The Yalnix Kernel

CS230/330 Operating Systems, Winter 2001

Handed out : January 9, 2001

Due : March 1, 2001

1 Preface

In this project, you will be designing and implementing a simple operating system kernel. This document provides an overview of the entire kernel project. At first glance, 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 that follow the material presented in the lectures. Furthermore, your instructor and TAs all have experience working on prior versions of the kernel project. Therefore, do not panic if you do not understand all 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.

2 Introduction

Inspired by the results of election 2000, Larry has decided to launch a new dot-com business venture aimed at improving voting technology. Thus, with Chicago-MBA in hand, Larry managed to secure $120 million in venture capital to start his company Votémon.com. Votémon's business plan is to provide *free* electronic voting machines to districts throughout the country. In exchange, voters will be forced to view banner ads and watch movie trailers while making their most important civic decisions. In addition, they will be able to shop online and solve large systems of coupled partial differential equations. Of course, in the process, the Votémon system will record votes and buying habits so that they can be sold to direct marketers for a nice profit–this, of course, is the real innovation (and presumably patentable business practice).

Unfortunately, due to the downturn in the stock market, the $35 million post-IPO party in Cozumel, and the $20 million spent on futuristic personal work-pods, Larry now only has $1452.22 with which to build the prototype. For this, he has turned to the University of Chicago Computer Science Department. Thus, desperately needing the money the pay off the $30 he spent on Christmas Chili lights, Dave has accepted a $100 grant knowing that he could get all of the students in CS230 to build the system for free.

Therefore, your task is to write the operating system for the Votémon prototype, which in honor of the new President you have decided to call "Yalnix" (a horrible Texas-based pun on "Unix"). Since the real Votémon hardware has yet to be fabricated, your operating system will run a machine simulator that has been developed for Sun SPARC Solaris systems. 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, I/O, demand paging, and interprocess communication.

3 Project Overview

You will be implementing a fully functional operating system kernel from scratch. Although this kernel runs on a machine simulator, you will be responsible for the design and implementation of the *entire* kernel (i.e., you will not be given any skeleton code, nor will you be writing components for an existing kernel). Because of this, you have the freedom to implement your kernel however you see fit as long as it runs on the emulated hardware and provides the functionality described in this document.

Roughly speaking, your kernel will have to address three general problems. First, it needs to control the low-level Votémon hardware. To do this, you will have to write a collection of low-level device drivers. Second, your kernel needs to provide the abstraction of a "process." A process is simply a running program. Therefore, your kernel will need to figure out how to allocate memory between processes and to schedule their execution on the CPU. Finally, your kernel must provide a system call interface to user programs. Application programs will use this interface to request the services of your operating system.

The rest of this document describes various aspects of the system in detail. As the project is completed, you will receive additional handouts containing additional information and implementation tips.

4 The Votémon Hardware

The Votémon system currently consists of 1 Mbyte of DRAM, an 8 Mbyte flash-RAM storage card, a serial port, an LCD display, and a pen input device. All of these devices are controlled through a series of privileged machine registers. Furthermore, some of the devices use DMA (Direct Memory Access) to transfer data. In addition, the hardware generates a number of traps and interrupts that your kernel must handle in response to various events.

The file `/yalnix/include/hardware.h` contains all of the constants and data structures needed to control the hardware in your kernel implementation. You may want to look at this file as you read through this section.

4.1 Access to Privileged Machine Registers

The hardware is controlled by a collection of special registers that can be read or modified using privileged CPU instructions. The following privileged functions can be used by your kernel to access the registers:

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

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

- `unsigned int ReadRegister(int which)`

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

Each machine register is identified by a unique integer constant that is passed as the `which` argument. The `hardware.h` file defines symbolic names as for these constants as described in the following table:

Register 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_CONSOLE	Write character to console	Yes	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_PHYSM	Size of physical memory	Yes	No
REG_DCTLR	Display control register	No	Yes
REG_PEVTR	Pen event register	Yes	No
REG_SSTCR	Serial status and control register	Yes	Yes
REG_SADDR	Serial address register	No	Yes
REG_FCTLR	Flash control register	No	Yes
REG_FADDR	Flash address register	No	Yes
REG_FNBLK	Number of blocks on flash device	Yes	No
REG_POWLR	Power level register	Yes	No

All of the machine registers are 32 bits wide. In C, their values are represented as a type of `unsigned int`. You must use a C cast to convert other data types (such as pointers) 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`.

