The Yalnix Kernel

CS230/330 Operating Systems, Spring 2000

1 Introduction

For most of the quarter, you will be designing and implementing a simple operating system kernel. This document is intended to provide an overview of the entire kernel project. At first, this is going to appear very daunting if not impossible. However, keep in mind that the project is divided into several well-defined sub-projects (to be handed out separately) and that they follow the topics presented in the class lectures. In addition, your instructor and TAs have all had extensive experience working on prior versions of the kernel project. Therefore, do not panic if you do not understand every bit of this document after a first reading—it will all become clear in due course. In the meantime, you might just want to sleep with this document under your pillow so you can soak up all of those OS vibes.

Important Note: You will not receive a passing grade for the course unless your complete kernel passes a series of basic tests. It is critical that you not fall behind and that you see us for assistance if you run into trouble. We *want* you finish the project and are here to help you do just that.

2 Overview

Having broken his last three Palm Pilots, Dustin (a.k.a., "fearless TA") Mitchell has decided that he needs to create a new virtually indestructible hand-held computing device. Therefore, during his vast amounts of free time, he decided to design his own mobile computing system–something that he has decided to call the "Survivalist." However, since Dave is only paying him $17.42 per week, the only way he could obtain the resources to build the prototype was to turn to some slick venture capitalists from Seattle. Unfortunately, after raising $10 million in VC, they decided that they'd much rather take the money and move to Kona, Hawaii to become extreme surfers. As a result, the contract to the build the prototype was left to Bubba's Pipe Welding and Dot-Com Emporium of Amarillo, Texas for a grand total of $1745.52. Unfortunately, Bubba, just back from his Y2K bomb-shelter in Aspen, Colorado wasn't feeling particularly motivated to spend much time on the project. Therefore, he haphazardly assembled a prototype from a few spare parts that happened to be lying around the shop and put them together into a 16 pound cast-iron case. Although the prototype shorted out and burst into a rather spectacular white-hot ball of flame shortly after power-on, Bubba had fortunately constructed a hardware emulator for the "not-so-impervious-to-destruction" Survivalist system for use on SPARC machines running Solaris.

Using this emulator, your task is to implement an operating system kernel for the Survivalist hand-held computer system, which in deference to Bubba and his bomb shelter you have decided to call "Y2K" which is short for "Yalnix 2000" where "Yalnix" is an awful Texas-based pun on "Unix" and "2000" is, well, something that the marketing department decided to add as an afterthought. Yalnix includes the main process and memory management features of Unix, plus support for a variety of I/O devices including a serial port, a graphical display device, a pen-based input device, and a flash-memory storage card. Although your kernel needs to talk to a variety of low-level devices, it will not be necessary to program any portions of your project in assembly language nor will you need to worry about any low-level CPU-specific details. However, you will need to implement process management, memory management, process scheduling, an I/O subsystem, support for a simple filesystem, and interprocess communication using pipes.

To do this, your kernel will implement three distinct kinds of "interfaces." First, your kernel must support user-requests in the form of system calls. Second, your kernel must handle error conditions in the form of traps generated by the MMU and CPU. Finally, the kernel must service interrupts generated by various I/O devices. In addition to this, your kernel will include an additional entry point that is used during the boot process.

This document describes the hardware environment in which your Yalnix kernel will run along with the complete set of system calls that must be provided to user processes.

3 The Survivalist Hardware

The raw hardware is controlled through a series of priviledged machine registers and a small collection of BIOS functions. In addition, the hardware also generates a number of *traps* (or interrupts) that your kernel will have to handle.

This section describes the raw hardware devices in some detail. In addition, the file `/yalnix/include/hardware.h` defines all of the constants and structures that you will need to use in your kernel implementation (you may want to refer to it as you read through this section).

3.1 Access to Privileged Machine Registers

The hardware contains a collection of special registers that can only be read or written using priviledged CPU instructions. These registers are used to control various I/O devices and hardware features on the machine. The following functions are used for this purpose:

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

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

- `unsigned int ReadRegister(int which)`

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

Each privileged machine register is identified by a unique integer constant passed as the `which` argument. The file `hardware.h` defines symbolic names for these constants. The symbolic names will be used for the rest of this document.

The registers provided by the hardware are:

Reg. Name	Description	Readable	Writable
REG_VECTOR_BASE	Interrupt vector Address	Yes	Yes
REG_VM_ENABLE	Enable virtual memory	Yes	Yes
REG_TLB_FLUSH	Flush the TLB	No	Yes
REG_PTBR0	Region 0 page table base	Yes	Yes
REG_PTLR0	Region 0 page table length	Yes	Yes
REG_PTBR1	Region 1 page table base	Yes	Yes
REG_PTLR1	Region 1 page table length	Yes	Yes
REG_DISPCTL	Display controller	No	Yes
REG_PENEVT	Pen event register	Yes	No
REG_FLBLKSZ	Flash block size	Yes	No
REG_FLNBLK	Number of blocks on flash device	Yes	No

The purpose of these registers will be described in more detail later. All privileged machine registers are 32 bits wide. Their values are represented by these two instructions as values of type `unsigned int` in C. You must use a C "cast" to convert other data types (such as addresses) to type `unsigned int` when calling `WriteRegister`, and must also use a "cast" to convert the value returned by `ReadRegister` to the desired type if you need to interpret the returned value as anything other than an `unsigned int`.

3.2 Traps (Interrupts)

In addition to control registers, the hardware generates a number of traps (or interrupts). These are signals that are sent to the kernel when an error occurs or when a device needs the attention of the operating system. More details about how you will use these traps will appear later in this document. For now, consider this to a brief list of the traps that can be generated by the hardware.

- TRAP_KERNEL

  This trap is generated when a user program wants to execute a system call. See section 5.3.

- TRAP_CLOCK

  This is a periodic interrupt generated by the machine's hardware clock. Your kernel will use this to implement time-sharing and to context-switch between different programs. See Section 5.2.

- TRAP_ILLEGAL

  This trap occurs when a user program executes an illegal CPU instruction. The kernel should kill the offending program when this occurs.

- TRAP_MEMORY

  This trap results when a user process tries to access a memory address that is not mapped in any of the hardware page tables. It can also occur if a user process tries to access a portion of memory for which it has insufficient access permissions (if a user program tries to access memory in the kernel for instance). See Section 4.1.

- TRAP_MATH

  This trap occurs when a math operation executed by a user program generates a fault. For example, division by zero. The process that generate the error should be terminated when this occurs.

- TRAP_SERIAL_RECEIVE

  This interrupt occurs when incoming data has been received by the serial I/O device.

- TRAP_SERIAL_TRANSMIT

  This interrupt occurs when outgoing data on the serial device has been successfully sent. See Section 5.7.3 for details.

- TRAP_PEN_EVENT

  This interrupt occurs whenever there is activity on the pen input device. After this interrupt has occurred, the status of the pen device should be read by reading the REG_PENEVT register. See Section 5.7.2.

- TRAP_FLASH_ACCESS

  This interrupt occurs whenever the last I/O operation performed on the flash card storage device has been completed. See Section 5.7.5

When a trap (interrupt) occurs, the CPU looks in an interrupt vector table to find out the location of a function that will be called to handle the interrupt. The above constants (e.g., TRAP_CLOCK) represent offsets into the vector table. As far as implementation, the interrupt vector table is nothing more than an array of pointers to functions that your kernel must initialize. To do this, the priviledged machine register

REG_VECTOR_BASE

must be set to point to the vector table created by the kernel. The vector table must be statically dimensioned by your kernel with size TRAP_VECTOR_SIZE, but only the entries defined above are used by the hardware. All other entries in the vector table must be initialized by your kernel to NULL.

The functions that handle traps (whose pointers are in the vector table) must have the following prototype:

void *Trap_Handler*(struct cpu_context_frame *)

(See section 4.3 for a description of the type struct cpu_context_frame).

Also, to simplify the programming of your kernel, the kernel is mostly not interruptible, That is, while executing inside your kernel, the kernel cannot be interrupted by any other interrupt, exception or syscall. Thanks to this, you are unlikely to need special synchronization procedures such as semaphores, monitors, or interrupt masking inside your kernel. Any interrupts that occur while inside your kernel are held pending by the hardware and will be raised only once you return from the current trap.

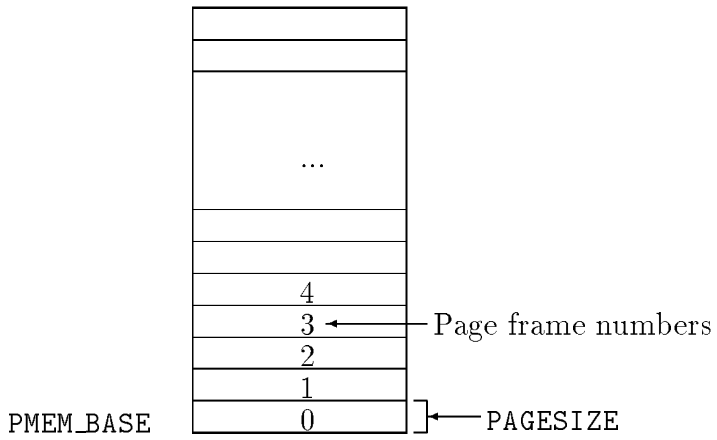

Figure 1: Physical Memory Layout

3.3 Memory Management Hardware

3.3.1 Physical Memory

Currently, the hardware provides large physical memory space which is to be shared by many concurrently executing processes. This memory begins at the address defined by `PMEM_BASE` which is found in the `hardware.h` file. Although the current version of the hardware provides 4 Mbytes of memory, the actual amount of memory is supplied to the kernel during the boot process and may vary according to the hardware configuration.

Logically, the physical memory is divided into a collection of pages of size `PAGESIZE`. These pages are referred to by numbers known as *page frame numbers*. To better illustrate the idea, the physical layout of the machine looks something like that shown in Figure 1.

Your kernel will need to keep track of what pages are in use and what pages are free. In practice, this isn't very hard. For example, in the current hardware, physical memory is divided up into 512 pages with a pagesize of 8192 bytes. To track free memory pages, you could just create an array of 512 entries and record a 1 or a 0 in each slot to indicate whether or not a page was in use (of course, there are also more elegant ways to do this).

