on it, no new \texttt{TtyTransmit} may be executed for this terminal.
int TtyReceive(int tty_id, void *buf, int len)

When the user completes an input line on a terminal by typing either newline (’\n’) or return (’\r’), the RCS 421 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. In the interrupt handler, the kernel should execute a TtyReceive operation for this terminal, in order to retrieve the new input line from the hardware. The new input line is copied from the hardware for terminal tty_id into the kernel buffer at virtual address buf, for maximum length to copy of len bytes. The value of len must be equal to TERMINAL_MAX_LINE bytes, since the kernel cannot otherwise know the length of the new input line before calling TtyReceive. The buffer must be in the kernel’s virtual memory (i.e., it must be entirely within virtual memory Region 1). After each TRAP_TTY_RECEIVE interrupt, the kernel must do a TtyReceive and save the new input line in a buffer inside the kernel, e.g., until a user process requests the next line from the terminal by executing a kernel call to read from this device.

The actual length of the new 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, since the blank line is terminated by a newline character. When an end of file character (control-D) is typed, TtyReceive returns 0 for this line. 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 by TtyReceive 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.

In reality (on real hardware), the buff address passed to TtyTransmit or TtyReceive would be interpreted by the hardware as a physical address. However, to simplify the project, on the RCS 421, this address is interpreted as a virtual address. Each terminal in the project is simulated using a separate X window, similar to the terminal behavior in the first project.

The operation of interrupts on the RCS 421 computer system is explained in more detail in Section 2.4.

2.3.2 The Hardware Clock

The RCS 421 computer system is equipped with a hardware clock that generates periodic interrupts to the operating system kernel. When a clock interrupt occurs, the hardware generates a TRAP_CLOCK interrupt.

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

The operation of interrupts on the RCS 421 computer system is explained in more detail in Section 2.4.

2.4 Interrupts, Exceptions, and Traps

2.4.1 Interrupt Vector Table and Types

Interrupts, exceptions, and traps are all handled in a similar way by the hardware, switching into kernel mode and calling a handler procedure in the kernel by indexing into the interrupt vector table based on the type of trap. The following types of interrupts, exceptions, and traps are defined by the RCS 421 hardware:

- TRAP KERNEL: This type of 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 type of interrupt results from the machine’s hardware clock, which generates periodic clock interrupts. The period between clock interrupts is exaggerated in the simulated hardware in order to avoid using up too much real CPU time on the shared CLEAR hardware systems.

- **TRAP_ILLEGAL**: This type of exception results from the execution of an illegal instruction by the currently executing user process. An illegal instruction can be, for example, an undefined machine language opcode or an illegal addressing mode.

- **TRAP_MEMORY**: This type of exception results from an attempt to access memory at some virtual address, when the access is disallowed by the hardware. 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 current page protection specified in the corresponding page table entry.

- **TRAP_MATH**: This type of 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 type of interrupt is generated by the terminal device controller hardware when a line of input is completed from one of the terminals attached to the system.

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

The interrupt vector table is stored in memory as an array of pointers to functions, each of which handles the corresponding type of interrupt, exception, or trap. The name of each type listed above is defined in `comp421/hardware.h` as a symbolic constant. This number is used by the hardware to index into the interrupt vector table to find the pointer to the handler function for this trap.

The privileged machine register `REG_VECTOR_BASE` is used by the hardware as the base physical address in memory where the interrupt vector table is stored. After initializing the table in memory, the kernel must write the physical address of the table to the `REG_VECTOR_BASE` register. The entire interrupt vector table must be contiguous in physical memory, beginning at the physical address given in the `REG_VECTOR_BASE` register.

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. Any entries in the interrupt vector table other than those defined above must be initialized to NULL.

To simplify the programming of the 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 semaphores or monitors 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 current interrupt, exception, or trap. Although this is somewhat unrealistic, it is possible to configure real hardware and the kernel to behave this way, and it greatly simplifies the programming of the kernel for the project.

### 2.4.2 Hardware-Defined Exception Information Structure

When an interrupt, exception, or trap occurs, the RCS 421 hardware pushes an `ExceptionInfo` structure onto the kernel stack to save the current CPU state, and passes a pointer to this structure on the stack to the
corresponding handler function. Each interrupt, exception, or trap handler function thus should be defined as follows:

```c
void name(ExceptionInfo *)
```

When the handler function returns, the hardware restores the CPU state from the values then in this `ExceptionInfo` structure, popping it off of the kernel stack.

The format of the `ExceptionInfo` structure is defined in the file `comp421/hardware.h`. You can include this file in your source file by using

```c
#include <comp421/hardware.h>
```

_Do not copy this file into your own directory._ The following fields are defined by the RCS 421 hardware within an `ExceptionInfo` structure:

- **int vector**: An integer indicating the type of interrupt, exception, or trap that has occurred, such as `TRAP_ILLEGAL`, etc. This is the index (subscript) into the interrupt vector table used to call this handler function.

- **int code**: An integer value giving more information on the particular interrupt, exception, or trap. The meaning of the integer `code` value varies depending on the type of interrupt, exception, or trap. The defined values of `code` are summarized in Table 2.

- **void *addr**: This field is only meaningful for a `TRAP_MEMORY` exception. It contains the memory address whose reference caused the exception.

- **unsigned long psr**: The value of the CPU’s processor status register (PSR) at the time of the interrupt, exception, or trap. The `PSR_MODE` bit in the PSR defines the current execution mode of the CPU. Specifically, if `((psr & PSR_MODE))` is true, the CPU was executing in kernel mode when the interrupt, exception, or trap occurred; if this bit is set in the `psr` value, the CPU will execute in kernel mode after returning from the handler.

- **void *pc**: The hardware program counter register value at the time of the interrupt, exception, or trap. The value of `pc` will be restored to the hardware program counter register on return from the handler.

- **void *sp**: The hardware stack pointer register value at the time of the interrupt, exception, or trap. The value of `sp` will be restored to the hardware stack pointer register on return from the handler.

---

**Table 2  Summary of the Meaning of Code Values in the `ExceptionInfo` Structure**

<table>
<thead>
<tr>
<th>Vector</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAP_KERNEL</td>
<td>Kernel call number</td>
</tr>
<tr>
<td>TRAP_CLOCK</td>
<td>Undefined</td>
</tr>
<tr>
<td>TRAP_ILLEGAL</td>
<td>Type of illegal instruction</td>
</tr>
<tr>
<td>TRAP_MEMORY</td>
<td>Type of disallowed memory access</td>
</tr>
<tr>
<td>TRAP_MATH</td>
<td>Type of math error</td>
</tr>
<tr>
<td>TRAP_TTY_TRANSMIT</td>
<td>Terminal number causing interrupt</td>
</tr>
<tr>
<td>TRAP_TTY_RECEIVE</td>
<td>Terminal number causing interrupt</td>
</tr>
</tbody>
</table>
Table 3  Meaning of Individual Code Values for TRAP_ILLEGAL

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAP_ILLEGAL_ILLOPC</td>
<td>Illegal opcode</td>
</tr>
<tr>
<td>TRAP_ILLEGAL_ILLOPN</td>
<td>Illegal operand</td>
</tr>
<tr>
<td>TRAP_ILLEGAL_ILLADR</td>
<td>Illegal addressing mode</td>
</tr>
<tr>
<td>TRAP_ILLEGAL_ILLTRP</td>
<td>Illegal software trap</td>
</tr>
<tr>
<td>TRAP_ILLEGAL_PRVOPC</td>
<td>Privileged opcode</td>
</tr>
<tr>
<td>TRAP_ILLEGAL_PRVREG</td>
<td>Privileged register</td>
</tr>
<tr>
<td>TRAP_ILLEGAL_COPROC</td>
<td>Coprocessor error</td>
</tr>
<tr>
<td>TRAP_ILLEGAL_BADSTK</td>
<td>Bad stack</td>
</tr>
<tr>
<td>TRAP_ILLEGAL_KERNELI</td>
<td>Linux kernel sent SIGILL</td>
</tr>
<tr>
<td>TRAP_ILLEGAL_USERIB</td>
<td>Received SIGILL or SIGBUS from user</td>
</tr>
<tr>
<td>TRAP_ILLEGAL_ADRALN</td>
<td>Invalid address alignment</td>
</tr>
<tr>
<td>TRAP_ILLEGAL_ADERR</td>
<td>Non-existent physical address</td>
</tr>
<tr>
<td>TRAP_ILLEGAL_OBJERR</td>
<td>Object-specific HW error</td>
</tr>
<tr>
<td>TRAP_ILLEGAL_KERNELB</td>
<td>Linux kernel sent SIGBUS</td>
</tr>
</tbody>
</table>

- **unsigned long regs[NUM_REGS]**: The contents of eight general purpose hardware registers in the CPU at the time of the interrupt, exception, or trap (the value NUM_REGS is defined to be 8). For example, by convention, for a kernel call, these registers are loaded with the arguments for the call before executing the trap instruction, and thus for a TRAP_KERNEL, the arguments to the kernel call can be accessed using this regs array. More information on the use of the regs array in handling a kernel call is available in Section 3.2. The regs values will be restored to the corresponding hardware registers on return from the handler.

The values of the privileged machine registers (the REG_* registers, defined above in Section 2.1.2) are not included in the exception information (ExceptionInfo) 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.

If you are interested, more details on the meaning of the individual code values for TRAP_ILLEGAL, TRAP_MEMORY, and TRAP_MATH are listed in Tables 3, 4, and 5, respectively, and are defined in the include file comp421/hardware.h. In particular, these three tables show the #define name for each code value as defined in hardware.h.

These individual code values are defined for information only. **You do not need (and you should not use) these code values as any part of the operation of your kernel**, other than perhaps using them as part of printing an error message when one of these types of exceptions occur; I get these values directly from Linux and report them in the code value so you can print more informative messages if you like.