4.2 Traps

When a device needs attention or a program generates an error, a trap (or interrupt) is generated. The following list briefly describes all of the traps that can be generated by the hardware:

- `TRAP_KERNEL`

  This trap is generated when a user program executes a system call. See section 6.3.

- TRAP_CLOCK

  This is a periodic timer 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 6.2.

- TRAP_ILLEGAL

  This trap occurs when a user program tries to execute an illegal CPU instruction. When it occurs, the offending process should be terminated.

- 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 5.1.

- TRAP_MATH

  This trap occurs when a math operation executed by a user program generates a fault. For example, division by zero.

- TRAP_SERIAL

  This interrupt is generated by the serial device when it receives data or when it has finished sending data.

- TRAP_PEN

  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_PEVTR register.

- TRAP_FLASH

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

- TRAP_POWER

  This trap occurs when the user presses the power button. When received, your kernel should suspend operation and exit.

When a trap occurs, the CPU looks in an interrupt vector table to find out the location of an interrupt handler function. The above constants (e.g., TRAP_CLOCK) represent integer offsets into the vector table. The interrupt vector table is nothing more than an array of pointers to functions that your kernel must initialize. To do this, the privileged machine register REG_VECTOR_BASE must be set to point to the vector table within your kernel. The vector table must be statically dimensioned by your kernel with a size of TRAP_VECTOR_SIZE. 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 interrupt handler functions (the entries in the vector table) must have the following prototype:

```
void Trap_Handler(cpu_context_frame *)
```

(See section 5.5 for a description of the type `cpu_context_frame`).

Finally, it should be noted that the hardware blocks the delivery of all other interrupts until your kernel returns from the current trap (e.g., your trap handling functions will not be interrupted by other traps). This means that your kernel is generally not interruptible and that you will not need to implement kernel locking and synchronization primitives (be thankful).

4.3 Memory Management Hardware

Due to power considerations, the hardware only contains a modest amount of physical memory (DRAM). However, to provide additional storage and persistence after a power shutdown, the hardware additionally provides access to a large non-volatile flash-RAM storage card. A virtual memory system and MMU controller is then used to manage physical memory and to provide programs with the illusion of a large address space.

4.3.1 Physical Memory (DRAM)

As programs run, the CPU can only access data and instructions that are stored in the DRAM memory. Therefore, in order to run concurrent processes, your kernel will need to partition the physical memory in a way that allows it to be shared. The physical memory begins at an address defined by `PMEM_BASE` which is found in `hardware.h`. Although the current version of the hardware provides only 1 Mbyte of DRAM, the actual amount of memory is supplied to the kernel during the boot process and may vary in future revisions of the hardware. The size of physical memory can be read from the `REG_PHYSM` register.

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.

In the figure, the 1 Mbyte of physical memory begins at physical memory address `0x1000000` (page frame 2048) and extends to address `0x10fffff` (page frame 2175). Physical memory addresses lower than `PMEM_BASE` are used by the boot PROM and raw hardware devices. These addresses should not be used by your kernel. Similarly, attempts to access physical memory above the top of available memory will result in undefined hardware behavior (and a possible hardware fault).

In order to run concurrent user programs, your kernel will need to keep track of the physical memory pages that are in use and those that are available. To do this, you might just create an array of 128 entries and record a 0 or 1 to indicate usage. Of course, there are more elegant ways to do this. Also, the amount of physical memory may change as the hardware is improved so it would be a bad idea to hard-wire a fixed sized array like this in your kernel.