3.3.2 Virtual Memory

Although the kernel and all user programs are stored in physical memory, all of these programs are run within a virtual memory space that is managed by a MMU controller which implements a paged memory mapping scheme using page tables.

The virtual address space of the machine is divided into two *regions* called region 0 and region 1. By convention, region 0 is used by the operating system kernel and region 1 is used by user processes executing on the Yalnix system. Region 0 will be managed by your kernel so that it is shared by all processes. Region 1 will be used to map the currently

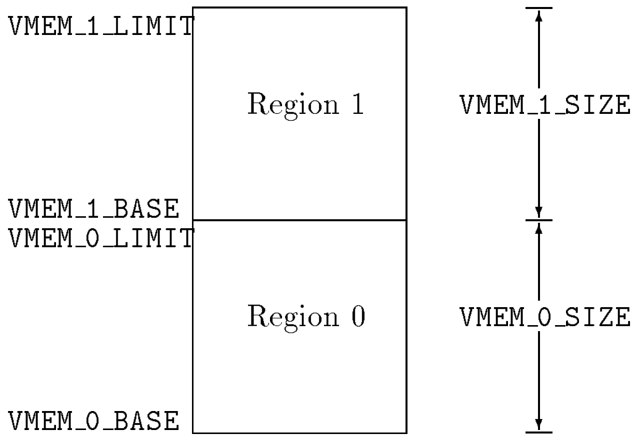

Figure 2: Virtual Memory Layout

running user process and will be changed by the OS whenever a context switch occurs. The hardware provides these two regions so that it can protect kernel data structures from illegal tampering by user applications. When the CPU is executing in priviledged mode, references to both regions are allowed. However, when the CPU is not executing in priviledged mode, references to region 0 are forbidden. The hardware does all of this automatically without any intervention from your kernel.

The virtual address space and its regions are defined by a number of constants located in the `hardware.h` file.

- `VMEM_0_BASE`: The base address of virtual memory region 0.
- `VMEM_0_SIZE`: The size of virtual memory region 0.
- `VMEM_0_LIMIT`: The lowest address not part of virtual memory region 0.
- `VMEM_1_BASE`: The base address of virtual memory region 1.
- `VMEM_1_SIZE`: The size of virtual memory region 1.
- `VMEM_1_LIMIT`: The lowest address not part of virtual memory region 1.

Virtual addresses greater than or equal to `VMEM_0_BASE` and less than `VMEM_0_LIMIT` are in region 0, and virtual addresses greater than or equal to `VMEM_1_BASE` and less than `VMEM_1_LIMIT` are in region 1. All other virtual addresses are illegal. Graphically, the structure of the virtual address space is shown in Figure 2.

Like physical memory, the virtual memory regions are divided into pages, each of size `PAGESIZE`. These pages are accessed by numbers known as *virtual page numbers*. The size of the virtual address space is not necessarily the same as physical memory. In fact, in the current implementation, each of the virtual memory regions is large enough to contain all of the available physical memory.

It is also important to note that the division of virtual memory into regions has nothing to do with the organization of the physical memory nor are the regions considered to be "segments." Thus, you are not going to be dividing the physical memory into separate contiguous regions like this. In fact, each page of physical memory can be assigned arbitrarily to either virtual memory region as needed (and a physical memory page might even be mapped to multiple locations within the virtual address space in special circumstances).

3.3.3 Address Manipulation

Throughout the implementation of your kernel, you will need to manipulate memory address, virtual page numbers, and physical page frames. Furthermore, you may need to convert between the various representations. To do this, a number of constants and macros are defined in `hardware.h` for your use.

- `PAGESIZE`

  The size of a memory page in both virtual and physical memory.

- `PAGEOFFSET`

  This is a bit mask that can be used to extract the location of a memory address within a page. To use it, simply compute `addr & PAGEOFFSET`.

- `PAGESHIFT`

  This is a constant that contains the number of bits that address must be shifted to the right in order to obtain its page number or the number of bits that a page number must be shifted to the left to obtain its base address. For example, if $vp1$ is a virtual address located in Region 1, ($vp1$ `- VMEM_1_BASE) >> PAGESHIFT` would compute the page number within the region.

- `PAGEMASK`

  This is a bit mask that can be used to extract the base-address of a page given a memory address. Use `addr & PAGEMASK` to get the base address.

- `UP_TO_PAGE(`x`)`

  Rounds the address x up the next highest page boundary. If x is on a page boundary, it returns x.

- `DOWN_TO_PAGE(`x`)`

  Rounds the address x down to the next lowest page boundary. If x is on a page boundary, it returns x.

All of the above constants and macros work perfectly well with both virtual and physical addresses. However, the interpretation of results may vary depending on the region. For example, computations involving addresses in region 0 may differ slightly from those involving addresses in region 1.

3.3.4 Page Tables

Whenever a reference to a virtual address is made, the MMU hardware translates it into a physical memory address. This translation occurs automatically without intervention from the kernel. However, the mapping that specifies the translation is controlled by the Yalnix kernel which is responsible for filling in the values of a single-level direct mapped page table for **each** of the virtual memory regions.

A page table is created by allocating a contiguous array of page table entries that define the virtual memory mapping for every single page in a virtual memory region. Internally, the kernel is allowed to put the page table wherever it wants as long as it tells the MMU where to find it. This is done by writing to the following priviledged registers.

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

The MMU hardware never writes these registers: only the kernel can change the virtual-to-physical address mapping.

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

For example, if the reference is to region 0 and the first virtual page number of region 0 is $vp0$, the MMU looks at the page table entry that is $vpn - vp0$ page table entries above the address in `REG_PTBR0`. Likewise, if the reference is to region 1 and the first virtual page number of region 1 is $vp1$, the MMU looks at the page table entry that is $vpn - vp1$ page table entries above the address in `REG_PTBR1`. This process is illustrated in Figure 3.

This lookup to translate vpn to its currently mapped pfn is carried out wholly in hardware. You will notice that in carrying out this lookup, the hardware needs to manipulate page table entries in order to index into the page table (size of the entry needed) and also to

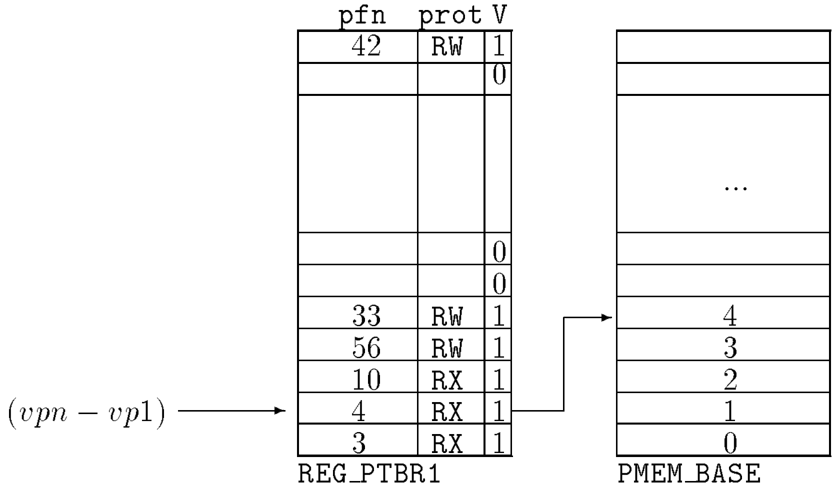

Figure 3: Page Table Lookup for Region 1

extract the pfn from the entry (format of each entry needed.) In manipulating page table entries, then, the hardware has to know the format of the entries.

The format of the page table entries is in fact dictated by the hardware. The file `hardware.h` contains the definition of a C data structure (`struct pte`) that has the same memory layout. You can use that C structure to manipulate page tables from your C programs.

A page table entry is 32-bits wide, and contains the following fields:

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

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

The registers `REG_PTBR0` and `REG_PTBR1` contain *virtual* addresses. In a real operating system, these registers would contain *physical* addresses, but in order to simplify your programming task, the MMU has been designed to access the page tables by virtual addresses located in region 0.

Finally, it is important to realize that your kernel will have to manipulate a variety of different page tables as it runs. In particular, **each** user process will have its own page table that is mapped to region 1 of the virtual memory space. Whenever the kernel performs context switch to another program, it will need to change the region 1 page table by writing a new value to the `REG_PTBR1` register. In addition, your kernel will need to manage a region 0 page table for its own memory needs.

3.3.5 The Translation Lookaside Buffer (TLB)

Most architectures contain a TLB to speed up address translation. The TLB caches address translations so that subsequent references to the same virtual page do not have to retrieve the corresponding page table entry anew. This is important because it can take several memory accesses to retrieve a page table entry[1].

Our machine also contains a TLB. The TLB is managed by the hardware MMU, so you do not need to worry about caching page table entries yourself. Whenever a program accesses a virtual page, the mapping of that page to a physical frame number is automatically stored in the TLB. However, at times, your kernel will need to flush all or part of the TLB. In particular, after changing a page table entry in memory, the TLB will contain a stale mapping for the virtual addresses corresponding to the modified page table entry. Also, when carrying out a context switch from one process to another, the MMU will continue to map the cached entries of the extant process; unless the TLB is flushed, this will cause virtual addresses in the new address space to be mapped to the physical pages of the old address space.

The hardware provides the `REG_TLB_FLUSH` privileged machine register 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 the entire TLB.
- `TLB_FLUSH_0`: Flush all mappings for virtual addresses in region 0 from the TLB.
- `TLB_FLUSH_1`: Flush all mappings for virtual addresses in region 1 from the TLB.
- *addr*: Flush only the mapping for virtual address *addr* from the TLB, if present.

The symbolic constants above are all defined in `hardware.h`.

3.3.6 Initializing Virtual Memory

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

[1] In the case of multi-level page tables and inverse page tables, it might take many memory accesses; in the case of a demand-paged memory system, it might even take a disk access to retrieve a page table entry.