### 2.5 Additional CPU Machine Instructions

In addition to the hardware features of the RCS 421 computer system already described, the CPU provides two additional instructions for special purposes:

- **void Pause(void)**

  The Pause instruction 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
Table 4  Meaning of Individual Code Values for TRAP_MEMORY

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAP_MEMORY_MAPERR</td>
<td>No mapping at addr</td>
</tr>
<tr>
<td>TRAP_MEMORY_ACCERR</td>
<td>Protection violation at addr</td>
</tr>
<tr>
<td>TRAP_MEMORY_KERNEL</td>
<td>Linux kernel sent SIGSEGV at addr</td>
</tr>
<tr>
<td>TRAP_MEMORY_USER</td>
<td>Received SIGSEGV from user</td>
</tr>
</tbody>
</table>

Table 5  Meaning of Individual Code Values for TRAP_MATH

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAP_MATH_INTDIV</td>
<td>Integer divide by zero</td>
</tr>
<tr>
<td>TRAP_MATH_INTOVF</td>
<td>Integer overflow</td>
</tr>
<tr>
<td>TRAP_MATH_FLTDIV</td>
<td>Floating divide by zero</td>
</tr>
<tr>
<td>TRAP_MATH_FLTOVF</td>
<td>Floating overflow</td>
</tr>
<tr>
<td>TRAP_MATH_FLTUND</td>
<td>Floating underflow</td>
</tr>
<tr>
<td>TRAP_MATH_FLTRES</td>
<td>Floating inexact result</td>
</tr>
<tr>
<td>TRAP_MATH_FLTINV</td>
<td>Invalid floating operation</td>
</tr>
<tr>
<td>TRAP_MATH_FLTSUB</td>
<td>FP subscript out of range</td>
</tr>
<tr>
<td>TRAP_MATH_KERNEL</td>
<td>Linux kernel sent SIGFPE</td>
</tr>
<tr>
<td>TRAP_MATH_USER</td>
<td>Received SIGFPE from user</td>
</tr>
</tbody>
</table>

real operating system on real hardware, this is alright since there is nothing else for the CPU to do. Operating Systems generally provide an idle process that is executed in this situation, and which is typically just an empty infinite loop. However, in order to be nice to other users on the real CLEAR Linux machines you will be using in the project, your idle process should not just be an empty loop in this way. Instead, the idle process in your Yalnix kernel should be a loop that executes the Pause instruction in each loop iteration; the idle process should never exit.

- void Halt(void)

  The Halt instruction completely stops the CPU (ends the execution of the simulated RCS 421 computer system). Specifically, 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 CLEAR Linux terminal).

3 Yalnix Operating System Details

This section describes the Yalnix operating system, a reasonably simple, mythical operating system that can be implemented on many types of computer systems including the RCS 421.

This section is written partially in the style of a real operating system architecture manual. Everything in this section is something that a real designer, implementer, or user of the operating system would need to know about the operating system. Certain features of the RCS 421 computer system on which your kernel
will execute are mentioned here only because you will need them in the project; the Yalnix operating system could also be implemented on other hardware platforms.

3.1 Yalnix Kernel Calls

User processes request services from the operating system kernel by executing a “kernel call” (sometimes called a “system call” or “syscall”). A kernel call, from the point of view of a user process, is simply a procedure call to a small piece of code (written in assembly language) that executes a “trap” instruction to enter kernel mode in the operating system to request some service of the kernel.

The following kernel calls, shown here as the procedure call interface used by a user process, are defined in Yalnix:

- **int Fork(void)**

  The `Fork` kernel call is the only way an existing process can create a new process in Yalnix (there is no inherent limit on the maximum number of processes that can exist at once). 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 `Fork` kernel call as if each 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 (nonzero) process ID of the new (child) process, while the value returned in the child is 0; as noted in Section 6.1, the process ID for each (real) process should be greater than 0 (you should use the process ID value of 0 for your idle process). When returning from a successful `Fork`, you can return first either as the new child process or as the parent process (think about why you might want to do one or the other). If, for any reason, the new process cannot be created, `Fork` 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

  ```c
  main(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, this call returns `ERROR` to the calling program as the result of `Exec` and does not run the new program.

- **void Exit(int status)**

  `Exit` is the normal means of terminating a process. The current process is terminated, and the integer value of `status` is saved for later collection by the parent process if the parent calls to `Wait`. However, when a process exits or is terminated by the kernel, if that process has children, each of these children should continue to run normally, but it will no longer have a parent and is thus
then referred to as an “orphan” process. When such an orphan process later exits, its exit status is not saved or reported to its parent process since it no longer has a parent process; unlike in Unix, an orphan process is not “inherited” by any other process.

When a process exits or is terminated by the kernel, all resources used by the calling process are freed, except for the saved status information (if the process is not an orphan). The Exit kernel call can never return.

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

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

- **int GetPid(void)**

  Returns the process ID of the calling process.

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

  Brk sets the operating system’s idea of the lowest location not used by the program (called the “break”) to addr. Since memory is allocated to a process in units of whole pages, the actual break is in effect rounded up to a multiple of PAGESIZE, although the specified addr value need not itself be a multiple of PAGESIZE. This call has the effect of allocating or deallocating enough memory to cover only up to the specified address. Locations not less than the break (and below the allocated memory for the process’s stack) are not in the address space of the process and will thus cause an exception if accessed. The value 0 is returned by Brk on success. If any error is encountered (for example, if not enough memory is available or if the address addr is invalid), the value ERROR is returned.

- **int Delay(int clock_ticks)**

  The calling process is blocked until clock_ticks clock interrupts have occurred after the call. Upon completion of the delay, the value 0 is returned. If clock_ticks is 0, return is immediate. If clock_ticks is less than 0, ERROR is returned instead.

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

  Read the next line of input (or a portion of it) from terminal tty_id, copying the bytes of input into the buffer referenced by buf. The maximum length of the line to be returned is given by len. A value of 0 for len is not in itself an error, as this simply means to read “nothing” from the terminal. The line returned in the buffer is not null-terminated. Any call to TtyRead returns part of (or all of) a line of input that came from a single TtyReceive. Any call to verb—TtyRead— never returns more than a single line of input that came from a single TtyReceive.

  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’s buffer, and the remaining bytes of the line are saved by the kernel for the next TtyRead (by this or another process) for this terminal. If the length of the next available input line is shorter than \( \text{len} \) bytes, only as many bytes are copied to the calling process’s buffer 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 \text{ERROR} is returned.

- \( \text{int TtyWrite(int tty_id, void *buf, int len)} \)

  Write the contents of the buffer referenced by \( \text{buf} \) to the terminal \( \text{tty_id} \). The length of the buffer in bytes is given by \( \text{len} \); the maximum value of \( \text{len} \) is limited to \text{TERMINAL_MAX_LINE}. A value of 0 for \( \text{len} \) is not in itself an error, as this simply means to write “nothing” to the terminal.

  The calling process is blocked until all characters from the buffer have been written on the terminal. On success, the number of bytes written (which should be \( \text{len} \)) is returned; in case of any error, the value \text{ERROR} is returned.

Function prototypes for all of the Yalnix kernel calls defined above are defined in the include file comp421/yalnix.h, in the directory /clear/courses/comp421/pub/include/comp421. You can include the file comp421/yalnix.h in your source file by using

\[
\text{#include <comp421/yalnix.h>}
\]

Some additional Yalnix kernel calls are also defined in comp421/yalnix.h, although you need not implement them for this project.

\textit{Do not copy this file into your own directory.}

As noted above, a kernel call is requested by a user process by calling one of the procedures defined above, but that procedure just executes a “trap” instruction to enter kernel mode and request that service of the operating system kernel. As such, a kernel call begins execution in your kernel as a call to your TRAP_KERNEL handler. The code value included in the ExceptionInfo for this call gives the kernel call number indicating which service is being requested (e.g., \text{YALNIX_FORK}, \text{YALNIX_EXEC}, etc.). The procedures defined above are used by user processes to call your kernel; how you implement the behavior of each kernel call inside your kernel is up to you. In particular, you are not required to (and generally should not) use the procedure names above for the implementation of each kernel call inside your kernel.

For example, the definition of the Delay kernel call above takes a single integer argument, but for the implementation of this kernel call inside your kernel, you can name the procedure that you write to implement this call any name you like, and you can pass to that procedure inside your kernel any arguments you like, with any return value you like. If, however, you use the name Delay for the implementation of this inside your kernel, the C compiler will insist that you conform to the function prototype above, which is probably not what you want to do in the implementation inside your kernel.

\textit{Unless otherwise noted above, all Yalnix kernel calls return the value \text{ERROR} in case of any error in processing that kernel call.} The value \text{ERROR} is defined in the include file comp421/yalnix.h. You should be careful in your kernel to actually check for all possible error conditions for each kernel call.

The file comp421/yalnix.h also defines a constant “kernel call number” each type of kernel call, used to indicate to the kernel the specific kernel call being invoked; this type value is visible to the kernel in the code field of the ExceptionInfo structure passed by the hardware for the trap, as described in Section 3.2.

Your kernel must verify all arguments passed to a kernel call, and should return \text{ERROR} if any arguments are invalid. In particular, this includes checking that a pointer value passed to the kernel call is valid before you use the pointer in your kernel. Specifically, your kernel must verify that the pointer points into the process’s memory and should look in the page table to make sure that the entire area (such as defined by a
pointer and a specified length) is readable and/or writable (as appropriate) before the kernel actually tries to read or write there. For null-terminated C-style character strings (C strings like this are passed to `Exec`), you will need to, in effect, check the pointer to each byte as you go (although this effect can be implemented more efficiently than actually checking for each individual byte). For simplicity, you should 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, in effect, check access to each byte, checking each until it found the ’\0’ at the end. Insert calls to these two routines as needed at the top of each kernel call to verify the pointer arguments before you use them.

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

### 3.2 Interrupt, Exception, and Trap Handling

As described above in Section 2.4, each type of interrupt, exception, or trap that can be generated by the hardware has a corresponding type value used by the hardware to index into the interrupt vector table in order to find the address of the handler function to call. The operating system kernel must create the interrupt vector table and initialize each corresponding entry in the table with a pointer to the relevant handler function within the kernel. The address of the interrupt vector table must also be written into the `REG_VECTOR_BASE` register by the kernel at boot time.

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

The range of possible index (subscript) values in the interrupt vector table is defined by the hardware and ranges from 0 to `TRAP_VECTOR_SIZE–1`. The interrupt vector table must be exactly this size (this number of entries). Each entry not needed must be initialized to `NULL`.

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

- **TRAP_KERNEL**: Execute the requested kernel call, as indicated by the kernel call number in the `code` field of the `ExceptionInfo` passed by the hardware by reference to this trap handler function. The arguments passed by the user process for this kernel call are also available to the kernel beginning in `regs[1] in the ExceptionInfo` passed by reference to the kernel’s `TRAP_KERNEL` handler function. The return value from the kernel call should be returned to the user process in the `regs[0]` field of the `ExceptionInfo`. Note that for the `Exit` kernel call, there is no return value, since `Exit` causes the calling process to be terminated by your kernel.

- **TRAP_CLOCK**: Your Yalnix kernel should implement round-robin process scheduling with a time quantum per process of 2 clock ticks. After the current process has been running as the current process continuously for at least 2 clock ticks, if there are other runnable processes on the ready queue, perform a context switch to the next runnable process.

- **TRAP_ILLEGAL**: Terminate the currently running Yalnix user process and print a message giving the process ID of the process and an explanation of the problem; and continue running other processes.

- **TRAP_MEMORY**: The kernel must determine if this exception represents an implicit request by the current process to enlarge the amount of memory allocated to the process’s stack, as described in more detail in Section 3.4.4. If so, the kernel enlarges the process’s stack to “cover” the address that was being referenced that caused the exception (the `addr` field in the `ExceptionInfo`) and then
returns from the exception, allowing the process to continue execution with the larger stack. Otherwise, terminate the currently running Yalnix user process and print a message giving the process ID of the process and an explanation of the problem; and continue running other processes.

- **TRAP_MATH**: Terminate the currently running Yalnix user process and print a message giving the process ID of the process and an explanation of the problem; and 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 **ExceptionInfo** passed by reference by the hardware to this interrupt handler function. The kernel should read the input from the terminal using a **TtyReceive** hardware operation and if necessary should buffer the input line for a subsequent **TtyRead** kernel call by some user process.

- **TRAP_TTY_TRANSMIT**: This interrupt signifies that the previous **TtyTransmit** hardware operation on some terminal has completed. The specific terminal is indicated by the **code** field in the **ExceptionInfo** passed by reference by the hardware to this interrupt handler function. The kernel should complete the **TtyWrite** kernel call and unblock the blocked process that started this **TtyWrite** kernel call that started this output. If other **TtyWrite** calls are pending for this terminal, also start the next one using **TtyTransmit**.

As described above, a **TRAP_KERNEL** trap occurs when a Yalnix user process requests a kernel call for some function provided by the kernel. A user process 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, a library of assembly language routines is provided that performs this trap for the user process. This library provides a standard C procedure call interface for kernel calls, as indicated in the description of each kernel call 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.

Upon entry to your kernel’s **TRAP_KERNEL** handler function, the **code** field of the **ExceptionInfo** structure indicates which kernel call is being invoked (as defined with symbolic constants in **comp421/yalnix.h**). You can include the file **comp421/yalnix.h** in your source file by using