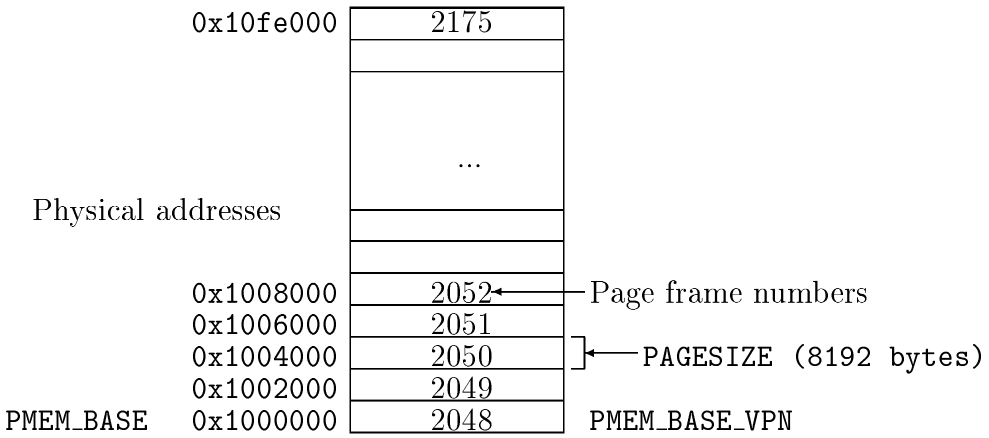

Figure 1: Physical memory layout

4.3.2 Flash memory

The flash memory card is not directly accessible by the CPU. That is, there is no way for the CPU to directly read or write data on the flash card. Therefore, the only way to access this data is to first copy it to DRAM and to write it back out to the flash card after modifications have been made.

Like physical memory, the flash device is organized as a collection of blocks, each of size FLASH_BLOCKSIZE. Currently the size of each block is 1024 bytes (note that this is *different* than the pagesize of DRAM). The total number of flash memory blocks can be obtained by reading the value of the `REG_FNBLK` register. This value may change depending on the size of the flash card that a user has installed.

Access to the flash card is managed through a pair of hardware registers and DMA memory transfer. Specifically, to initiate a memory transfer, it is first necessary to write a *physical memory address* into the `REG_FADDR` register. Then, a flash card operation is written into the `REG_FCTLR` register. The value written in the `REG_FCTLR` has the following format:

Bits 28-31	Bits 24-27	Bits 0-23
opcode	number of blocks	block number

The opcode field specifies the type of transfer and is one of the following values:

- `FLASH_READ`. Transfer a data block from flash memory to physical memory.
- `FLASH_WRITE`. Transfer a data block from physical memory to flash memory.

For example, the following code shows how one would initiate a flash card data transfer.

```
/* Transfer data from the flash card */
char    *paddr;          /* Raw physical memory address */
int      blocknum;       /* Starting block number on the flash card */
int      nblocks;        /* Number of blocks to transfer */

WriteRegister(REG_FADDR, (unsigned) paddr);
WriteRegister(REG_FCTLR, (FLASH_READ << 28) | (nblocks << 24) | blocknum);
...
```

Memory transfers involving the flash storage device must observe proper memory alignment. Specifically, the physical memory address written to the `REG_FADDR` register must be a multiple of `FLASH_BLOCKSIZE`. Upon completion of the memory transfer, a `TRAP_FLASH` trap is generated.

To manage the flash memory, your kernel will need to maintain a separate table that indicates free and used blocks. Furthermore, to implement swapping, you will need to provide some kind of mapping between virtual memory pages and blocks on the flash card.

4.3.3 Virtual Memory

Although the kernel, user programs, and data are stored in physical memory and the flash card, every program runs within a virtual memory space that is managed by a special MMU controller. This controller provides a simple paged memory scheme that relies upon the use of 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 running user process and will be changed by the OS whenever it performs a context switch. 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 privileged mode, references to both regions are allowed. However, when the CPU is not executing in privileged 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.