At this point, virtual memory cannot yet be active, since it is the kernel's responsibility to initialize the location and values of the page tables. Virtual memory cannot be correctly used until page tables have been built and the memory trap vector has been set. In order to make it possible to initialize virtual memory, the hardware initially begins execution of the kernel with virtual memory disabled. The boot ROM loads the kernel into the beginning of *physical* memory, and the machine uses only *physical* addresses when accessing memory until virtual memory is enabled by the kernel. When ready, the kernel enables virtual memory translation by executing:

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

After execution of this instruction, *all* values used to address memory are interpreted *only* as *virtual* memory addresses by the MMU. Virtual memory cannot be disabled after this.

Further details about initializing your kernel's memory can be found in the Section 4.2.

3.4 The Serial Port

The hardware features a serial port that would ordinarily be used to communicate with a personal computer. However, for the purposes of debugging and development, you are going to use the serial port as a debugging console where you will be able to launch programs using a simple Unix-like shell and interact with the hardware like it was a text-based terminal.

When a line of input is received on the serial device, the hardware generates a `TRAP_SERIAL_RECEIVE` interrupt. Once received by your kernel, the following BIOS function should be used to read the data from the device:

- `int SerialReceive(char *buf, int len)`

  This copies at most `len` bytes from the serial port input buffer into the buffer pointed to by `buf`. The function returns the number of bytes that were actually read. The constant `SERIAL_MAX_LINE` specifies the maximum number of bytes that the hardware can receive before generating the `TRAP_SERIAL_RECEIVE` interrupt. To simplify programming, you can always specify `len` to be `SERIAL_MAX_LINE`.

To send data to the serial port, you can use the following BIOS function:

- `void SerialTransmit(char *buf, int len)`

  This initiates a transfer of `len` bytes located at location `buf` to the serial port. `len` must be less than `SERIAL_MAX_LINE` and `buf` must point to memory that is located in the kernel's memory address space. A `TRAP_SERIAL_TRANSMIT` interrupt will be generated when the transmission of data has been completed. It is illegal to make another call to `SerialTransmit()` before any previous call has completed its transmission (doing so will generate a hardware error).

It is important to realize that proper use of the serial device requires you to fully understand the interaction between these BIOS functions and the related traps. In particular, it is illegal to call `SerialReceive()` unless a `TRAP_SERIAL_RECEIVE` interrupt has been received. Similarly, it is illegal to make a repeated call to `SerialTransmit()` until the `TRAP_SERIAL_TRANSMIT` interrupt has been received–indicating successful competion of the last operation.

3.5 Flash Storage Device

The flash storage device is used to provide persistent data storage. Although this device is obstensibly a memory card, the hardware treats as a block device that behaves a lot like a disk drive. The following BIOS function is used to transfer blocks of memory to and from the flash device:

- `void FlashTransfer(int op, int blk, void *buffer);`

  Initiates a data transfer to or from the flash device. If `op` is `FLASH_READ`, data is transferred from the flash device to kernel memory. If `op` is `FLASH_WRITE`, data is transferred from kernel memory to the flash memory. `blk` specifies the block number on the device. `buffer` is a pointer to a kernel memory buffer that must be large enough to accomodate an entire block of data.

When a transfer operation is performed, the flash device remains busy until it signals competion of the last operation with a `TRAP_FLASH_ACCESS` interrupt. It is illegal to make another call to `FlashTransfer` before the `TRAP_FLASH_ACCESS` interupt has been received (doing so will generate an unrecoverable hardware fault). Also, since the flash I/O controller uses DMA to perform the transfer, the buffer that is passed to `FlashTranfer()` must remain allocated until the I/O operation has signaled its completion.

Closely related to the flash device are two special purpose registers. The `REG_FLBLKSZ` register contains the block size being used on the device. The `REG_FLNBLK` register contains the total number of blocks on the device. Your kernel will need to read these registers to properly configure its block I/O driver. See section 5.7.5 for more details.

3.6 The Display Device

The hardware features a 256x256 pixel LCD display controller. This device is controlled by writing a display code to the `REG_DISPCTL` register. Each display code is a 32 bit integer that is encoded as follows:

Bits 24-31	Bits 16-23	Bits 8-15	Bits 0-7
opcode	argument	x coordinate	y coordinate

The different display opcodes are as follows:

`DISPLAY_POINT`	Set a point. arg(1) = black, arg(0) = white
`DISPLAY_LINE`	Draw line from last drawn point.
`DISPLAY_CLEAR`	Clear the display.
`DISPLAY_CHAR`	Draw a character. arg = ASCII code.
`DISPLAY_BOX`	Draw solid box from last drawn point.

Your kernel does not have to perform any special decoding of these operations. However, user programs that want to issue raw commands to the display device will do so by writing data to a special display device that you will have to implement (see Section 5.7.1).

3.7 The Pen Device

The hardware also features a touch screen that can be used to detect the presence of a pen. Whenever the pen is activated, the hardware generates a `TRAP_PEN_EVENT` interrupt. When generated, your kernel should read the status of the pen device by reading from the `REG_PENEVT` register. The return value of this register is formatted as follows:

Bits 24-31	Bits 16-23	Bits 8-15	Bits 0-7
event	unused	x coordinate	y coordinate

As with the display device, the event field specifies the type of event that has occured. It is one of the following values

`PEN_PRESS`	The pen has touched the display
`PEN_RELEASE`	The pen has been lifted from the display
`PEN_MOTION`	The pen has moved.

Your kernel will not have to interpret the meaning of the `REG_PENEVT` register itself. However, it will have to queue events and pass them to on to user processes that request to read from the Pen device (see section 5.7.2).

3.8 Bootstrapping Entry Point

On boot, the hardware starts executing the code in the *Boot ROM.* This firmware knows just enough to know where your kernel is located on a storage device and how to load it into the bottom physical memory. Once loaded, the ROM jumps to the routine:

```
void
KernelStart(char *cmd_args[], int pmem_size, struct cpu_context_frame *context)
```

This routine is part of your kernel. It must be written by you, and it is the function that carries out all the operations needed to initialize the virtual memory system as well as any of the devices. It is described further in section 6.2.

3.9 Miscellaneous Hardware Operations

Normally, the CPU continues to execute instructions even when there is no useful work available for it to do. In a real operating system on real hardware, this is all right since there is nothing else for the CPU to do. Operating systems usually provide an *idle process* which is executed in this situation, and which is typically an empty infinite loop. However, in order to be nice to other users on the class machine, your idle process should not loop in this way. Our hardware provides the instruction:

- `void Pause(void)`

  Stop the emulated CPU until the next trap occurs.

The hardware also provides another instruction to stop the CPU.

- `void Halt(void)` Completely halts the CPU and does not begin execution until rebooted (i.e., until the Yalnix process is started again from the Unix command line). You should use this instruction to end emulation and exit from your kernel.

4 Process Memory Management and CPU Context

When a user program is running on the hardware, its state of execution is encapsulated within a *process*. Associated with each process is a page table defining its memory use and a CPU context frame that contains the state of various CPU registers. In addition to managing the resources assigned to processes, the kernel must manage memory and resources for its own use. This section provides an overview of how the kernel is supposed to manage memory and the CPU context for both user processes and itself.

4.1 User Processes

As previously mentioned, region 1 of the virtual address space is reserved for user processes. As these processes run, they consume memory in two ways. First, a program may dynamically allocate memory on the heap using `malloc()` or `new`. Second, a program may require memory to be allocated on the program stack. The stack is where local variables to functions are stored as your program executes. Thus, if your program was highly recursive or involved a deeply nested collection of function calls, you would expect to see the stack grow significantly in size.

The conventional way to handle both types of memory allocations is to allocate memory from opposite ends of the virtual address space and for the heap and the stack to grow towards each other. In this case, the program and data area of a process would start at `VMEM_1_BASE` and grow upwards. The stack would be placed at the top of the virtual address space `VMEM_1_LIMIT` and grow downwards. Figure 4 illustrates this arrangement.

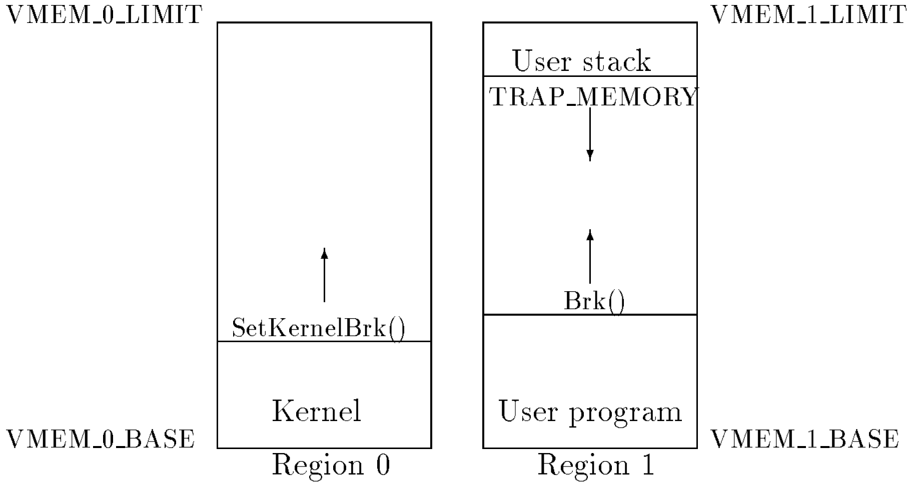

Figure 4: Process and Kernel Memory Management

In general, the area of memory containing the program, data, and heap of a program is called the *program area* and the area containing the stack is called the *stack area*. The limit of the program area (the lowest address not part of the program area) is called the process' *break* and is set from Yalnix user processes with `Brk()` system call. You will need to implement this system call as described in section 5.3.

The library with which your user programs are linked contain versions of `malloc()` and similar procedures that will call your kernel's `Brk()` syscall. Thus, while your kernel must manage user processes' via the `Brk()` syscall, your user programs can call `malloc()`, `realloc()`, etc, as if you were writing regular UNIX programs.

The stack area of a process is automatically grown by your Yalnix kernel in response to a memory exception resulting from an access to a page between the current break and the current stack pointer which has not yet been allocated. When the user process grows the stack beyond its currently allocated stack, the kernel will receive a `TRAP_MEMORY` exception. If the memory address that caused the exception is between the current break for the executing process and below the currently allocated memory for the stack, your kernel should attempt to grow the stack downwards to "cover" this address, if possible. In general, this will involve grabbing as many free frames of physical memory as are necessary. If there is insufficient memory, your kernel will have to kill the user process that caused the exception, but should continue to run all other user processes.