```
#include <comp421/yalnix.h>
```

*Do not copy this file into your own directory.*

The arguments to this call, supplied to the library procedure call interface in C, are available in the **regs** array in the **ExceptionInfo** structure passed by reference to your **TRAP_KERNEL** handler. Each argument passed to the library procedure is available in a separate **regs** entry, in the order passed to the procedure, beginning with **regs[1]**. In other words, the first argument to the kernel call is in **regs[1]**, the second argument to the kernel call is in **regs[2]**, and so on.

Each kernel call returns a single integer value, indicated in Section 3.1, which becomes the return value from the C library procedure call interface for the kernel call. When returning from a **TRAP_KERNEL** trap, the value to be returned from this interface should be placed by the kernel in **regs[0]** in the **ExceptionInfo**.

When the kernel terminates a process (e.g., after a **TRAP_ILLEGAL**, not as a result of **Exit()**), your kernel should print a message, giving the process ID of the process and some explanation of the problem. The exit status reported to the parent process of the terminated process when the parent calls the **Wait** kernel call (as described in Section 3.1) should be the value **ERROR**, as if the child had terminated by calling **Exit(ERROR)** itself. You can print the message from your kernel with **printf** or **fprintf**, in which case it will print out on the Linux window from which you started your kernel, or (better, but not required) you can use the hardware-provided **TtyTransmit** to print the message on the Yalnix console.
3.3 Kernel Booting Entry Point

Your kernel is not a complete program — it is more like a library of functions that get called on interrupts, exceptions, and traps. As such, your kernel does not have a main procedure like typical C programs do.

Instead, when booted, your kernel begins executing at a procedure named KernelStart, which you must write in your kernel. The procedure named KernelStart is automatically called by the bootstrap firmware in the computer, as follows:

```c
void KernelStart(ExceptionInfo *info,
                 unsigned int pmem_size, void *orig_brk, char **cmd_args)
```

The procedure arguments passed to KernelStart are built by the bootstrap firmware during initialization of the machine. The KernelStart procedure should initialize your operating system kernel and then return. The arguments to KernelStart are defined as follows:

- The info argument is a pointer to an initial ExceptionInfo structure (which is defined in Section 2.4.2). The KernelStart procedure is called in the same way as the interrupt, exception, and trap handlers, as described in Sections 2.4 and 3.2, except that KernelStart is passed additional arguments (described below). As when an interrupt, exception, or trap handler is called by the hardware, an ExceptionInfo is pushed onto the kernel stack, and then a pointer to that ExceptionInfo structure is passed as an argument to KernelStart. This initial ExceptionInfo records the state of the machine at boot time, and any changes made in the values in this ExceptionInfo control how and where the machine will execute when KernelStart returns. The KernelStart procedure should initialize the operating system kernel and then return, starting the first process executing. More details on initializing the operating system kernel and in the use of the info argument passed to KernelStart are provided in Section 3.6.

- The pmem_size argument gives the total size of the physical memory of the machine you are running on, as determined dynamically from the hardware by the bootstrap firmware. This total size of physical memory is given in units of bytes.

- The orig_brk argument gives the initial value of the kernel’s “break.” That is, this address is the first address that is not part of the kernel’s initial heap.

- 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 Linux terminal) to start the machine and thus the kernel. The cmd_args vector is terminated by a NULL pointer.

You must write a procedure named KernelStart as part of your kernel. Since this is the place at which your kernel first starts executing at boot time, it is recommended that you write and test a simple version of a KernelStart procedure as the first step in building your kernel.

3.4 Memory Management

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

Memory management for user processes is reasonably straightforward. To enlarge the program area (the heap) of the user process, the process calls the Brk kernel call. In general, processes don’t do this explicitly themselves, however, since, for example, the standard C library malloc function will call Brk automatically if it needs more memory. To instead enlarge the stack area of the user process, the process
Yalnix Virtual Address Space Layout

Each process has its own kernel stack (constant size)

Each process has its own Region 0

Region 0

Region 1

Kernel stack

User stack

Trap Memory

Brk()

Heap

Global variables

User program

Only 1 kernel, shared by all processes

Figure 3 Yalnix Virtual Address Space Layout

simply tries to use the new stack pointer value, which causes a TRAP_MEMORY exception if not enough memory is currently allocated for the process’s stack. Details of the procedure for growing the stack of a user process are given in Section 3.4.4.

Memory management for the kernel itself is a bit more complicated. The kernel may also use malloc for dynamic memory allocation for its own data structures, but unlike malloc from a user program, the kernel can’t rely on someone else to actually allocate the memory for it. The kernel must allocate the physical memory into Region 1 for the kernel’s own use, as needed for malloc calls made by the kernel. Details of this procedure are given in Section 3.4.2. In addition, the kernel must initialize and enable virtual memory at boot time; details of this procedure are given in Section 3.4.3.

### 3.4.1 Virtual Address Space Layout

The Yalnix operating system uses the virtual memory management hardware provided by the RCS 421 computer system to implement a separate virtual address space for each process, with the kernel shared by all processes. The layout of the virtual address space implemented by Yalnix is shown in Figure 3.

The machine’s virtual address space is divided into two “Regions,” named Region 0 and Region 1. Region 0 is used by each user process, and Region 1 is used by the kernel and is shared by all processes. Each of the two regions is the same size, defined as VMEM_REGION_SIZE (which also equals VMEM_0_SIZE and VMEM_1_SIZE). Region 0 begins at virtual address VMEM_0_BASE (which is defined equal to 0) and ends at virtual address VMEM_0_LIMIT. Region 1 begins at virtual address VMEM_1_BASE (which is defined equal to VMEM_0_LIMIT) and ends at virtual address VMEM_1_LIMIT. The constant VMEM_BASE (which is defined equal to VMEM_0_BASE and is thus equal to 0) is the virtual address of the beginning of the entire virtual address space, and the constant VMEM_LIMIT (which is defined equal to VMEM_1_LIMIT) is the virtual address of the end of the entire virtual address space.
The limit of the program’s heap (the lowest address above the heap that is not part of the process’ address space) is called the process’s break and is set from Yalnix user processes by the Brk kernel call, which you will implement in your kernel.

The break of a user process is initialized (when a program begins running) to an address defined in the executable program file from which the process was loaded by Exec. Details on how to load a program for Exec are provided in the file load.template. This is a template for a function LoadProgram that can load a Yalnix program into memory. The file load.template is located in the directory /clear/courses/comp421/pub/templates. You should copy and modify this template as described in the load.template file itself. You can also modify any part of the code in (your copy of) this file to fit with the rest of your kernel, such as by changing or adding arguments passed to the LoadProgram function. After the Exec completes and a program begins execution, the process itself allocates more memory as needed by calling the kernel’s Brk kernel call.

The user stack for each process begins (at the top of the stack) at virtual address USER_STACK_LIMIT. The size of the user stack of a process will be automatically grown by your Yalnix kernel in response to a TRAP_MEMORY exception resulting from an access to a page between the current break and the current stack pointer that has not yet been allocated. This happens during normal execution as the program makes procedure calls that automatically expand the stack downwards. Details of this are given in Section 3.4.4 below.

Each process executing on Yalnix also has a separate kernel stack, located in that process’s Region 0. The kernel stack is fixed in size at KERNEL_STACK_PAGES pages (or KERNEL_STACK_SIZE bytes). The kernel stack begins at (at the top of the stack) at virtual address KERNEL_STACK_LIMIT and grows down (to lower virtual addresses) from there. The bottom address limit of the kernel stack is virtual address KERNEL_STACK_BASE. Each time the hardware calls a handler procedure in the kernel, the hardware pushes an ExceptionInfo structure onto the kernel stack, which includes the then-current value of the stack pointer register. Before pushing this information onto the stack, the hardware automatically changes the stack pointer to the value KERNEL_STACK_LIMIT, and the value of the stack pointer register saved in the ExceptionInfo is the original value of the stack pointer before the value of this register is changed to KERNEL_STACK_LIMIT.

The kernel stack is located at the top of Region 0, and there is a separate kernel stack for each process. By having a separate kernel stack for each process, it is possible for the kernel to block the current process while inside the kernel. When performing a context switch, the kernel can then switch to the kernel stack of the newly selected process (the process that is being context switched in), thus leaving the state of the old process’s kernel stack entirely unmodified (in that process’s separate Region 0) until that process is selected to run again next and is context switched in. All of that process’s local variables inside the kernel and all of the subroutine return addresses and such that represent the dynamic state of the process inside the kernel when it was last context switched out, are all still intact and again available when the process is next context switched in. The process can thus begin execution again after the context switch at exactly the point it was at inside the kernel when it last ran.

### 3.4.2 Kernel Memory Management

Unlike the user stack for each user process, the kernel stack for each process is fixed in size, as noted above in Section 3.4.1. Thus, after setting up the kernel stack for a process, the kernel need not explicitly manage the kernel stack for that process.

In contrast to the kernel stack, the kernel heap does require management by your operating system kernel. Each time some procedure such as malloc called by your kernel needs more pages added to the kernel’s memory, it calls a function named SetKernelBrk, defined as follows:

```c
int SetKernelBrk(void *addr)
```
In a user program, malloc calls the Brk kernel call to request the operating system kernel to add additional memory to the process’s address space. When the kernel calls malloc, however, the internal kernel function SetKernelBrk is called instead. (Internally, malloc keeps a list of free block of memory and carves up existing free blocks as necessary, and only asks for more total memory kernel when its existing available memory cannot satisfy the requested allocation.)

Before your kernel has enabled virtual memory, the entire physical memory of the machine is already available to your kernel. The job of SetKernelBrk in this case is very easy. You need only keep track of the location of the current break for the kernel. The initial location of the break is passed to your kernel’s KernelStart procedure as the orig_brk argument, as described in Section 3.3. Calls to SetKernelBrk, before virtual memory has been enabled, simply move this break location up (or down) to the new location specified as addr in the SetKernelBrk call. You will need to know the current location of your kernel’s break in order to correctly initialize page table entries for your kernel’s Region 1 before enabling virtual memory.

After you have enabled virtual memory, the job of the SetKernelBrk function that you will write becomes more complicated. After this point, your implementation of SetKernelBrk, when called with an argument of addr, must allocate physical memory page frames and map virtual pages as necessary to make addr the new kernel break. In your kernel, you should keep a flag to indicate if you have yet enabled virtual memory, so your implementation of SetKernelBrk can know which behavior (for before virtual memory has been enabled, or for after virtual memory has been enabled) it should perform for each call to SetKernelBrk.

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

Note that the malloc implementation that you use inside your kernel (and its helping function SetKernelBrk, which you must write) is completely separate from the malloc implementation that each user process uses. Thus, the kernel has its own break location and the list of free blocks of memory managed by malloc is completely separate from that in any process. Likewise, each process running on Ynalix has its own break and is linked with its own copy of the malloc library routines (if the process uses dynamic memory allocation), making its list of free blocks of memory also separate from each other process.

### 3.4.3 Initializing the Page Tables and Virtual Memory

As described in Section 2.2.6, when the machine is booted, the machine’s virtual memory subsystem is initially disabled and is only enabled once your operating system kernel writes a nonzero value into the REG_VM_ENABLE register. Before doing this, your kernel must initialize the page table and page table registers.

A consequence of this procedure for enabling virtual memory is that the kernel will have already been loaded into memory and executed for a while before virtual memory is enabled. During this execution, all addresses used by the kernel during its execution are treated by the hardware as physical addresses, but after enabling virtual memory, all addresses then used by the kernel during its execution are treated by the hardware 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 for the kernel so that all addresses 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 current physical address space layout, as illustrated in Figure 4. The hardware makes this possible by guaranteeing that \texttt{PMEM\_BASE}, the beginning address of physical memory, is the same value as \texttt{VMEM\_BASE}, the beginning address of virtual memory (which is thus also the beginning address of Region 0 of virtual memory). In building the initial page tables in the kernel before enabling virtual memory, the kernel should make sure, for every page of physical memory \texttt{pfn} that is then in use by your kernel, a page table entry should be built so that the new virtual page number \texttt{vpn} = \texttt{pfn} maps to physical page number \texttt{pfn}. That is, for all of these virtual pages, their new virtual page number is the same as their (existing) physical page number (page frame number).

To correctly build these initial page table entries, your kernel must know how many page table entries to build, and your kernel must know the correct protection settings to initialize in each of these page table entries. In particular, the page table entries for your kernel text should be set to “read” and “execute” protection for kernel mode, and the page table entries for your kernel data/bss/heap should be set to “read” and “write” protection for kernel mode; the user mode protection for both kinds of kernel page table entries should be “none” (no access).

The end of the kernel’s heap can be known by your kernel from the initial “break” address passed as \texttt{orig\_brk} to your \texttt{KernelStart} function and from any calls to your \texttt{SetKernelBrk} function that may have occurred so far in the execution of your kernel. Specifically, the end of your kernel’s heap begins...
at \texttt{orig\_brk} and may move up by subsequent calls to \texttt{SetKernelBrk}. The end of the kernel’s text is simpler and can be known by your kernel from the following address:

\begin{verbatim}
&\_etext
\end{verbatim}

The value of \&\_etext (that is, the \emph{address} of the symbol \_etext) is the lowest address that does not contain executable instructions (text) in your kernel. The value of \&\_etext will always be on a page boundary. The symbol \_etext is automatically placed in the symbol table of your kernel by the linker, although it is not really a variable. The \emph{address} of this symbol is what you want.

### 3.4.4 Growing a User Process’s Stack

When a process pushes onto its user mode stack, it simply decrements the current stack pointer value and attempts to write into memory at the location now pointed to by the stack pointer. Normally, this works with no problem, 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, growing into a new page, which will cause a \texttt{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 \texttt{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 \texttt{TRAP\_MEMORY} exception, if the virtual address being referenced that caused the exception (the \texttt{addr} value in the \texttt{ExceptionInfo}) is in Region 0, 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 terminate the currently running Yalnix user process, but should continue running any other user processes.

When growing the stack, you should leave at least one page unmapped (with the valid bit in its page table set to 0) between the heap and user stack in Region 0, so the stack will not silently grow into and overlap with the heap without triggering a \texttt{TRAP\_MEMORY}. This unmapped page between the program area and the stack is known as a “red zone” into which the stack (and the break) 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 Context Switching

When some handler function in the kernel is called by the hardware due to an interrupt, exception, or trap, the hardware enters kernel mode and switches to the kernel stack. The current user process stack then contains the local variables and return addresses from each function called by the user process leading up to the point at which the interrupt, exception, or trap occurred. As the kernel’s handler function executes (and possibly calls other internal functions inside the kernel), the kernel stack then contains the local variables and return addresses for each function inside the kernel called by this process during the execution since entering the kernel, including the handler function itself. Both of these stacks (the user stack and the kernel stack) are located in virtual memory in Region 0. Also in Region 0 is the text, data, bss, and heap used by the executing user process.

A context switch from one process to another can only be performed while executing inside the kernel. To perform a context switch, the kernel switches from one Region 0 page table to another Region 0 page table (and also, at the same time, writes to the \texttt{REG\_TLB\_FLUSH} register to flush all the Region 0 entries.
from the TLB). This causes the kernel stack, as well as the user stack and the rest of the contents of Region 0 to switch. However, this switch occurs at some place deep inside your kernel, at whatever point in the code your kernel performed the necessary operations to accomplish this switch. Since your kernel would then be executing on the kernel stack, switching the kernel stack while using the kernel stack could create great confusion for the kernel’s own execution. For example, the return address and local variables for the current function would suddenly “disappear” when the Region 0 virtual memory is replaced with a different Region 0.

To simplify the operation of context switching and to avoid this confusion to the kernel’s own execution, we provide the function