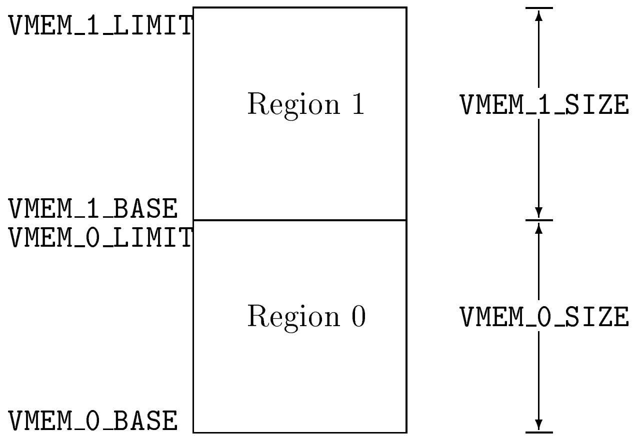

Figure 2: Virtual Memory Layout

- `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 virtual memory region provides an address space of 16 Mbytes (2048 pages) which is more than enough to accomodate the 1 Mbyte of physical memory.

It is 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).

4.3.4 Address Manipulation

Throughout the implementation of your kernel, you will need to manipulate memory addresses, 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`

  This is the size (in bytes) of a memory page in both virtual and physical memory.

- `PAGEOFFSET`

  This is a bit mask that can be used to extract the offset 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 an 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 that 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 the results may vary depending on the region. For example, address computations often need to include the constants `VMEM_0_BASE` or `VMEM_1_BASE` depending on the memory region of interest.

4.3.5 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 privileged registers.

- `REG_PTBR0`: Contains the *physical* memory address of the page table for region 0 of virtual memory.

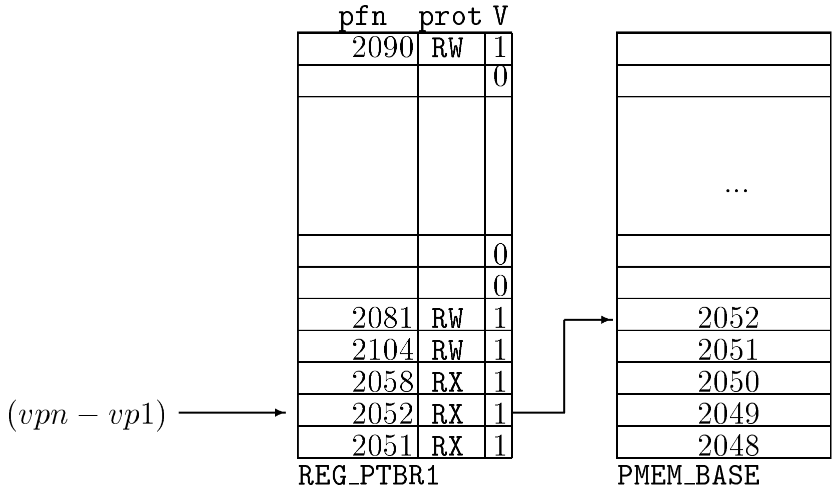

Figure 3: Page Table Lookup for Region 1

- `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 *physical* 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.

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.

The lookup to translate vpn to its currently mapped pfn is carried out wholly in hardware. As a result, the hardware imposes a strict format on the structure of the page table entries. The file `hardware.h` contains the definition of a C data structure (`struct pte`) that has the memory layout required by the MMU. You will need to use this data structure to manipulate page table entries from your kernel.

Each 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`.
- `access`: (1 bit) This bit is set when the hardware accesses a memory page (reading or writing). It is never cleared by the hardware–your kernel will need to do that.
- `dirty`: (1 bit) This bit is set whenever the MMU detects a write to a page. Your kernel can examine this bit to determine which pages have been modified. This bit is never cleared by the hardware.
- `user1`: (1 bit) User definable. Not used by the hardware. Your kernel can use this for its own purposes.
- `user2`: (1 bit) User definable. Not used by the hardware.
- `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. You might also want to use this field when implementing demand paging from the flash card.

The registers `REG_PTBR0` and `REG_PTBR1` contain *physical* memory addresses. As it turns out, the page table for each virtual memory region exactly fits into a single page of physical memory. Therefore, page tables can easily be allocated by your kernel by simply grabbing a free page of physical memory and placing its address in the appropriate page table register. However, in order to the modify the contents of the page tables, you will also need to map these pages someplace with in the kernel memory map.