When growing the stack, you should leave *at least one* page unmapped (with the valid bit in its page table zeroed) between the program and stack areas in region 1, so the stack will not silently grow into the program area without triggering a `TRAP_MEMORY`.

4.2 Kernel Memory

In the previous section we explained that at boot time, the hardware loads the kernel into a range of *physical* addresses starting at `PMEM_BASE`. As it turns out, `PMEM_BASE` is exactly the same as `VMEM_0_BASE`. Therefore, when initializing the virtual memory system, you should map all of the physical pages used by the kernel to the exact same set of pages in region 0 by making sure that $vpn == pfn$ in that range of memory addresses at the bottom of region 0.

When making this initial memory mapping, you will need to know where the kernel actually ends in memory. To find out, you can use the following:

- `sbrk(0)` will return the lowest address not in use by the kernel.
- `&_etext` is the lowest address that does not contain executable instructions.

Prior to enabling virtual memory, your kernel has full access to the physical memory. If it needs to allocate memory dynamically, it can grab more memory using `malloc()` as normal. Prior to initialization of the virtual memory system, kernel memory allocations simply increase the kernel's break within the physical address space and consume pages of physical memory at the bottom of the physical address space.

After virtual memory has been enabled, and the kernel is running, it can continue to allocate dynamic memory using `malloc()`. However, unlike the previous paragraph (in which the kernel consumes a contiguous chunk of physical memory), newly allocated kernel memory can be located anywhere within the physical memory space. To handle this type of allocation, you will need to implement a special version of the `Brk()` system call for use by the kernel itself.

In the library with which your kernel is linked by the provided `Makefile`, the familiar procedures such as `malloc()` and `realloc()` have been modified so that they call the procedure `SetKernelBrk()` when they need to augment their heap of available memory. `SetKernelBrk()` must be provided by you. The C function prototype of this procedure is:

- `void SetKernelBrk(caddr_t addr)`

This procedure should grow or shrink the space allocated to your kernel so that all addresses between `PMEM_BASE` and `addr`, and no others, are valid kernel addresses. Therefore, it must allocate and map or unmap pages, as required, in order to make `addr` the new kernel break.

The C type `caddr_t` (C address type) is usually defined by the UNIX system include files, and, for your project, it is defined by including `hardware.h`.

If running on real hardware, your kernel would also have to manage its own stack, but we have eliminated that concern for the sake of simplicity. You may assume that the kernel has no stack in the time-space continuum (a feat of the Yalnix design committee).

4.3 CPU Context

In order to switch execution from one user process to another, you need to be able to save the state of the running process, and restore the state in which some other process found itself at a previous time. Most of this state is active in physical memory, and can be made unavailable or available by changing the page tables. The rest of the state of the running process (program counter, stack pointer, etc) is passed to your kernel when a trap is executed through the trap vector, in the form of an argument to the function in your kernel that receives the trap.

This argument is of type `struct cpu_context_frame`, which is defined in `hardware.h`. The following fields are defined within a cpu context frame:

- `int frame_len`: The length (in bytes) of the entire exception frame. The fields defined here form a common header on the exception frame, and are followed by a variable number of bytes of additional hardware information defining the state at the time of the exception. The `frame_len` field defines the total length (including this header) of the cpu context frame. The maximum length of a cpu context frame is defined in `hardware.h` as `MAX_CPU_CONTEXT_FRAME`.
- `int vector`: The vector number of the particular trap, e.g, `TRAP_ILLEGAL`.
- `int code`: A code value giving more information on the particular trap. Its meaning varies depending on the type of trap. In particular, for `TRAP_KERNEL`, it specifies the type of syscall that produced the current trap (see section 5.3).
- `caddr_t addr`: This field is only meaningful for the `TRAP_MEMORY` exception. It contains the memory address being referenced that caused the exception.
- `caddr_t pc`: The *program counter* value at the time of the trap.
- `caddr_t sp`: The *stack pointer* value at the time of the exception.
- `u_long regs[8]`: The contents of eight general purpose CPU registers at the time of the exception. In particular, for a `TRAP_KERNEL` syscall, these values give the arguments passed by the user process to the syscall and are used to return the result value from the syscall to the user process. This usage is defined further in section 5.3.

In order to switch contexts from one user process to another, you need to save the cpu context of the running process into its process control block, select another process to run, and restore that process' previously stored cpu context frame into the cpu context argument passed by reference to your trap handler. The hardware takes care of extracting and restoring the actual hardware state into and from the trap handler's cpu context argument. When copying a cpu context back and forth, you absolutely *must* copy the number of bytes indicated by its `frame_len` field, and **not** the number of bytes returned by `sizeof(struct cpu_context_frame)`.

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

5 The Yalnix Kernel

So far, the discussion has focused exclusively on the underlying hardware of the system. In this section, we describe the major components that will form the implementation of your Yalnix kernel. These are as follows:

- User processes.
- Process scheduling.
- System calls.
- The I/O subsystem and device drivers.

5.1 User Processes

In your kernel, you will need to implement a process control block (PCB) structure that encapsulates the notion of a "user process." This data structure minimally needs to contain the following:

- An integer process ID that uniquely identifies the process.
- A CPU context (`struct cpu_context_frame`).
- A memory map specifying the virtual to physical memory mapping. For example, you may need to store the region 1 page table for the process.
- A list of file descriptors for all open devices and files being used by the process.
- A process state (e.g., whether or not it is running or not).

In addition to this, you will probably need to create a few general purpose data structures such as lists and priority queues that can be used to contain PCBs. This is because processes spend much of their time waiting for I/O devices, waiting to be scheduled on the CPU, and so forth.

5.2 Process Scheduling

Your kernel should use the `TRAP_CLOCK` interrupt to implement a round-robin process scheduling scheme. Whenever the `TRAP_CLOCK` interrupt is received, your kernel should save the context for the currently running process and switch to the next process on a ready process queue. The only exception to this rule is the case in which only a single process (other than the kernel idle process) is in a runable state. In this case, the currently running process should be allowed to continue (i.e., your kernel should never switch to the idle process if there is useful work to do).

It is important to note that the `TRAP_CLOCK` interrupt handler also has to check for processes that are sleeping due to a `Delay()` system call. For example, your kernel will have to check the sleep queue for processes that need to be awakened and put them back on to the ready queue if their time has expired.

5.3 Yalnix System Calls

5.3.1 How to handle `TRAP_KERNEL`

When a user process wants to call the kernel, it executes a trap instruction that results in a `TRAP_KERNEL` interrupt in the kernel. Upon entry to your kernel, the `code` field of the cpu context frame indicates *which* kernel call is being invoked (as defined with symbolic constants in `yalnixcall.h`.)

To simplify the interface, a library of assembly routines is provided that performs the trap from a user process. This library provides a standard C procedure call interface for the syscalls described shortly. The arguments supplied to each of the library calls are available in the `regs` fields of the cpu context frame received by your `TRAP_KERNEL` handler. Each argument passed to the library procedure is available in a separate `regs` register, in the order passed to the procedure, beginning with `regs[0]`.

Each syscall returns a single integer value, which becomes the return value from the C library procedure call interface for the syscall. When returning from a `TRAP_KERNEL`, the value to be returned should be placed in the `regs[0]` register by your kernel. If any error is encountered for a syscall, one of the error codes of the form `ERROR_`***type*** defined in `yalnix.h` should be returned in `regs[0]`.

The system calls that you will be implementing fall into three general categories:

- Process management.
- Memory management.
- I/O.

Each of these are now described.

5.3.2 Process management

The following syscalls pertain to the creation, destruction, and management of user processes.

- `int Fork(void)`

  Create a new child process as a copy of the parent (calling) process. On success, in the parent, the new process ID of the child is returned. In the child, the return value should be 0. If there is not enough memory to perform the fork operation, the value `ERROR_NOMEM` should be returned.

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

  Replace the currently running program in the calling process 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 receives these arguments as arguments to its `main` procedure. The *argvec* is formatted the same as any C program's *argv* vector. The last entry in the *argvec* vector must be a `NULL` pointer to indicate the end of the list. By convention, `argvec[0]` is the name of the program to be run, but this is controlled entirely by the calling program. The first argument to the new program's `main` procedure is the number of arguments passed in this vector. On success, there is no return from this call in the calling program, but, rather, the new program begins executing at its entry point, and its conventional `main(argc, argv)` routine is called. If the call fails, one of the following error codes should be returned (depending on the nature of the failure):

  - `ERROR_NOMEM`. The new process requires more memory than is available.
  - `ERROR_NOEXEC`. The filename specified is not an executable file in the right format.
  - `ERROR_NOENT`. The filename specified does not exist.
  - `ERROR_FAULT`. One of the arguments points to an illegal address.

  We will provide a template of the `LoadProgram` procedure needed to load an executable program from disk. This procedure reads a program image from an executable file (the file must have been compiled as indicated in the provided `Makefile`,) allocates memory for the new process, and creates its initial stack with the arguments given. This template can be found in `/yalnix/lib/load.template`.

- `int Exit(int` ***status*** `)`

  Terminate execution of the current process and save the integer *status* for possible later collection by the parent process on a call to `Wait`. All resources used by the calling process are freed, except for the saved *status*. This system call should always succeed so the return value is meaningless. When a process exits or is aborted, its children should continue to run normally event though they will no longer have a parent. When the orphans exit at a later time, you need not save or report their exit status since there is no longer anybody around to care.

- `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`, it is added to a FIFO queue of child processes not yet collected by its specific parent. After the `Wait` call, the child process is removed from this queue. If none of the child processes have called `Exit()`, the calling process is blocked until its next child calls `Exit`. The process ID of the child process is returned on success, and its exit status is copied to the integer referenced by the *status_ptr* argument. If the calling process has no children, this function should return the value `ERROR_CHILD`. If `status_ptr` points to an invalid memory address, `ERROR_FAULT` should be returned.

- `int GetPid(void)`

  Returns the process ID of the calling process.