```c
int ContextSwitch(SwitchFunc_t *func, SavedContext *ctxp,
                  void *p1, void *p2)
```

to help with the context switch process. This function is purely software (not a hardware function); we provide it for your use in this form simply because writing it yourself would be very messy, since it involves complicated low-level machine-specific details (and on our simulated hardware, Linux-specific details).

The type `SwitchFunc_t` (switch function type) is a messy C typedef of a special kind of function. `ContextSwitch` and the type `SwitchFunc_t` are defined in `comp421/hardware.h`, in the directory `/clear/courses/comp421/pub/include/comp421`. Do not copy this file into your own directory. The `ContextSwitch` function temporarily stops using the standard kernel stack and calls the function `func` using a temporary, different stack not located in Region 0.

For example, suppose you named your switch function `MySwitchFunc`. Then you would declare `MySwitchFunc` as:

```c
SavedContext *MySwitchFunc(SavedContext *ctxp, void *p1, void *p2)
```

Since `MySwitchFunc` is called while not using the normal kernel stack, `MySwitchFunc` can easily and safely do what it needs to do to switch Region 0 to complete the context switch.

The two “void *” arguments `p1` and `p2` passed to `ContextSwitch` will be passed unmodified to `MySwitchFunc`. You should use them, for example, to point to the current process’s PCB and to the PCB of the new process to be context switched in, respectively. In `MySwitchFunc`, you should do whatever you need to do to context switch between these two processes.

The `ctxp` argument (the pointer itself) passed to `ContextSwitch` is likewise passed unmodified to `MySwitchFunc`. However, before calling `MySwitchFunc`, the `ContextSwitch` function saves a copy of the current CPU state (e.g., registers) in this `SavedContext` structure. When returning from `MySwitchFunc`, you should return a pointer to the `SavedContext` that records the CPU state you want to have restored when `ContextSwitch` returns. Inside your function `MySwitchFunc`, before returning, you should also do the necessary operations to switch from one process’s Region 0 page table to the other process’s Region 0 page table.

Specifically, you should include a `SavedContext` within each process’s PCB. Suppose this member of the PCB structure is called `ctx`, and suppose `pcb1` currently points to the PCB of the process that you are context switching from and `pcb2` points to the PCB of the process you are switching to. Then, to context from a first process indicated by `pcb1` to a second process indicated by `pcb2`, you would call `ContextSwitch` as follows:

```c
ContextSwitch(MySwitchFunc, &pcb1->ctx, (void *)pcb1, (void *)pcb2)
```

Inside `MySwitchFunc`, you should switch from using the Region 0 page table associated with the process indicated by `pcb1` to using the Region 0 page table associated with the process indicated by `pcb2`. Then,
when MySwitchFunc returns, it should return &pcb2->ctx. To later context switch back to the first process, you would call ContextSwitch again with the roles of pcb1 and pcb2 reversed. The switching from one Region 0 page table to another must be done from within MySwitchFunc.

The actual SavedContext structure definition is provided by comp421/hardware.h. However, you don’t need to worry what is in a SavedContext. On a real machine, this structure would be full of low-level hardware-specific stuff; on the RCS 421 simulation, it is full of low-level Linux-specific (and internal simulation-specific) stuff.

When you call ContextSwitch, the current process will in effect be blocked inside the kernel exactly where it called ContextSwitch, with exactly the state that it had at that time. ContextSwitch and MySwitchFunc do the context switch, and when this process is later context switched back to, this process’s earlier call to ContextSwitch returns. The return value of ContextSwitch in this case is 0. If, instead, any error occurs, ContextSwitch 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 ContextSwitch 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. ContextSwitch only helps you with actually switching the state of the process and later restoring it. Your own kernel code has to take care of the rest itself.

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

### 3.6 Bootstrapping and Kernel Initialization

As noted above in Section 3.3, your kernel begins execution at a function named KernelStart, which you must write. Your kernel does not have a main function. Your KernelStart function is called with a pointer to an ExceptionInfo structure (among other arguments). Your kernel should use this ExceptionInfo as the basis for “cloning” others.

In particular, your KernelStart function should create both an idle process and an init process; as noted in Section 6.1, the idle process should have Yalnix process ID 0 and the init process should have Yalnix process ID 1. One of these two processes will run as the first process when KernelStart returns (normally, this should be init, but for testing, you could try running idle first instead). The other of these two processes will be context switched to possibly sometime later. At that time, this process must have an initialized kernel stack, since when ContextSwitch returns after this context switch, this process will be executing in your kernel. Likewise, you must have a corresponding initialized SavedContext to use in loading the CPU state of this process.

In general, before allowing the execution of user processes, the KernelStart routine should perform any initialization necessary for your kernel or required by the hardware. In particular, in addition to any initialization you may need to do for your own data structures, your kernel initialization should probably include the following steps (the order of some of these steps can be changed to suite how you want to structure your initialization):

- Initialize the interrupt vector table entries for each type of interrupt, exception, or trap, by making them point to the correct handler functions in your kernel.

- Initialize the REG_VECTOR_BASE privileged machine register to point to your interrupt vector table.
• Build a structure to keep track of what page frames in physical memory are free. For this purpose, you might be able to use a linked list of physical frames, implemented in the frames themselves. Or you can have a separate structure, which is probably easier, though slightly less efficient. This list of free page frames should be based on the `pmem_size` argument passed to your `KernelStart`, but be careful not include any memory that is already in use by your kernel.

• Also, as your kernel later allocates and frees pages of physical memory, be very careful not to accidentally end up using the same page of physical memory twice for different uses at the same time; for example, be careful when you allocate a free page of physical memory, to actually remove that page from your list of free physical pages, and when you free a page of physical memory, to actually add that page back correctly (and only once) to your list of free physical pages. If you do end up using the same page of physical memory twice for different uses at the same time, the results can be very strange and difficult to debug (think about why this is so).

• Build the initial page tables for Region 0 and Region 1, and initialize the registers `REG_PTR0` and `REG_PTR1` to define these initial page tables.

• Enable virtual memory.

• Create an “idle” process to be run by your kernel when there are no other runnable (ready) processes in the system. The idle process should be a loop that executes the `Pause` machine instruction on each loop iteration. The idle process can be loaded from a file using `LoadProgram`, in the same way as a normal user process. Or, if you think about it, it is possible to have the code for the idle process “built into” the rest of the code for your kernel, for example by placing the idle process’s code in a separate C procedure, so that the machine language code for the idle process then automatically gets loaded into memory along with the rest of your kernel; you wouldn’t actually call this procedure, but rather you can initialize the `pc` value for the idle process to the address of this code for idle.

• Create the first “regular” process, known as the `init` process, and load the initial program into it. In this step, guide yourself by the file `load.template` that we provide, which shows a skeleton of the procedure necessary to load an executable program from a Linux file into memory as a Yalnix process. This initial process will serve a role similar to that of the “init” process in Unix/Linux as the parent (or grandparent, etc.) of all processes running on the system. However, in Yalnix, unlike in Unix and Linux, when a process’s parent exits, that process is not “inherited by” the init process. Rather, as noted in Section 3.1, that process continues to run but simply has no parent process. To run your initial program you should put the file name of the init program on your shell command line when you run your kernel. This program name will then be passed to your `KernelStart` as one of the `cmd_args` strings.

• Return from your `KernelStart` routine. The machine will begin running the program defined by the current page tables and by the values returned in the `ExceptionInfo` structure (values which you have presumably modified to point to the program counter and stack pointer, etc., of the initial process).

3.7 Loading Yalnix User Programs from Disk

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

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

Specifically, we provide a template for a function called LoadProgram, which can load a program from a Linux executable file into a Yalnix process address space. This function template is in the file load.template in the directory /clear/courses/comp421/pub/templates. You should make a copy of this 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 >>>> and follow the instructions there.

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

In addition, this file is only a template. You should feel free to modify it in any way you want to or need to in order to utilize it in your kernel. For example, you may add arguments to the LoadProgram function, as you like.

Also, note that you can only load and execute user programs under Yalnix that have been compiled and linked using the rules in the provide Makefile.template (see Section 5). In particular, you cannot run under Yalnix any programs that have been compiled and linked instead to run under Linux.

3.8 Additional Terminal Interface

As described above in Section 2.3.1, the hardware interface to the terminals is through the TtyTransmit and TtyReceive hardware instructions. Only your kernel can execute these instructions. Your Yalnix kernel must provide kernel calls TtyWrite and TtyRead to allow user programs access to the terminals.

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

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

that works similarly to the standard Unix printf function, but does its output using the Yalnix TtyWrite kernel call. The first argument to TtyPrintf is the Yalnix terminal number to write to. 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 comp421/hardware.h).

3.9 Shutting Down the Yalnix Kernel

To shut down your Yalnix kernel, execute a Halt instruction from within your kernel (i.e., while in kernel mode).

Specifically, whenever your kernel handles an Exit() for the last process or the last process is terminated by your kernel (e.g., from an exception), you should execute a Halt to shut down the entire kernel. That is, whenever there are no more remaining processes in existence (other than the idle process), your kernel should Halt the machine. The idle process itself should never call Exit() or be terminated by your kernel, so that it is available to run when all other processes are blocked for some reason. You may also Halt the machine from within your kernel at other times, if you need to as part of handling errors encountered by your kernel, or if you are (temporarily) using it to aid in debugging your kernel.
As noted above, the Halt instruction completely stops the CPU, and the hardware simulation then ends. Before calling Halt in your kernel, you should print an informative message, including stating why you are shutting down the Yalnix kernel.

4 Project Logistics

4.1 Grading

This project will be graded primarily on correctness and efficiency. However, significant lack of documentation or elegance within your code will adversely affect your grade. You should use reasonable variable names and include some comments, particularly in tricky parts of the code, but, for example, you don’t need to put extensive comments in code that clearly just does what it looks like.

4.2 For Students Enrolled in COMP 521 or ELEC 552

For students taking COMP 521 or ELEC 552, the graduate student version of this course, in addition to implementing your Yalnix kernel as described here, you must also write and submit a report addressing issues including the factors affecting the performance of the design in your solution to the project and the issues involved in scaling the project to larger sizes in larger systems. Students taking COMP 421 or ELEC 421, the undergraduate student version of this course, may ignore this section, as this project requirement does not apply to you.

For students taking the graduate version of this course, your report must be written individually and not in collaboration with your partner in your group for this project; each student must write their own individual report. Your report should address issues such as, but not necessarily limited to, the general points below:

- What design choices did you or your group make in your solution to the project that you intended to improve the performance of your solution? Explain these design choices and why they should improve performance.

- What design choices would you or your group have made in your solution, if you’d had more time to implement them, that could have improved the performance of your solution? Explain these design choices and why they should have improved performance.

- What are the biggest issues affecting the ability of your solution, and what your solution could have been if you’d had more time, to scale to larger sizes and larger systems? Think about things such as systems with more physical memory, or a larger virtual address space, more processes being run at the same time, more frequent context switches, etc. Think about factors including both execution time performance and memory consumption.

In thinking about and discussing these issues in your report, you should consider them from the point of view of how they would affect an entirely real system. For example, do not discuss issues of performance or how to scale the system in light of the simulated hardware used in the project; rather, discuss performance and scale as would be appropriate for your kernel on entirely real hardware. Also, try to be as specific as you can in discussing each issue, rather than just making generalized, vague statements.

Your report must be limited to a maximum of 6 pages, although there is no minimum required length for your report; the report may be formatted in either single-column or double-column format and should use a font size no smaller than 10 points. Your report will be graded based on what you say in it, not on how many pages you fill. In writing your report, please try to think critically about what you want to say and to say it clearly.
Note that your report should not simply be a description of your implementation. Rather, your report must specifically address the issues mentioned above.

In addition to the limit of a maximum report length of 6 pages, your report must be submitted in PDF format (i.e., not in other strange formats such as Microsoft Word).

Unlike for the first project, to submit your report for this project, please email it to me at 

dbj@cs.rice.edu

The file name for your report must be 

report-lab2-netid.pdf

where netid is your Rice NetID name. Only reports named as described above that are emailed to me will be graded. When I receive your report email, I will send you an acknowledgment email letting you I received your report; if you do not receive this acknowledgment email, please let me know right away. Do not include your report with the files that you submit using the lab2submit program (defined in Section 4.3 below).

The deadline for emailing your report to me is the same as the submission deadline for the project as a whole.

4.3 Turning in Your Project

Your project is due by 11:59 PM, Friday, March 30, 2018. The project should be done in groups of 2 students, and you should only submit one project for your group (only one group member should submit the project for the group).

Similar to turning in your Lab 1, everything you want to turn in for your group should be in a single directory (or its subdirectories). Before turning in your project, please create a plain text file named “README” in the directory where your files for the project are located. Your “README” file must be in plain text format (i.e., not in PDF format or in other strange formats such as Microsoft Word).

Please describe in your “README” file anything you think it would be helpful for the TAs to know in grading your project. This might, for example, mean describing unusual details of your algorithms or data structures, and/or describing the testing you have done and what parts of the project you think work (or don’t work). Also, in your “README” file, please list the names and NetIDs of both members in your project group.

Actually submitting your project for grading will be done similarly to what was done for Lab 1. Specifically, to submit your project for grading when you are ready, one of the group members should perform the following two steps:

• First, on CLEAR, change your current directory to the directory where your files for the project for grading are located. For example, use the “cd” command to change your current directory. When you run the submission program, it will submit everything in (and below) your current directory, including all files and all subdirectories (and everything in them, too). Please make sure all files you want us to see for grading are located in this single directory and/or in subdirectories of this single directory.

• Second, on CLEAR, run the submission program 

/clear/courses/comp421/pub/bin/lab2submit

This program will check with you that you are in the correct directory that you want to submit for grading, and will confirm with you your name and the name of the partner you have been working with on the project. Finally, the lab2submit program will normally just print
SUCCESS! Your Lab 2 project submission is complete.

If you get any error messages in running `lab2submit`, please let me know.

- After your submission, you should also receive an email confirmation of your submission, including a listing of all of the individual files that you submitted.

*Only one of the two group members need to do the above steps.* If both members of a project group run the `lab2submit` program, the one who runs it second will simply overwrite the submission of the one who ran it first. Only one submission per group will be recorded.