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.

4.3.6 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 from memory. This is important because it can take several memory accesses to retrieve a page table entry[1].

The yalnix hardware also contains a TLB that is managed by the hardware MMU. Whenever a program accesses a virtual page, the mapping of that page to a physical frame number is automatically stored in the TLB. However, in certain cases, your kernel will need to flush all or part of the TLB. In particular, after changing a page table entry in memory, the TLB might 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 previous 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`.

4.3.7 Initializing Virtual Memory

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

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. In order 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 explicitly enabled by the kernel. To enable virtual memory, the kernel must first write the locations of the region 0 and region 1 page tables to the `REG_PTBR0` and `REG_PTBR1` registers. Then, it should execute the following:

[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.

```
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 it has been enabled.

Further details about kernel memory initialization can be found in the Section 5.2.

4.4 The Serial Port

The hardware features a serial port that is ordinarily used to communicate with a personal computer and other add-on devices. However, for the purposes of debugging and development of your kernel, you are going to use the serial port as a debugging console where you will be able to launch programs using a simple text-based shell.

The serial port is a line-oriented device that uses DMA to transfer data to and from physical memory. Two special registers are used to control the serial device:

- `REG_SADDR`. A write to this register sets the *physical* memory address that will be used in the next write to the `REG_SSTCR` register. Typically, this register is used to set the location of a buffer from which data will be sent or into which data will be received.

- `REG_SSTCR`. This register is used to issue commands to the serial device and to query the device about the last serial interrupt. The format of this register is as follows

Bits 24-31	Bits 16-23	Bits 0-15
opcode	unused	Length (in bytes)

  where the `opcode` field is one of `SERIAL_RECEIVE` or `SERIAL_TRANSMIT`.

To write data to the serial device, the *physical* memory address of the data is first placed in the `REG_SADDR` register. Afterwards, a `SERIAL_TRANSMIT` operation is written to the `REG_SSTCR` register. The length field of this operation should contain the number of bytes of data to send. Upon completion of the write, a `TRAP_SERIAL` interrupt will be generated. A maximum of 65536 bytes can be sent with a single write operation. However, your kernel will probably want to send data in smaller segments than that.

To receive data from the serial device, your kernel must first create an incoming data buffer and write its *physical* memory address to the `REG_SADDR` register. Then, a `SERIAL_RECEIVE` operation must be written to the `REG_SSTCR` register. In this case, the length field should contain the maximum number of bytes that can be stored in your kernel buffer. When a new line of input is received, a `TRAP_SERIAL` interrupt will be generated. The constant `SERIAL_MAX_LINE` specifies the maximum number of bytes that the hardware can receive on a single line. To simplify programming, your incoming data buffer should always be at least that large.

It is important to note that the serial device can support simultaneous read and transmit operations. That is, a transmit command can be issued while the device is waiting for input and input can be received while the device is transmitting. Whenever either command is issued to the device, the completion of that command is signalled by a `TRAP_SERIAL` interrupt. However, when this interrupt occurs, your kernel will have to read the value of the `REG_SSTCR` register to find out more information. When this register is read, the `opcode` field contains either `SERIAL_RECEIVE` or `SERIAL_TRANSMIT` to indicate the command that generated the interrupt. In addition, for receives, the length field will contain the number of bytes that were actually read by the device.

Although the device can support simultaneous read/transmit operations, *it is illegal to issue a new operation to the device until the previous operation of the same type has completed.* In other words, a new transmit command can not be issued until the previous transmit has completed. In addition, please note that the serial device operates on *physical memory addresses.* Writing a virtual memory address in the `REG_SADDR` register is likely to lead to extremely erratic and peculiar behavior.

4.5 The Display Device

The hardware features a 256x256 pixel LCD display. The display has its own microcontroller that understands simple commands written to a special `REG_DCTLR` register. Each display command is a 32 bit integer 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 display operations. However, your kernel must allow user programs to write display codes to the display device using a special system call (see Section 6.3.4).

4.6 The Pen Device

The hardware features a touch screen that detects the presence of a pen. Whenever the pen is touching the display, the hardware generates a `TRAP_PEN` interrupt. When generated, your kernel should read the status of the pen device by reading from the `REG_PEVTR` register. The return value of this register is encoded 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_PEVTR` register itself. Rather, it will simply have to queue up events and pass them on to user processes that request to read from the Pen device (see section 6.3.4).

4.7 Power Management

The hardware is powered by a set of batteries. Furthermore, various devices consume power as they are used. The remaining power can be obtained by reading from the `REG_POWLR` register. This returns a 16-bit unsigned integer. In addition, a `TRAP_POWER` interrupt is generated whenever the user hits the power key (pressing 'esc' when the mouse is over the Yalnix display window).

In designing your kernel, you will want to give some consideration to power consumption. Certain devices such as the flash card consume more power than others. As a result, clever algorithms and elegance in your implementation will have a noticeable impact on battery life.

4.8 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:

```
int
main(char *cmd_args[], cpu_context_frame *context);
```

This routine is part of your kernel and you must implement it. Normally, this function carries out all of the operations needed to initialize the virtual memory system as well as the I/O devices. Further details can be found in section 7.2.

4.9 Miscellaneous Hardware Operations

Normally, the CPU continues to execute instructions even when there is no useful work available for it to do. As a result, operating systems usually provide an *idle process* that is executed in this situation. Typically, this is just an empty infinite loop. However, in order to conserve power, the Yalnix hardware provides the following special instruction:

- `void Pause(void)`

  Puts the CPU in standby mode until the next trap or interrupt 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 exit from your kernel.

Finally, to assist in the debugging of your kernel, the hardware provides a special character output port (the console). To write a character to this device, the following command can be used:

```
WriteRegister(REG_CONSOLE, c);      /* c is a single ASCII character */
```

Unlike other devices, the console port does not generate interrupts. Once a character has been written, the value of the `REG_CONSOLE` register remains non-zero until the port is ready to write another character. Therefore, to use the console, your kernel will have to use polling to determine when it is safe to write a character. For example:

```
while (ReadRegister(REG_CONSOLE));   /* Spin until not busy */
WriteRegister(REG_CONSOLE,c);        /* Write the next character */
```

If characters are written to the console while the device is busy, they are ignored.

5 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 each 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 user processes and itself.

5.1 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.

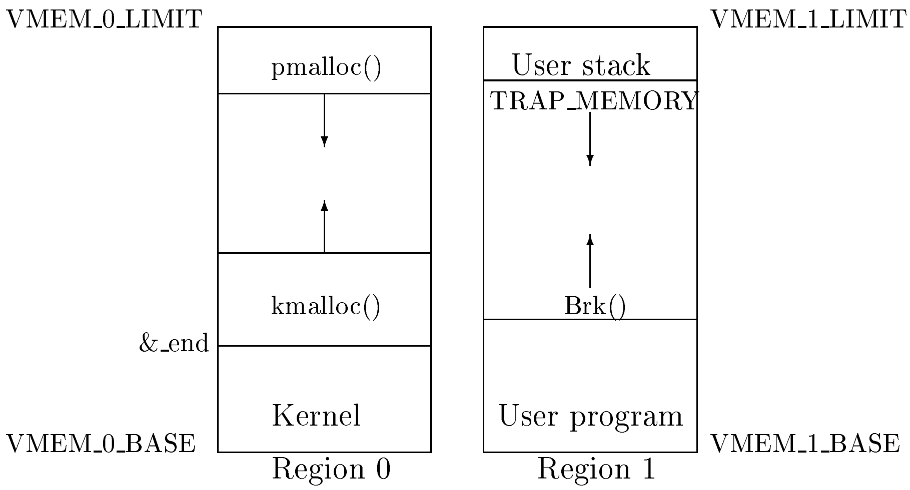

Figure 4: Process and Kernel Memory Management

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.

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 6.3.

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.

Due to the limited amount of physical memory, your kernel should limit the size of the user stack to STACK_MAX (currently 32 Kbytes). If the stack grows beyond this size, the kernel should kill the process with an out of memory error.

5.2 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. To make it easier for the kernel to run after virtual memory is enabled, 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 ends in memory. To find out, you can use the following values:

- `&_end` is the last address in the kernel's data segment.
- `&_etext` is the lowest address that does not contain executable instructions.

Thus, when your kernel boots, one of the first things it will do is examine the value of `&_end` and `&_etext` to find out how much memory it is using. All of this memory should be marked as "in use" and initialized in the kernel page table. The remaining physical memory pages should be placed on a free memory list before virtual memory is enabled.

To implement dynamic memory management, your kernel must implement it's own memory allocation functions. In fact, the standard C `malloc()`, `realloc()`, and `free` functions are `not` available to your kernel! Therefore, it is up to you to devise a suitable replacement.

There are two types of kernel memory management that your kernel is likely to require. The first is a page allocator that allocates memory in page-sized chunks. To do this, you should implement two functions:

- `void *pmalloc(int npages)`. Allocates `npages` pages of memory and returns a pointer to their virtual memory address. Returns 0 if no more physical memory is available.
- `void pfree(void *start, int npages)`. Releases `npages` of memory starting at address `start`.

The page allocator is particularly useful for creating large data structures such as page tables and PCBs. Since the virtual address space for the kernel is relatively large (16 Mbtyes), an easy way to implement the page allocator is to simply start at the top of the region 0 page table and look for any empty region that is big enough to hold the requested number of pages.

The second type of kernel memory allocator you will need is one optimized for small memory allocations (less than a page). To do this, implement two functions similar to `malloc()` and `free()` as follows:

- `void *kmalloc(int nbytes)`. Allocate an object of `nbytes` and return a pointer to its virtual memory location. Returns 0 if no more memory is available.
- `void kfree(void *ptr)`. Free an object previously created with `kmalloc()`.

The implementation of these functions is entirely up to you. One approach, is to implement a simple memory management scheme that utilizes a "kernel break" like the normal C `malloc()` function. This approach is described in the K&R C Programming Language book on page 185. Alternatively, you could try to divide virtual memory up into a collection of memory pools for different sizes of objects. Or, as a worst case scenario, you could just use the page allocator for everything and waste a lot of memory. In any case, a certain amount of creativity on your part will make the difference between an efficient kernel and a bloated kernel.

Finally, 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).

5.3 Demand Paging

Since the amount of physical memory is relatively small, your kernel is going to be severely limited in its ability to run many concurrent programs. To work around this limitation, your kernel should use the flash storage card as a backing store–allowing user processes to grow as large as available flash memory (even if this greatly exceeds the amount of actual physical RAM).

To make this work, your kernel should implement demand-paging or swapping to the flash card. In this case, when physical memory is exhausted, your kernel should go out and find a suitable physical page to replace (the victim page). This page should then be written to the flash card (if necessary) and released to the system for other uses. When a page is replaced, its corresponding page table entry should be invalidated and a record kept to indicate the location of the stored page on the flash device. If the victim page is accessed in the future, the demand paging system should bring the original page back into physical memory and update the page table entry accordingly.

The implementation of demand paging is likely to be somewhat difficult given the wide variety of page replacement algorithms and complicated effects on other parts of the kernel. You are free to implement any demand paging algorithm that you wish. In addition, the hardware provides some support for demand paging in the form of access and dirty bits which are set in each page table entry by the MMU. Your kernel can look at these values to try and figure out the best memory page to replace.

To "simplify" the implementation of demand paging, your kernel should adopt a few design strategies. First, all of the memory used by the kernel should be pinned in physical memory (e.g., kernel memory should never be swapped out). This will eliminate the possibility of something really nasty happening such as accidentally swapping out a page containing a process page table or a kernel buffer. Second, your kernel should maintain a small number of "spare" memory pages for its own use. These pages may be needed to satisfy additional kernel memory requests that might occur during page replacement processing. Finally, certain system calls like `RFork()` and `Exec()` tend to consume large amounts of memory. Therefore, the implementation of these system calls will probably have to interact with the demand-paging system in some manner.

5.4 Page Sharing

In order to implement threads (the `RFork()` system call), your kernel needs to support page-sharing between processes. This means that certain physical memory pages will shared by more than one process and be located in more than one page table. To make this work correctly, your kernel should manage the use of each physical memory page with a reference counting scheme. As the sharing of a page increases, its reference count should increase. As processes terminate or memory is released, the reference count should decrease. Only when the reference count reaches zero should your kernel put the physical page back on to its free memory list.

Page sharing introduces further complicates when combined with demand paging. In particular, your page replacement scheme should probably try to avoid swapping of pages with high reference counts. Furthermore, when such pages are written to the flash card and replaced, your kernel will need to update *all* of the page tables that might have reference that physical page.

5.5 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 `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 6.3).