- `int Delay(int clock_ticks)`

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

5.3.3 Memory management

- `int Brk(caddr_t addr)`

  Allocate memory to the calling process' program area, enlarging or shrinking the program's heap storage so that the value *addr* is the new break value of the calling process. The break value of a process is the address immediately above the last address used for its program instructions and data. This call has the effect of allocating or deallocating enough memory to cover only up to the specified address, rounded up to an integer multiple of `PAGESIZE`. The value 0 is returned on success. On failure, the function should return `ERROR_NOMEM` to indicate insufficient memory.

5.3.4 Device and File I/O

All of the I/O devices of the hardware (display, pen, serial port, and flash card) are controlled through a single I/O interface consisting of the following functions.

- `int Open(char *name, int flags)`

  Open a device or file and return an integer file descriptor. The *name* argument is used to specify both the device of interest and an optional file on that device. Your kernel must be able to recognize the following device names:

  - `/dev/serial`. The serial device.
  - `/dev/display`. The display device.

- `/dev/pen`. The pen input device.
- `/dev/flash/`***`filename`***. A file on the compact flash storage device.

The flags argument specifies a mode in which the file or device should be opened. It is the bitwise-OR of the following values.

- `IO_RDONLY`. Open *name* for reading only.
- `IO_WRONLY`. Open *name* for writing only.
- `IO_RDWR`. Open *name* for both reading and writing.
- `IO_CREAT`. If *name* doesn't exist, create it.
- `IO_NONBLOCK`. Specify non-blocking I/O.

If *name* can't be opened or the value of *flags* is inappropriate, one of the following error messages should returned:

- `ERROR_FAULT`. *name* points to an illegal address.
- `ERROR_NOENT`. *name* doesn't refer to known device or file.
- `ERROR_MFILE`. The maximum number of allowable open files are in use by the process.
- `ERROR_NFILE`. The maximum number of allowable open files on the entire system has been reached.
- `ERROR_INVAL`. The value of *flags* is invalid.
- `ERROR_NOMEM`. There isn't enough memory available to open *name*.

- `int Write(int `***`fd`***`, caddr_t `***`buf`***`, int `***`len`***`)`

  Write the contents of the buffer referenced by *buf* to the I/O device or file referenced by the file descriptor *fd*. The length of the buffer in bytes is given by *len*. By default, the calling process is blocked until all bytes from the buffer have been written. In the case of non-blocking I/O, the function should either initiate the write operation immediately or return `ERROR_AGAIN` to indicate that the operation would block. The return value should be *len* if the `Write` is successful. Otherwise, one of the following error codes should be returned:

  - `ERROR_FAULT`. *buf* refers to an illegal address.
  - `ERROR_BADF`. *fd* is a bad file descriptor.
  - `ERROR_INVAL`. *len* is invalid (a negative number perhaps).
  - `ERROR_NOSPACE`. No space left on device.
  - `ERROR_FBIG`. Maximum file size exceeded.
  - `ERROR_AGAIN`. Operation would block (non-blocking I/O).

- ERROR_MODE. File mode does not support this operation.
- ERROR_NOSUPP. Operation not supported by the device.

- `int Read(int fd, caddr_t buf, int len)`

  Read data from the I/O device or file referenced by the file descriptor *fd* into the buffer *buf*. The maximum amount of data to read (in bytes) is given by *len*. By default, the calling process is blocked until all bytes from the buffer have been written. However, if nonblocking I/O was specified using the `IO_NONBLOCK` flag to `Open` the function should return immediately even if no data was available (it should return `ERROR_AGAIN` in this case). The return value should be the number of bytes actually read on success. Otherwise, one of the following error codes should be returned:

  - `ERROR_FAULT`. *buf* refers to an illegal address.
  - `ERROR_BADF`. *fd* is a bad file descriptor.
  - `ERROR_INVAL`. *len* is invalid (a negative number perhaps).
  - `ERROR_EOF`. The end of the file was reached or no more input is available.
  - `ERROR_AGAIN`. Operation would block (non-blocking I/O).
  - `ERROR_MODE`. File mode does not support this operation.
  - `ERROR_NOSUPP`. Operation not supported by the device.

- `int Close(int fd)`

  Close the device or file referenced by *fd*. This should remove the file descriptor from the list of open files for the calling process. In addition, if the calling process is the only process currenting using the device or file, the kernel should remove the device from its internal list of open files. If *fd* does not correspond to a valid file descriptor, `ERROR_BADF` should be returned.

- `int Unlink(char *name)`

  Removes the file *name*. For this project, this operation only applies to files located on the flash device. When a file is removed, it's directory entry should be destroyed and all of the blocks forming the contents of the file placed back on to the free block list. Returns 0 on success, `ERROR_NOENT` is *name* doesn't exist, `ERROR_FAULT` if *name* is an illegal memory address, or `ERROR_NOSUPP` if the operation isn't supported.

- `int Dup2(int fd1, int fd2)`

  This duplicates the file descriptor *fd1* so that *fd2* refers to the same file or device as *fd1*. If the value of *fd2* already corresponds to an open device, it should be closed first. It is important to stress that all this function does is make the file descriptor *fd2* refer to exactly the same open file or device as *fd1*. Internally, the kernel should just update a few pointers and increase the reference count on *fd1*. Returns 0 on success. Returns `ERROR_BADF` if *fd1* doesn't refer to an open device. Returns `ERROR_INVAL` if *fd2* is a bad file descriptor value.

- `int Pipe(int fds[2])`

  This call creates an I/O mechanism called a pipe and returns two file descriptors in the array *fds*. A "pipe" represents a unidirectional flow of data where data written to the input end of the pipe (*fds[0]*) can be read from the output end of the pipe (*fds[1]*). This can be used to implement a simple form of interprocess communication. On success, this function creates two file descriptors, places them in *fds*, and returns 0. Otherwise, one of the following error codes is returned:

  - `ERROR_MFILE`. The maximum number of open files for process.
  - `ERROR_NFILE`. The system-wide open file table is full.

  (Note: This function differs from its Unix counterpart in that the pipe that is created is unidirectional).

- `int Seek(int fd, int offset, int whence)`

  This call moves the current file pointer to a new position. If *whence* is `SEEK_SET`, *offset* specifies a position from the beginning of the file. If *whence* is `SEEK_CUR`, *offset* specifies a position relative to the current file pointer. If *whence* is `SEEK_END`, *offset* specifies a position relative to the end of the file. On success, the value of the file pointer is returned as the number of bytes from the beginning of the file. On failure, the function should return one of the following error codes:

  - `ERROR_BADF`. Bad file descriptor passed in *fd*.
  - `ERROR_INVAL`. Bad value specified for *whence* or the specified *offset* does not represent a valid position within the file.
  - `ERROR_NOSUPP`. Operation not supported.

- `int Fcntl(int fd, int cmd, int arg)`

  This system call is used to specify device specific processing on an open device or file. The interpretation of *cmd* and *arg* depends on the device referred to by *fd*. However, typical values are the same as those passed to the *flags* argument of the `Open()` call. Returns one of the following error codes on failure:

  - `ERROR_BADF`. Bad file description passed in *fd*.
  - `ERROR_INVAL`. *cmd* or *arg* is a bad value.

- `int Sync()`

  This call synchronizes all of the I/O devices by writing unsaved data. This system call would typically be called prior to system shutdown. This primarily applies only to the flash memory device. If your kernel caches any information such as the free block list, inodes, or the device directory, the `Sync()` system call should write all of this information back to the device.

5.3.5 Argument checking

Your kernel should **verify all arguments passed to a syscall**, and should return an appropriate error if any arguments are invalid. In particular, you need to verify that a pointer is valid before you use it. This means you need to look in the page table in your kernel to make sure that the entire area (such as a pointer and a specified length) are readable and/or writable (as appropriate) before your kernel actually tries to read or write there. For C-style character strings (null-terminated) you will need to check the pointer to each byte as you go (C strings like this are passed to `Exec`.) 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 check access to each byte, checking each until it found the '\0' at the end. Insert calls to these two routines as needed at the top of each syscall to verify the pointer arguments before you use them.

Such checking of arguments is important for two reasons: security and reliability. An unchecked `Read`, 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` which would never be serviced because the kernel is not interruptible.

5.4 The I/O Interface

In the previous section, the only system calls pertaining to device I/O were versions of the standard `Open()`, `Read()`, `Write()`, `Close()`, `Unlink()`, `Seek()`, and `Fcntl()` system calls. For the most part, all of these functions operate on integer file descriptors as returned by the `Open()` system call.

5.4.1 File descriptors

One of the primary jobs of your kernel will be to provide the mapping between integer file descriptors and a corresponding I/O device. To do this, your kernel needs to manage a series of file descriptor tables. First off, the integer file descriptor value that is returned to a user process should represent an index into a file descriptor array that is attached to each active process. This array entry, in turn, should refer to an entry in some kind of system-wide open file table containing information about each open device or file. This process is illustrated in Figure 5. The constants `MAX_USER_FILE` and `MAX_KERNEL_FILE` (found in `yalnix.h`) define the sizes of the process and kernel file tables respectively.

A key feature of this arrangement is that an open file or device may, in fact, be shared by multiple processes. Therefore, information about open files is typically not part of a process control block, but rather collected in some kind of global kernel table. Meanwhile, the file descriptor table for each process is really nothing more than a list of integers or pointers that refer to an entry in this global table.

The entries in the open file table should minimally be pointers to some kind of open I/O

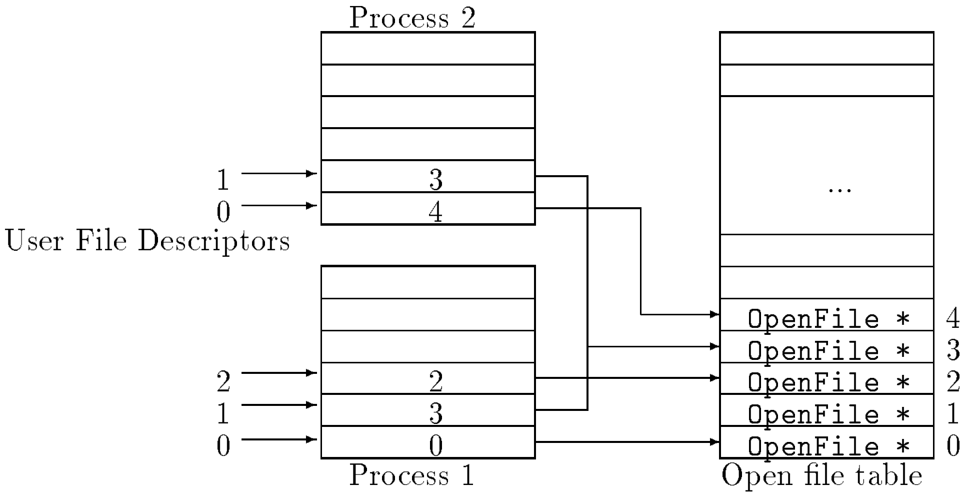

Figure 5: File descriptors

device structure (`OpenFile *`). The exact format of this structure depends on the low-level device as well as the details of your implementation (this is a structure that *you* have to define in your kernel). However, there are a few common fields you should probably have.