Note, however, that *both students in a group are jointly responsible for what the group submits for their project*. Only one of the two students in a group actually does the project submission, but both partners should agree on what is to be submitted. Also, after a project submission, both partners should receive email (also sent automatically to the instructor) detailing the submission and the files that were included in it. Please carefully check this email when you receive it to make sure that the right files were submitted. In any case, if you have a concern about what your partner has submitted for your group, try to work it out with your partner (and if needed, you can redo the submission with a different version of the project files); if you still have a concern about a submission by your group, contact the instructor.

And again, for students taking the *graduate student* version of this course, COMP 521 or ELEC 552, *do not* include your project report in the files that your group submits with the `lab2submit` program. Rather, please email your report to me as described above in Section 4.2. Also remember, *each* student must write their *own individual* report, not jointly together with your group partner for the project (or together with anyone else).

### 4.4 Recommended Project Milestone Checkpoint

This is a large project, with a significant amount of time to work on it before the due date. We offer the following advice to help you work your way successfully through the project:

- Start working on the project early (like now, if you haven’t started already).

- Try to work on the project consistently throughout this time.

If you have any questions, ask a TA or the instructor, or post a message on Piazza.

As a further aid in helping you complete the project by the due date, the schedule of Important Dates for this project at the beginning of this handout includes a recommended *project milestone checkpoint* as an intermediate milestone point within the project. By the date listed above, you should have completed a substantial portion of the project. As a *minimal* guideline, by this date, we suggest that you should have completed the following project milestone: your Yalnix kernel should by then be able to successfully do the following:

- Execute `KernelStart` successfully, initializing the kernel and the machine, including initializing interrupts, enabling virtual memory, and creating the `idle` and `init` processes.

- Be able to execute a simple `init` process that only performs the `GetPid` kernel call, the simplest of the required kernel calls.

- Be able to execute a more interesting `init` process that successfully uses the `Delay` kernel call to delay itself, causing your kernel to context switch to the `idle` process while `init` is blocked, and be able to context switch back to `idle` when it becomes unblocked when its `Delay` has been completed. Be sure your kernel can successfully context switch back and forth more than once by using `Delay` multiple times in your test `init` process.
Note that this should be a minimum goal to achieve by the checkpoint deadline. Indeed, you are strongly encouraged to get to this milestone well before then, and to complete more of the project by the checkpoint date.

The project milestone checkpoint will not be graded, and there is nothing to turn in for it. This schedule and goal for the checkpoint is for your own use and guidance in working on the project. You are ultimately responsible for completing the entire project by the overall due date, but if you have not completed this milestone on schedule, you will find it very hard to finish the whole project by the due date (remember to also allow time in your schedule planning for the COMP 421 midterm exam, as listed above in the Important Dates). Remember, start on the project early and try to work on it consistently throughout the available time.

5 Your Assignment

To review, in order to complete the project, you must implement a Yalnix kernel that runs on the provided RCS 421 computer simulation. Specifically, you must implement:

- a `KernelStart` routine to perform kernel initialization;
- a `SetKernelBrk` routine to change the size of kernel memory;
- a procedure to handle each defined type of interrupt, exception, or trap; and
- a procedure (called by your `TRAP_KERNEL` handler) to implement each defined Yalnix kernel call.

Note that the kernel call prototypes given in Section 3.1 and in `comp421/yalnix.h` are the form in which Yalnix user-level programs use these kernel calls. The function names and prototypes of the procedures that implement them inside your need not be the same. The code in your `TRAP_KERNEL` handler calls whatever functions inside your kernel it wants to, with whatever arguments it wants to, in order to provide the effect of each type of kernel call.

The recommended structure for your kernel would be to make one function in your kernel for each type of kernel call, but to pick some other name for that function inside the kernel. For example, you could name the function you use to implement the `Delay` inside your kernel `KernelDelay`, and you could then pass any arguments to it and get any return value from it you decide is convenient in your own implementation. Since this procedure is entirely internal to your kernel, called only by your `TRAP_KERNEL` handler, there is no confusion between this and the `Delay` function called by user processes outside the kernel to invoke this function and to generate the trap necessary to enter your kernel to request it.

In addition, the following requirements apply to this project:

- Your solution must be implemented in the C programming language (e.g., not in C++ or other languages).
- Your solutions must run in the CLEAR/Linux environment, using the supplied RCS 421 computer system simulation environment, as described in Section 2.
- Your kernel must provide the interface and features described in Section 3.
- The project should be done in groups of two students.
- Your kernel must compile and link to an executable program named `yalnix` that is set up to run with the simulated RCS 421 environment provided.
• Your project directory must include a Makefile that can be used to make your kernel and any Yalnix user-level test programs you used in testing your kernel. A template Makefile is available as Makefile.template in the directory /clear/courses/comp421/pub/templates. You should copy this file to your project directory and edit it as described in the comments in the file. In particular, this template Makefile contains special rules for compiling and linking programs, which must be used for the project to work.

• For each individual student taking COMP 521 or ELEC 552, the graduate student version of this course, you must also write and separately email to me a report addressing issues including the factors affecting the performance of the design in your solution to the project and the issues involved in scaling the project to larger sizes in larger systems, as described above in Section 4.2. Each individual student must write and email to me their own separate report.

6 Some Implementation Hints

6.1 Process IDs

Each process in the Yalnix system must have a unique integer process ID, starring with the idle process having Yalnix process ID 0 and the init process having Yalnix process ID 1. Since an unsigned integer allows over 4 billion processes to be created before overflowing its range, you may (for simplicity in this project) assign sequential numbers to each new process created and not worry about integer overflow (real operating system kernels must worry about this, though).

6.2 Fork

On a Fork, a new process ID is generated and a new PCB is allocated. The kernel stack (which contains the saved exception information (ExceptionInfo) structure from when the user executed the TRAP_KERNEL to call Fork) and saved kernel context for the child process are created as copies of the parent (calling) process. New physical memory is allocated for the child process, into which you copy the parent’s address space contents. Since the CPU can only access memory by virtual addresses (using the page tables), you must map both the source and the destination of this copy into the virtual address space at the same time. You need not map all of both address spaces at the same time, however: the copy may be done piece-meal, since the address spaces are already naturally divided into pages (e.g., you can do the copy a page at a time).

6.3 Exec

On an Exec, you must load the new program from the specified (Linux) file into Region 0 of the calling process, which will in general also require changing the process’ page tables. The program counter in the pc field of the exception information (ExceptionInfo) structure must also be initialized to the new program’s entry point. The template LoadProgram function described in Section 3.7 shows how to do this and takes care of most of the messy details for you.

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

To complete initialization of the stack for a new program, the argument list from the Exec must first be copied onto the stack. The template LoadProgram function also takes care of most of this for you. The stack pointer in the sp field of the exception information (ExceptionInfo) structure should be
initialized by LoadProgram. Like all addresses that you use, this must be a virtual address. Again, all these details are presented in the form of the C procedure LoadProgram in the file load.template.

6.4 Suggested Plan of Attack

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

- Read this document carefully. Try to understand the details described in it, and perhaps read this document again. Ask a TA or the instructor if you don’t understand parts of this document. Although there are a lot of pages here to read, this document is intended to be very complete and to guide you through the project.

- Think through and sketch out some high-level pseudo-code for at least some of the kernel calls, interrupts, and exceptions. Then decide on the things you need to put in what data structures (notably in the Process Control Block, or PCB) to make it all work. Iterate until the pseudo-code and the main prototype data structures mesh well together. Do not worry whether you have everything you will eventually need defined in your PCB structure; you can always add fields to the PCB later (although you must be careful that you recompile everything that uses the PCB definition if you add or change fields in the PCB structure definition).

- In designing your kernel data structures, think carefully about the difference in the C programming language between a data structure created as an “automatic” variable, one created as a “static” or external variable, and one whose storage was allocated by malloc. In particular, think about where the memory (the storage) for each of these types of variables is located and when (if ever) that memory will be freed. Be careful in which way you create each of your kernel’s data structures.

- Read about the tracing feature using TracePrintf, described in Section 7.2, that you can use in this project. As you work on the project, particularly during debugging, you may find this feature very useful; with it, you can, for example, on the command line dynamically turn up or down (or on or off) different types of debugging tracing of your kernel or Yalnix user programs (or the RCS 421 hardware).

- Take a look at the template LoadProgram function in the file load.template. You need not understand all the details, but make sure you understand the comments that are preceded by “>>>>” markers.

- Write an initial version of your KernelStart function to bootstrap the kernel. You need to initialize the interrupt vector table here and point the REG_VECTOR_BASE register at it. Note, however, do not start out with an empty procedure for each of your interrupt, exception, or trap handlers, as doing so can make your subsequent debugging sometimes very confusing. Instead, it is highly recommended that you write a separate handler procedure for each defined entry in the interrupt vector table, with at least a TracePrintf call and a Halt in it; you can then replace that Halt with the real handler code in that procedure once you have written it. In your KernelStart function, you also need to build the initial page tables and enable virtual memory by writing a 1 into the REG_VM_ENABLE register. The support code that makes the RCS 421 simulation work does a lot of error checking at this point, so if you get through this far with no errors, you are probably doing OK (although we can’t check for all possible errors at this point).
• Write an *idle* program (a single infinite loop that just executes a *Pause* instruction). Try running your kernel with just an *idle* process to see if that much works. The *idle* process should have Yalnix process ID 0.

• Write a simple *init* program. The *init* process should have Yalnix process ID 1. The simplest *init* program would just loop forever, but this form of the *init* program would generally only be useful for testing; normally (for example, as suggested in the following steps), *init* should do something “real,” including calling some Yalnix kernel calls, creating other processes, etc. Make sure you can get your Makefile to compile this as a Yalnix user program, since you will need this skill in order to write other test programs later. Modify your *KernelStart* to start this *init* program (or one passed on the Linux shell command line) in addition to the *idle* program.

• Implement the *GetPid* kernel call and call it from the *init* process. At this point your kernel call interface would seem to be working correctly.

• Implement the *Delay* kernel call and call it from the *init* process. Make sure your *idle* process then runs for several clock ticks, until the delay period expires. This will be the first proof that blocking of processes works and context switching work.

• Try *Delay* several times in the same program, to make sure your *init* and *idle* processes can correctly context switch back and forth several times. At this point you will have basically achieved the recommended checkpoint described in Section 4.4.

• Implement *SetKernelBrk* to allow your kernel to allocate substantial chunks of memory. It is likely that you haven’t needed it up to this point, but you may have (in this case implement it earlier).

• Implement the *Brk* kernel call and call it from the *init* process. At this point you have a substantial part of the memory management code working.

• Implement the *Fork* kernel call. If you get this to work you are almost done with the memory system.

• Implement the *Exec* kernel call. You have already done something similar by initially loading *init*.

• Write another small program that does not do much. Call *Fork* and *Exec* from your *init* process, to get this third program running as a child of *init*. Watch for context switches.

• Implement and test the *Exit* and *Wait* kernel calls.

• Implement the kernel calls related to terminal I/O handling. These should be easy at this point, if you pay attention to the process address space into which your input needs to go.

• Look at your work and wonder in amazement at the uncertain, frustrating, and rewarding road you have traveled.

## 7 Compiling, Running, and Using Yalnix

### 7.1 Compiling Your Kernel

Your kernel must be implemented in the C programming language (e.g., not in C++ or other languages). You may divide the source code for your kernel into any number of separate `.c` files, as you see fit.

The file `/clear/courses/comp421/pub/templates/Makefile.template` contains a basis for the Makefile I recommend you use for this project. The comments within it should be self-explanatory. In any edits you make to your Makefile, the arguments passed to the compiler and linker
to make the kernel and Yalnix user programs should not be changed. the Makefile, once modified as described here and in the comments in the template, will be able to automatically compile and link together whatever individual .c source files you used for your kernel; the resulting Yalnix kernel will be a program named yalnix in your current directory.

7.2 Running (Booting) Your Kernel

Your kernel will be compiled and linked as an ordinary Linux executable program, and can thus be run as a command from the CLEAR Linux shell prompt. When you run your kernel, you can put a number of Unix-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 from the CLEAR shell command line as:

```
./yalnix startup args init program init args...
```

All of the switches and arguments shown here on the command line are optional.

The startup args, if present on the command line, may include any of the following arguments that affect how the hardware simulation starts up:

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

To generate trace output from your kernel, you may call

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

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

- `-lk level`: Set the tracing level for the kernel for this run. The default tracing level is -1, if you enable tracing with the `-t` or `-s` switches. You can specify any level of tracing.

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

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

- `-n`: Do not use the X window system support for the simulated terminals attached to the Yalnix system. The default is to use X windows.

- `-s`: Send the tracing output to the Linux `stderr` file (this is usually your screen) in addition to sending it to the tracefile. This switch enables tracing, even if the `-t` switch is not specified.
• \(-P \ pmem\_size\): This argument is never required, but optionally, you may use it to test your Yalnix kernel with a different amount of physical memory installed in the (simulated) hardware. The value \(pmem\_size\) defines the size of the physical memory of the machine you are running on. The size of physical memory is given as \(pmem\_size\) in units of bytes. Using this argument is equivalent (in real life) to purchasing more physical memory for your computer, opening the computer case, and installing that extra memory. If you run your kernel without this argument, the default physical memory size is used.

The TracePrintf tracing feature is designed to aid in your debugging, and it is strongly recommended you make use of it in your kernel. Specifically, TracePrintf is similar to normal printf but is implemented specially to provide much more capability than normal printf.

First, with TracePrintf, you can, for example, on the command line dynamically turn up or down (or on or off) different types of debugging tracing of your kernel or Yalnix user programs (or the RCS 421 hardware). In particular, when putting TracePrintf calls into your code, you can use any integer value for level you want to, but you will generally want to use lower numbers for basic tracing information you want to see while your kernel executes, using higher level numbers for tracing more detailed information that you don’t want to (or need to) see every time you run your kernel; you can use a variety of different level values throughout your code, allowing more fine-grained selection of what tracing messages you see depending on what value you use on the command line when you run your kernel.

In addition, TracePrintf is implemented to depend on only very little of your kernel functioning correctly; you could have major things missing or incorrect in your kernel yet calls to TracePrintf are likely to still be able to execute successfully, whereas calls to printf in similar circumstances may fail and crash your kernel.

The \(startup\_args\) command line arguments described above are automatically parsed and interpreted for you before your KernelStart is called. The remaining arguments from the command line, after removing any of these \(startup\_args\) arguments, are passed to KernelStart in its \(cmd\_args\) argument. For example, if you run your kernel with the shell command line

\[
./yalnix -lk 5 -lh 3 -n init a b c
\]

then your KernelStart would be called with its \(cmd\_args\) argument set as follows:

\[
\begin{align*}
\text{cmd\_args}[0] & = "init" \\
\text{cmd\_args}[1] & = "a" \\
\text{cmd\_args}[2] & = "b" \\
\text{cmd\_args}[3] & = "c" \\
\text{cmd\_args}[4] & = \text{NULL}
\end{align*}
\]

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” if no \(cmd\_args\) are specified when Yalnix is run from the Linux shell.

7.3 Debugging Your Kernel

You may use TracePrintf and/or fprintf, printf, etc. within your kernel to generate any necessary debugging output you want to see; we recommend that you use TracePrintf. Please be aware that the insertion of such print statements may alter the behavior of your program. For example, they can change the timing of when things happen relative to when clock interrupts or terminal interrupts happen. Since procedure calls such as these utilize space on the stack, these calls may also have the side-effect of changing
the value of “uninitialized” variables in your code. And inserting them at various places in your code can possibly change the address of other parts of your kernel’s code or variables, changing what you might end up accessing using an “uninitialized” pointer variable or a pointer whose value has accidentally gotten overwritten by some bug in your code.

You may also use gdb to debug your kernel. However, because of the RCS 421 simulator’s heavy use of Unix signals internally, it is necessary to tell gdb to ignore certain events:

- For gdb, you can do this by typing the following at the gdb prompt:

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

Putting these commands in your .gdbinit file will relieve you from needing to type them every time you run gdb to debug your kernel.

7.4 Controlling Your Yalnix Terminals

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

The current RCS 421 machine configuration supports four terminals, numbered from 0 to 3, with terminal 0 serving as the Yalnix system console, and the other 3 serving as regular terminals. In fact, though, these uses are only a convention, and all four terminals actually behave in exactly the same way. The constant NUM_TERMINALS in comp421/hardware.h defines the total number of terminals supported. The constant TERMINAL_MAX_LINE defines the maximum length of line supported for either input or output on the terminals.

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

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

to act as the four terminals that the hardware configuration supports. Each of these windows is actually running a Unix xterm and can be controlled as such. For example, you can resize or inconify any of these windows, and can use the scroll bar to review output in the window. For those who like to play with configuring X windows, you can define the shell environment variable YALNIX_XTERM to contain xterm command line arguments to be used when starting the Yalnix terminal windows. You can also define the shell environment variables YALNIX_XTERM0, YALNIX_XTERM1, YALNIX_XTERM2, and YALNIX_XTERM3 to be further command line arguments to be used only when starting the corresponding terminal window. For example, you could define these four environment variables to contain appropriate “-geometry” options to automatically place each window on the screen for you. You can also include xterm options to change the font size in each window, change the foreground or background colors, etc.

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

Press RETURN to end simulation...

Once you hit the RETURN key, the machine simulation will exit and the console and terminal windows will disappear.
The terminal windows keep log files of all input and output operations on each terminal. The files `TTYLOG.0`, `TTYLOG.1`, `TTYLOG.2`, and `TTYLOG.3` record all input and output from each terminal separately; and the file `TTYLOG` collectively records all input and output from the four terminals together. The terminal logs show the terminal number of the terminal, either “<” for input or “>” for output, and the line input or output. You can type an end of file on a terminal with control-D as in Unix/Linux, which appears in the terminal log files as “`(EOF)`”.

8 COMP 421 on Piazza

As with Lab 1, we will be using Piazza for class discussion (please use the “lab2” folder). Again, when posting questions or answers about the project on Piazza, please be careful about what you post, to avoid inadvertently violating the COMP 421 course Honor Code policy described in the course syllabus and below in Section 9. Specifically, please do not post details about your own project solution, such as portions of your source code or details of how your code works, in public questions or answers on Piazza. If you need to ask a question that includes such details, please make your question private on Piazza by selecting “Instructor(s)” (rather than “Entire Class”) for “Post to” at the top, so that only the course instructor and TAs can see your posting.

Please try to regularly check Piazza for new questions and answers. These Piazza posts can be a valuable resource to you as you work on the project, for example clarifying issues even before you have realized that you have a problem and need those issues clarified; checking this can thus potentially save you a lot of time and possible frustration during the project, and can save you from posting a redundant question on Piazza that has been asked and answered already. To easily check for new messages on Piazza, you can, for example, click on the “Unread” or “Updated” buttons near the upper-left corner of the Piazza web view, to show any messages that, respectively, you have not read yet or have been updated since you last read them. You can also use the Piazza Search functionality to look for existing messages related to a given question you have or problem you may be having.

9 Honor Code Policy

The Honor Code is a special privilege and responsibility at Rice University. As stated in a student editorial published in the January 20, 2016 edition of The Rice Thresher: “As incoming students enter Rice, many are surprised by the degree to which the university’s Honor Code extends trust to the student body. . . . The privileges of the Honor Code stem from the idea that Rice’s aim is not just to instill knowledge in its students, but [to] also help them develop moral character. This idea is fundamental to Rice’s identity: Students can and should be held to a high moral character standard.”

Specifically, as previously stated, all assignments in this course are conducted under the Rice Honor Code, a code that you pledged to honor when you matriculated at Rice. You are expected to behave in all aspects of your work in this course according to the Rice Honor Code. When in doubt as to whether a specific behavior is acceptable, ask the instructor for a written clarification. Suspected Honor Code violations on the projects and/or exams in this course will be researched, documented, and reported in extensive detail to the Rice Honor Council or Rice Graduate Honor Council. For more information on the Rice Honor System, see http://honor.rice.edu/ and http://gradhonor.rice.edu/. In particular, you should consult the Honor System Handbook at http://honor.rice.edu/honor-system-handbook/
This handbook outlines the University’s expectations for the integrity of your academic work, the procedures for resolving alleged violations of those expectations, and the rights and responsibilities of students and faculty members throughout the process.

For the programming assignments in this course, students are encouraged to talk to each other, to the TAs, to the instructor, or to anyone else about the assignment. This assistance, however, must be limited to general discussion of the problem; each student or project group must produce their own solution to each programming project. Consulting or copying, in any manner, another student’s or project group’s solution (even from a previous class or previous year) is prohibited, and submitted solutions may not be copied from any source. Also, as noted above in Section 4.3, although only one of the two students in a project group should actually do the submission using the lab2submit program, both students in the group are responsible for what the group submits.

Also, for students taking COMP 521 or ELEC 552, for all projects, each student must write their own report on the project; for the last two projects, for which the project itself will be done in groups of two students, each student in the group must still write their own report, not working together on your report with your group partner. Submitted reports must not be copied, in whole or in part, from any source, and you must fully cite any references you use in preparing your report.

In addition, for all programming assignments, you may not place source code for your project on any publicly accessible repository (such as GitHub), including even after the end of the semester; to do so would be a violation of the Honor Code, as it would give aid to other students on the project. Also, if any such public repositories do exist, you may not refer them (or other such sources) in working on or producing your solution for such project.

I want to treat you all in this class as responsible adults. But please be aware that cheating on any of the programming projects in this class constitutes a Rice Honor Code violation. Submitting a case to the Rice Honor Council requires a lot of work on my part, but it will have a much larger impact on you, your future status as a student at Rice University, and your prospects for a degree from Rice. Please do not make me have to submit any Honor Council cases in this class. This will help both you and me. As I said, I want to treat you all in this class as responsible adults.
# Table of Contents

1 Project Overview .................................................. 1

2 The RCS 421 Computer System Hardware ................................ 2
   2.1 Machine Registers ............................................. 2
       2.1.1 General Purpose Registers ................................. 2
       2.1.2 Privileged Registers ....................................... 2
   2.2 Memory Subsystem ............................................. 3
       2.2.1 Overview .................................................. 3
       2.2.2 Physical Memory ......................................... 4
       2.2.3 Virtual Address Space .................................. 5
       2.2.4 Page Tables ............................................. 6
       2.2.5 Translation Lookaside Buffer (TLB) ...................... 8
       2.2.6 Enabling Virtual Memory ................................ 8
   2.3 Hardware Devices ............................................. 9
       2.3.1 Terminals ................................................. 9
       2.3.2 The Hardware Clock ..................................... 10
   2.4 Interrupts, Exceptions, and Traps ................................ 10
       2.4.1 Interrupt Vector Table and Types ....................... 10
       2.4.2 Hardware-Defined Exception Information Structure .... 11
   2.5 Additional CPU Machine Instructions ............................ 13

3 Yalnix Operating System Details ..................................... 14
   3.1 Yalnix Kernel Calls ........................................... 15
   3.2 Interrupt, Exception, and Trap Handling ............................ 18
   3.3 Kernel Booting Entry Point .................................. 20
   3.4 Memory Management .......................................... 20
       3.4.1 Virtual Address Space Layout ............................. 21
       3.4.2 Kernel Memory Management ............................... 22
       3.4.3 Initializing the Page Tables and Virtual Memory .......... 23
       3.4.4 Growing a User Process’s Stack ......................... 25
   3.5 Context Switching ............................................. 25
   3.6 Bootstrapping and Kernel Initialization .......................... 27
   3.7 Loading Yalnix User Programs from Disk ......................... 28
   3.8 Additional Terminal Interface ................................ 29
   3.9 Shutting Down the Yalnix Kernel ................................ 29

4 Project Logistics .................................................... 30
   4.1 Grading ....................................................... 30
   4.2 For Students Enrolled in COMP 521 or ELEC 552 ................. 30
   4.3 Turning in Your Project ...................................... 31
   4.4 Recommended Project Milestone Checkpoint ...................... 32

5 Your Assignment ................................................... 33
6 Some Implementation Hints
   6.1 Process IDs .............................................................................. 34
   6.2 Fork .......................................................................................... 34
   6.3 Exec ......................................................................................... 34
   6.4 Suggested Plan of Attack ............................................................ 35

7 Compiling, Running, and Using Yalnix
   7.1 Compiling Your Kernel ............................................................... 36
   7.2 Running (Booting) Your Kernel ................................................ 37
   7.3 Debugging Your Kernel .............................................................. 38
   7.4 Controlling Your Yalnix Terminals .............................................. 39

8 COMP 421 on Piazza ....................................................................... 40

9 Honor Code Policy .......................................................................... 40