First, your device structure will need to have a reference count. Since multiple processes can refer to the same open file (for example, a forked process shares the same set of open files as the parent), the reference count will be increased whenever a new file descriptor for that file is created. Similarly, whenever the user calls the `Close()` system call, the reference count should be decreased. When the reference count of an open file reaches zero, the kernel should close the file or device and remove it from the system-wide open file table.

The second thing that you will probably need is some kind of method table that points to the real implementations of the `Open`, `Read`, `Write`, `Seek`, `Fcntl`, and `Close` methods for a specific file or device. For example, when you execute the `Read()` system call in your kernel, that operation will first perform a lookup in the open file table based on the file descriptor passed to the system call. Then, based on the method table contained within the corresponding `OpenFile` structure, control will be passed to a specific implementation of `Read()` that knows how to actually perform the operation.

5.4.2 File descriptors and system calls

In addition to creating a file descriptor table for each process and a table of open files for the kernel, a number of system calls involve interactions with the I/O system.

- `Fork()`. When a process executes the `Fork()` syscall, all of the file descriptors in the parent should be duplicated in the child. These duplicated descriptors should refer to the exact same set of `OpenFile` structures as the parent. However, when duplicating the descriptor table, make sure you remember to increment the reference count of each open file. Otherwise, when either the child or parent closes one of its files, it will destroy the kernel `OpenFile` structure–leaving the other process with a stale pointer.

- `Exec()`. All of the files that were open in the process before the `Exec()` should remain open in the newly loaded program (this is how makes I/O redirection and pipes work). In reality, a kernel tends to close many of files that were open in the original process, but for the sake of simplicity, you will not need to do this.

- `Close()`. The `Close()` syscall should close a file by decrementing the reference count of the corresponding `OpenFile` structure in the kernel open file table. If this reference count reaches zero, the real `Close()` method for the device or file should be called. Otherwise, nothing happens (meaning that the file is still open elsewhere).

- `Dup2()`. The `Dup2()` syscall duplicates a file descriptor. All this function does is duplicate an entry in the process file descriptor table so that two different file descriptors point to the exact same `OpenFile` structure in the kernel open file table. Of course, this duplication process should also increase the reference count.

- `Exit()`. The `Exit()` syscall should close all of the open files for a process. This should be done immediately–not after the parent process has collected the exit code.

5.5 Blocking vs. Non-blocking I/O

By default, all of the I/O operations should implement blocking I/O. This means that the `Read()` and `Write()` calls should block the caller until the operation has actually been completed. In addition to this, all you should also implement non-blocking I/O. With non-blocking I/O, the corresponding operation first checks the status of the device to see if the requested operation can be immediately completed. That is, the `Read()` function would check to see if data was available and the `Write()` function would check to make sure the device wasn't busy. If so, the system call would execute as normal. Otherwise, both of these functions should immediately return with an error `ERROR_AGAIN` that tells the caller that the operation would have blocked and that they must try again later.

It is important to note that non-blocking I/O is not the same as asynchronous I/O. That is, non-blocking I/O does not mean that the functions return to the caller while the requested I/O operation is handled behind the scenes by the kernel. Instead, it just means that the kernel performs a simple yes/no check to see if the operation would block or not. If so, `ERROR_AGAIN` is returned. If not, the system call executes normally.

To specify non-blocking I/O, a process can either open a file with the `IO_NONBLOCK` flag set like this:

- `Open("/dev/serial", IO_RDONLY | IO_NONBLOCK);`

Or, it can enable the mode later using `Fcntl()` like this:

- `Fcntl(fd, IO_NONBLOCK, 1);`

5.6 Putting the pieces together

To implement the high-level Yalnix I/O system, you will need to do the following:

- Create a general purpose data structure (`OpenFile`) that contains a reference count, a device method table, and any device specific values that might be appropriate (such as the file pointer for an open file).
- Create an open-file table that contains all of the currently open files being manipulated by the kernel. The table should refer to `OpenFile` structures.
- Modify the PCB to contain a small file descriptor table that contains references to entries in the kernel open file table.
- Implement the `Open()`, `Read()`, `Write()`, `Close()`, `Unlink()`, and `Seek()` system calls so that they take a user file descriptor, perform a lookup in the kernel open file table, and pass control to a device-specific implementation of the desired system call. Note, the implementation of `Open()` and `Close()` will also involve some bookkeeping in the I/O system (in order to update the kernel file table).
- Modify the `Fork()`, `Exec()`, and `Exit()` system calls to properly duplicate or close a process' open file descriptors.
- Implement the `Dup2()` system call.
- Implement the `Pipe()` system call.

Further details about how to build the I/O system will be passed out in due course. However, one thing is certain—by the time you finish the implementation, you will have become a grand master of file descriptors.

5.7 Device Drivers

To complete the I/O system, you will need to write device drivers for all of the hardware devices (the display, the pen input, the serial port, and the flash storage card). The procedure for doing this is actually pretty simple—all you need to do is provide implementations of the `Open()`, `Read()`, `Write()`, `Close()`, `Seek()`, `Unlink()` methods for each device. Even this isn't as bad as it sounds because many of the hardware devices only need to provide some of these methods (for example, the display device only needs to support `Open()`, `Write()`, and `Close()`).

The remainder of this section describes each of the devices you will need to support in increasing order of implementation difficulty.

5.7.1 The LCD Display

Required operations: Open(), Write(), Close()

The LCD display device is accessed from user programs by opening the special file `"/dev/display"`. Since the device is write-only, the only accepted file I/O mode is `IO_WRONLY`. If any other file I/O mode is given to `Open()`, `ERROR_INVAL` should be returned to the user.

Once open, user programs control the display device using the `Write()` system call to send the display controller raw hardware operations as described in 3.6. Thus, all your device driver needs to do is read the contents of the user supplied buffer as a sequence of integer display codes and write them to the display controller register `REG_DISPCTL`.

The `Close()` system call does not need to do anything special to the display controller. Thus, all it has to do is free the corresponding `OpenFile` structure. There is no need to implement any of the other I/O system calls for the display device.

Since the display device accepts commands in the form in integers, your `Write()` function should check the word alignment and length of the user buffer. Otherwise, an errant user parameter could cause your kernel to fail with a Bus Error.

5.7.2 The Pen Input Device

Required operations: Open(), Read(), Fcntl(), Close()

The Pen input device is accessed from user programs by opening the special file `"/dev/pen"`. Since the device is read-only, the only accepted file I/O mode is `IO_RDONLY`. If any other file I/O mode is given to `Open()`, `ERROR_INVAL` should be returned to the user.

Once open, user programs can read events from the pen using the `Read()` system call. However, the implementation of this takes a little care.

Whenever the pen is used, the kernel receives notice in the form of a `TRAP_PEN_EVENT` interrupt. When this occurs, your kernel should first check to see if the pen device has been opened by any process. If so, the contents of the `REG_PENEVT` register should be read and the results saved in a kernel allocated event buffer. If no processes have opened the device, the events should simply be ignored (discarded). In addition to this, your kernel should place a limit on the size of its event buffer in case a process opens the device, but doesn't call `Read()` to receive the events. If the event buffer fills up, your kernel should start discarding incoming events.

When a user process calls `Read()`, the device driver should return all of the available pen events in the user supplied buffer (up to the maximum buffer length supplied by the user). If no pending events are available when `Read()` is called, there are two possible courses of action. First, if the user opened the pen device with the `IO_NONBLOCK` flag and no incoming data is available, `Read()` should return immediately and return the value `ERROR_AGAIN`. Otherwise, the `Read()` call should block and wait for the next `TRAP_PEN_EVENT` to occur.

If two user processes simultaneously attempt to read from the pen device, the resulting program behavior is unspecified (and is likely to be very bizarre from the standpoint of the user). Your kernel doesn't need to take any steps to sort this out. In particular, it should just return data to processes in the same order as the `Read()` calls are received.

The `Fcntl()` operation for the pen device only needs to recognize `IO_NONBLOCK`.

5.7.3 The Serial Port

Required operations: Open(), Read(), Write(), Fcntl(), Close()

The serial port is accessed from user programs by opening the special file `"/dev/serial"`. The device is bi-directional and can be opened for both reading and writing. Thus, your driver should support the `IO_RDONLY`, `IO_WRONLY`, and `IO_RDWR` open modes.

When incoming data is received on the serial port, a `TRAP_SERIAL_RECEIVE` interrupt is sent to your kernel. If a process has previously called `Read()` and is waiting for data, your kernel should read data from the device using `SerialReceive()`, hand it to the user process, and put the process back on to the ready queue. If no processes are waiting to read data, the kernel should save the input data and put it on some kind of input queue.

The `Read()` system call should operate by first checking the input data queue of the serial device to see if any data has been received, but not yet read. If so, the first line of data on the queue should be copied to the user process and control returned to the caller. If no input data is available, the calling process should be put on a queue of processes that are waiting to receive input. In the meantime, the kernel should context switch to another process while waiting for the input to arrive.

Implementation of the `Write()` system call is a little more involved. Specifically, you must address the following:

- If the serial port is inactive, the kernel should send up to `SERIAL_MAX_LINE` bytes of data to the device using `SerialTransmit()` and put the calling process on a transmit queue where it will sit until a `TRAP_SERIAL_TRANSMIT` interrupt occurs (indicating completion of the operation).
- If the serial port is busy due to a previous operation in progress, the kernel should simply put the calling process on the transmit queue and context switch to another job.
- When the `TRAP_SERIAL_TRANSMIT` interrupt is received, your driver should pull the next process off of the transmit queue to see if there is any more data that needs to be sent to the device. If so, another `SerialTransmit()` operation should be issued and the process put back on to the transmit queue. Otherwise, the process should just be added to the kernel ready queue and its return value set to indicate successful completion. A process that has called `Write()` should not be put back on the ready queue until a matching `TRAP_SERIAL_TRANSMIT` has been received.

By default, `Read()` and `Write()` operations block until data has been received or transmission has been completed. However, if the serial device was opened with the `IO_NONBLOCK` option to the `Open()` system call, the `Read()` and `Write()` system calls should return the value `ERROR_AGAIN` to indicate that the operation would have blocked. For non-blocking writes, the kernel should only write up to `SERIAL_MAX_LINE` bytes of data and return to the caller as soon as possible.

5.7.4 Pipes

Required operations: Pipe(), Read(), Write(), Fcntl(), Close()

Although not technically a "device", your implementation of the `Pipe()` system call effectively works like a device driver.

When a call to the `Pipe()` system call is made, your kernel needs to create a special I/O object known as a "pipe". Internally, a pipe consists of the following information:

- A queue of processes waiting to write to the pipe.
- A queue of processes waiting to read from the pipe.
- Optionally, a buffer used to store temporary data (although this isn't needed if you do it right).

In addition, the `Pipe()` system call needs to create **two** `OpenFile` objects (along with two distinct file descriptors). One of these is used to write information into the pipe. The other one is used to read data from the pipe.

As for the behavior of a pipe, here is how it should work:

- When a `Read()` operation is made on a pipe, the kernel should check to see if any processes are on the pipe's write queue. If so, your kernel should transfer as many bytes of data as possible between the writer and reader (up to the maximum size of the reader's buffer). It should then return control immediately back to the reader. If the process on the write queue was able to write all of its data, it should be put back on the ready queue as well.

  If no processes are the pipe's write queue, the reading process should be put on the pipe's reader queue and blocked until input is available (i.e., the kernel should context switch to another process).

- When a `Write()` operation is made, the kernel should check to see if any processes are waiting on the pipe's reader queue. If so, your kernel should transfer as many bytes of data are as possible between the writer and reader (up to the maximum size of the reader's buffer). If the writer was able to send all of its data, control should return back to the writer immediately. Otherwise, it should be placed on the writer queue and blocked until it can complete its transfer of data. The reading process should be put back on the ready queue as well.

If no processes are on the pipe's read queue, the writing process should be put on the pipe's writer queue and blocked until a reader shows up.

As with other devices you should implement support non-blocking I/O. To do this with a pipe, a user will have to enable the non-blocking mode using the `Fcntl()` command. For example:

```
int fds[2];
Pipe(fds);
Fcntl(fds[0], IO_NONBLOCK, 1);
```

Once enabled, non-blocking I/O has the usual semantics where `Read()` and `Write()` return the value `ERROR_AGAIN` if they would block.

5.7.5 Flash Storage Device

Required operations: Open(), Read(), Write(), Fcntl(), Seek(), Unlink(), Close(), Sync()

This is by far the most difficult part of the project to implement. Do not start working on this until you have everything else working first.

At the lowest level, the flash storage device is a block I/O device in which blocks of data can be transferred to and from memory using the `FlashTransfer()` function in the BIOS (see section 3.5). However, your device driver is going to manage the device as a very simple file system.

First, a few high level details:

- User programs will access the device by access by manipulating files with names of the form `"/dev/flash/`***`filename`***`"`. The high-level I/O subsystem should recognize this filename pattern and forward requests to the appropriate `Open()` and `Unlink()` calls for the device.
- Your implementation does not need to worry about subdirectories or symbolic links (you can put everything in one big directory).
- Your implementation **does** need to worry about inodes and other elements of the filesystem.

Low-Level Device Characteristics

To initialize the file system, your driver will need to query the device for its blocksize and number of blocks. This can be done by reading from the `REG_FLBLKSZ` and `REG_FLNBLK` registers.

Once you know the device characteristics, you next need to manipulate the free block map. This is a large bit-map indicating the usage of each block on the device. The size of the map (in bytes) can be computed from the register values above. Physically, the free block map is found starting at block 0 of the device and continues for however many blocks are required to hold the entire map. For example, if the device contains 16384 blocks and the block size is 512 bytes, the free block map would occupy the first 4 blocks of the device.

Immediately following the free block map is the inode corresponding to the device file directory. Thus, in the above example, the free block map would occupy blocks 0-3 while the directory inode would be found in block 4. Note: when the flash device is first initialized, it is filled entirely with 0 bytes. Your kernel should take steps to recognize this and initialize itself properly at startup.

The manner in which the rest of the device is organized depends on your implementation.

File Directory

The file directory of the device should be treated as a regular file that can be read using the special name of `"/dev/flash/."`. The contents of this file should be a series of records of type `struct yalnix_dirent` whose definition is found in the file `/yalnix/include/yalnixdir.h`. The structure of this record is as follows:

```
typedef struct yalnix_dirent {
  char   filename[28];
  int    inode;
} yalnix_dirent;
```

where `filename` contains the name of the file and `inode` is the block number of the inode for the file.

Note: the reason for rigidly defining the directory structure is so that user processes (which we will write to test your kernel) will be able to read the list of files from the device in a portable manner.

Inodes

The "inode" that is associated with each file contains information about the length of the file as well as the list of block numbers that make up its data. The format of the inode structure is something that you need to design and implement yourself. The key point is that the device directory provides a mapping of names to inodes. When manipulating files, it will be up to you to decide how to manipulate inodes and file data.

File system operations

Your file system needs to support all of the standard I/O operations:

- `Open()`. Creates a new file on the device. If a file already exists and it is opened as `IO_WRONLY`, the file should be truncated to zero length. Furthermore, a new file is only created if `Open()` is called with the `IO_CREAT` flag.
- `Read()`, `Write()`. These functions behave exactly as you would expect. However, for each open file, you will need to maintain a current file pointer that indicates the

current position within the file. Repeated reads or writes should update the file pointer as needed.

- `Seek()`. This function should be used to move the file pointer to a new position within the file.
- `Unlink()`. This removes a file from the file directory and releases its data blocks.
- `Sync()`. This should write any cached data in the kernel back to the file system.

In addition, your implementation should support non-blocking I/O in exactly the same manner as for the serial device.

Implementation details

In order to improve performance and make data management easier, your kernel should cache copies of the free block list, the device file directory, and any inodes currently in use. In addition, you will probably want to maintain a collection of internal buffers that hold partially read or written blocks during the course of performing `Read()` and `Write()` calls.

To prevent data corruption, your implementation of the `Sync()` system call should make sure that **all** of the kernel's cached file system information is written back to the device before the hardware simulation is terminated.

Finally, you will need to pay careful attention to the device resources. In particular, your kernel should be able to correctly respond if the device runs out of storage space or if the maximum file size is exceeded. You should also make sure that you aren't leaking blocks (for example, if you delete a file from the device, make sure you reclaim all of its space including its inode).

Should you corrupt the data on the flashcard while working on your implementation, you can clear the flashcard by removing the file 'flashcard' in your yalnix directory (the hardware will recreate it the next time you run)

The implementation complexity of the file system is substantial. For instance, in Dave's implementation of the kernel, the block device driver is about 5 times larger (in lines of code) than that of the serial device. Because of this, you will be receiving further handouts to help guide you through this minefield.

6 Implementation hints

6.1 Process Initialization

Process IDs

Each process in the Yalnix system must have a unique integer process ID. Since an integer allows for over 2 billion processes to be created before overflowing its range, you should simply assign sequential numbers to each process created and not worry about the possibility of wrap-around (though real operating systems do worry about this.) The process ID must be a small, unique integer, not a pointer. The data structure used to map a process ID to its process control block should not be grossly inefficient in space nor time. This probably means that you'll end up using a hash table indexed by process ID to store the process control blocks.

Fork

On a `Fork`, a new process ID is assigned, the cpu context from the running parent is copied to the child, and new physical memory is allocated into which to copy the parent's memory space contents. Since the CPU can only access memory by virtual addresses (using the page tables) you must map both the source and the destination of this copy into the virtual address space at the same time. 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.

Exec

On an `Exec`, you must load the new program from the specified file into the program region of the calling process, which will in general also require changing the process' page tables. The program counter in the `pc` field of the cpu context frame must also be initialized to the new program's entry point. A sample C procedure that can be used to do all this is available in `/yalnix/lib/load.template`

On an `Exec`, you must also initialize the stack area for the new program. The stack for a new program starts with only one page of physical memory allocated, which should be mapped to virtual address `VMEM_1_LIMIT - PAGESIZE`. As the process executes, it may require more stack space, which can then be allocated as usual for the normal case of a running program.

To complete initialization of the stack for a new program, the argument list from the `Exec` must first be copied onto the stack. The `load.template` file also does this inside the `LoadProgram` procedure. The SPARC architecture also requires that an additional number of bytes (defined in `hardware.h` as `INITIAL_STACK_FRAME_SIZE` be reserved immediately below the argument list. The stack pointer in the `sp` field of the cpu context frame should then be initialized to the lowest address of this space reserved for the initial stack frame. Like all addresses that you use, this must be a virtual address. Again, all these details are presented in the form of a C procedure in `load.template`

6.2 Kernel Initialization

Your kernel begins execution at the procedure `KernelStart`. As already shown in section 3.8, this function is called from the boot firmware, which expects it to conform to the following prototype:

```
void
KernelStart(char *cmd_args[], int pmem_size, struct cpu_context_frame *context)
```

The procedure arguments above are built by the boot ROM and passed to your `KernelStart` routine at boot time. The `cmd_args` argument is a vector (in the same format as `argv` for normal UNIX `main` programs), containing a pointer to each argument from the boot command line (what you typed at your UNIX terminal.) The `cmd_args` vector is terminated by a `NULL` pointer. The `pmem_size` argument is the size of the physical memory in the machine you are running, as determined by the boot ROM. The size of physical memory is given in units of bytes. Finally, the `context` argument is in the format of a cpu context frame (see section 4.3) although it is built by the boot ROM rather than by the hardware in response to a trap. Your kernel should use this cpu context frame as the basis for other cpu context frames, notably the cpu context frame that starts the initial process at boot time. Processes created from a `Fork`, instead, copy the cpu context frame from the parent process. (Note that in Yalnix, all processes except for the first process started at boot time are created by `Fork`.)

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

- Initialize the interrupt vector entries for each trap, by making them point to the correct subroutines in your kernel.
- Initialize the `REG_VECTOR_BASE` privileged machine register to point to your interrupt vector array.
- Build a structure to keep track of what page frames are free in physical memory. For this purpose, you might be able to use a linked list of physical frames, implemented in the frames themselves. Or you can have a separate structure, which is probably easier, though slightly less efficient. This list of free pages should be based on the `pmem_size` argument passed to your `KernelStart`. The list should of course not include any memory that is already in use by your kernel. The list may or may not include some of the physical frames targetted by the direct hardware translation of addresses in region 0.
- Enable virtual memory

- Create the idle process based on the cpu context passed to your `KernelStart` routine. The idle process is the process that gets scheduled onto the CPU when there is no other process ready. See `Pause` in section 3.9.

- Create the first process and load the initial program into it. The process will serve the role of the *init* process in UNIX[2]. To run your initial program you should put the file name of the *init* program on your shell command line when you start Yalnix. 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 cpu context frame (values which you have presumably modified to point to the initial context of the initial process.

6.3 Plan of Attack

Although the kernel project is being divided into distinct sub-projects, here is a feel for what is involved in the entire implementation effort.

1. "Wrap your brain" around the assignment. Read it carefully and understand first the hardware, then the operations you will need to implement. Understand all the points in the code at which your kernel can be executed, i.e, make a comprehensive list of kernel entry points (hint: the kernel runs in privileged CPU mode, which can only be set by the hardware.)

2. Write high-level pseudo-code for each syscall, interrupt and exception. Then decide on the things you need to put in what data structures (notably in the Process Control Block) to make it all work. Iterate until the pseudo-code and the main prototype data structures mesh well together.

3. Write a simple in-kernel idle "process" that prints a message whenever it runs and the `Pause()`'s, which should be once every second as it gets woken up by the clock interrupt.

4. Take a cursory look at `load.template`. You need not understand all the details, but make sure you understand the comments that are preceded by "`===>>>`".

5. Write the `TRAP_MEMORY` handler.

6. Write an idle program (a single file with a main that loops forever calling `Pause()`.) Write `KernelStart` and `LoadProgram` and use them to get the idle process running. If the idle process runs, you have successfully bootstrapped virtual memory!

[2]The init process in UNIX forks all other processes; in particular, it forks all the `getty` programs that display the login prompt on all terminals, and it forks off all the network daemons.

7. Write an "init" program. The simplest init program would just loop forever. Modify `KernelStart` to start this init program (or one passed in the `yalnix` command line) in addition to the idle program.

8. Make sure your init and idle programs are context switching. For this you'll probably need to implement a handler for `TRAP_CLOCK`.

9. Implement `GetPid` and call it from the init program. At this point your syscall interface is working correctly.

10. Implement the `Brk` syscall and call it from the init program. At this point you have a substantial part of the memory management code working.

11. Implement the `Delay` syscall and call it from the init program. Make sure your idle program then runs for several clock ticks uninterrupted. At this point you have your process queues working smoothly and can context switch in at least two places.

12. Implement the `Fork` syscall. If you get this to work you are almost done with the memory system.

13. Implement the `Exec` syscall. You have already done something similar by writing `LoadProgram`.

14. Write another small program that does not do much (print and delay on a loop, for instance.) Call `Fork` and `Exec` from your init program, to get this third program running. Watch for context switches.

15. Implement and test the `Exit` and `Wait` syscalls.

16. Implement some of the more simple device drivers (display, pen, and serial device) and get the basic I/O system calls to work. Once you've got a few of these work, you will have gotten the hang of it.

17. Implement pipes.

18. Implement the simple file-system for the flash storage device.

19. Implement and test anything else that is remaining. None of this should be difficult unless you are working on it the night before the deadline.

20. Look at your work and wonder in amazement.

21. Pass out.

One final note about the schedule is that time management is absolutely critical to success. It **is** possible to complete the kernel by making slow and steady progress. On the other hand, if you wait until the last minute to start, you may find yourself spending the night in the lab!

7 Your Assignment

To complete this assignment, you need to implement a Yalnix kernel that runs on the hardware defined in this document and handles all of the traps and system calls defined in this document. Specifically, you need to provide:

- A `KernelStart` routine to perform kernel initialization,
- A procedure to handle each defined trap.
- A procedure (called by your `TRAP_KERNEL` handler) to implement each defined Yalnix syscall.
- Implementations of the required I/O functions for each of the I/O devices.

8 Logistics

8.1 Compiling your kernel

The file `/yalnix/sample/Makefile.template` contains a basis for the `Makefile` we recommend you use for this project. The comments within it should be self-explanatory. There is some magic involved in the make macros with names following the pattern `*_LINK_FLAGS` which you may safely ignore but which may not be safely taken out.

The Sun C compiler and tools can be found in `/opt/SUNWspro/bin`. These are the development tools that have been used to test our reference kernel and which have been used to compile the Yalnix hardware emulator. The project has also been known to work with `gcc` and `gdb`.

You may use, `CC` and/or `g++`, at your peril: the hardware emulation has not been extensively tested with C++. Furthermore, most of the high level features of C++ actually tend to make life more difficult since they obscure the underlying details of how the operating system is supposed to operate. Also, unless you fully understand the details of using such features, you may not be able to debug your kernel. Last, but not least, if you choose to use advanced C++ features, there is a very good chance that neither myself or the TAs are going to be willing to help you when you get into trouble because we simply do not have the time to try and debug twenty levels of nested templates or the internals of the C++ library.

If you still feel the need to use C++ after listening to our warning, **you are still required to adhere to the following C++ coding rules**:

- You may **not** use templates nor any part of the STL.
- You may **not** use exceptions.
- You may **not** use namespaces.

- You may **not** use run time type information (RTTI).
- You may **not** use any part of the C++ library (iostreams, strings, etc...).
- You may **not** use operator overloading.
- You may **not** rely upon static construction of objects.
- You may **not** use funny casting operators (`dynamic_cast`, `reinterpret_cast`, etc...).
- You may use classes.
- You may use single inheritance.
- You may use virtual functions.

Violations to the above rules will result in a point deduction.

8.2 Running your kernel

Your kernel will be compiled and linked as an ordinary UNIX executable program, and can thus be run as a command from the 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. Suppose your kernel is in the executable file `yalnix`. Then you run your kernel as:

`yalnix [-t [`***tracefile***`]] [-lk` ***level***`] [-lh` ***level***`] [-s] [`***initfile*** `[`***initargs....***`]]`

For example,

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

The meaning of these switches is as follows:

- `-t`: This turns on "tracing" within the kernel and machine support code, and optionally specifies the name of the file to which the tracing output should be written. To generate traces 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 the `level` argument to `TracePrintf`, the output generated by `fmt` and `args` will be added to the trace. Otherwise, the `TracePrintf` is ignored.

  If you just specify `-t` without a *tracefile*, the trace file name will be `TRACE`. You can, if you want, give a different name with `-t foo`.

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

- `-lh` ***n***: Like -lk, but this one applies to the tracing level applied to the hardware. The higher the number, the more verbose, complete and incomprehensible the hardware trace.

- `-s`: Send the tracing output to the `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.

These switches are automatically parsed, interpreted and deleted for you before `KernelStart` is called. The remaining arguments in the command line are passed to `KernelStart` in its `cmd_args` argument. For example, when run with the sample command line above, `KernelStart` would be called with `cmd_args` as follows:

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

Inside `KernelStart`, you should use `cmd_args[0]` as the file name of the init program, and you should pass the whole `cmd_args` array as the arguments for the new process. You should provide a default init program name should the command line not provide you with one.

8.3 Debugging Your Kernel

You may use `fprintf`, `printf` and `TracePrintf` within your kernel. Please be aware that the insertion of print statements may alter the behavior of your program (thanks to Heisenberg).

If you are feeling lucky, you can also use the `dbx` debugger that comes with Solaris. However, because of the emulator's heavy use of signals, it is necessary to tell `dbx` to ignore certain events. You do this by typing

```
ignore 4 11 14 30
```

at the dbx prompt.

You may put "`ignore 4 11 14 30`" in the file `.dbxrc` in your project directory, if you don't want to have to type it every time you start up `dbx`. While you are at it, if you also put `dbxenv suppress_startup_message 3.2` in your `.dbxrc` file you will save yourself from having to see the long startup message every time you start up the debugger.

The equivalent to the above line for `gdb` should be something like:

```
handle SIGILL SIGFPE SIGSEGV SIGUSR1 pass nostop noprint.
```

9 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. Similarly, use of an "elegant" programming feature that results in unreasonable code-bloat will also result in a point deduction. You should use reasonable variable names and include comments in particularly tricky parts of the code, but you need not document code that does what it looks like. Also, when chosing to implement any particular feature, you should focus on solutions that are simple to understand and easy to debug over those that are excessively abstract and overly complicated.

Here is the approximate grading scale for the project:

- 70%. Correctness. Your kernel must be able to run a series of tests.
- 10%. On-time handin. This is for completing the subprojects on time.
- 10%. Programming style and efficiency.
- 10%. Participation (instructor discretion).

10 Approximate schedule

Please refer to the following dates when planning your time:

- Monday, April 3. Project handed out.
- Friday, April 7. You need to have formed a project group.
- Friday, April 14. Yalnix "Hello World" due.
- Friday, April 28. Basic memory management, virtual memory, and context switching.
- Friday, May 5. Process management.
- Friday, May 19. I/O Subsystem, device drivers.
- Friday, May 31. Everything!

Dates are subject to change at any notice. Graduating seniors must complete the project before the last day to submit grades.

11 Credits

The Yalnix emulation environment was originally developed by Dave Johnson at Rice University (now at Carnegie Mellon). I have made extensive modifications for my own sadistic purposes....