- `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 6.3.

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(cpu_context_frame)`.

The 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.

6 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.
- Device drivers.
- Demand paging.

6.1 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 (`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 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.

6.2 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.

6.3 Yalnix System Calls

6.3.1 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 these general categories:

- Process management.
- Memory management.
- I/O.
- Synchronization.
- Interprocess communication.

Each of these are now described.

Note: The system call names listed in this section are the names used by Yalnix *user* processes. They are not necessary the same function names or prototypes that you will use in the implementation of your kernel.

6.3.2 Process management

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

- `int RFork(int flags)`

  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. The `flags` argument determines the memory behavior of the fork operation. If set to `FORK_PROCESS`, the memory of the child process is a copy of the parent process. If set to `FORK_THREAD`, the child process gets its own copy of the parent's stack, but *shares* its text, data, and heap segments with the parent process (i.e., both processes map to the same set of physical pages). If set to `FORK_EXEC`, the child process runs in the *same* address space as the parent (e.g., the child has the exact same set of page table as the parent). The `FORK_EXEC` mode is most commonly used when the child is immediately going to execute an `Exec()` system call, thus replacing its memory mapping with that of the new executable. If there is not enough memory to perform the requested 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 with 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 or thread 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 pid, int * status_ptr)`

  Collect the process ID and exit status returned by a child process or thread of the calling program. `pid` indicates a specific child PID or can be set to 0 to indicate any child. When a child process exits, it should be added to a 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. If pid is not a valid process ID or does not correspond to a child of the calling process, ERROR_CHILD 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.

- `int Yield()`

  The calling process is descheduled and returned to the ready state (i.e., placed back on the ready queue). The scheduler then switches to the next runnable process. If no other runnable processes are available, this system call does nothing and control is returned. The function always succeeds and returns 0.

- `int Suspend(int pid)`

  Suspends the execution of a process *pid*. The target process remains stopped until a Resume() operation is performed. The function returns 0 on success and ERROR_PID if *pid* doesn't correspond to a valid process. See section 6.5.

- `int Resume(int pid)`

  Resumes the execution of a stopped process *pid*. Zero is returned on success and ERROR_PID if *pid* doesn't correspond to a valid process. If insufficient memory is available to resume the process, ERROR_NOMEM should be returned. See section 6.5.

6.3.3 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.

6.3.4 Device I/O

Each I/O device is controlled by a dedicated set of system calls.

- `int DisplayWrite(int *buf, int nitems)`

  Writes the contents of the buffer referenced by *buf* to the LCD display device. *buf* is a buffer containing integer display opcodes as described in section 4.5. *nitems* is the number of items in the buffer. The return value should be *nitems* if the `Write` is successful. Otherwise, one of the following error codes should be returned:

  - `ERROR_FAULT`. *buf* refers to an illegal address.
  - `ERROR_INVAL`. *len* is invalid (a negative number perhaps).

- `int PenRead(int *buf, int maxevents, int noblock)`

  Reads up to *maxevents* pen event codes into the buffer specified by *buf*. Each event is an integer encoded in the format described in section 4.6. Returns the number of events written into the buffer. If *noblock* is set to 0, this call blocks until a pen event is received. If set to 1, the call returns immediately even if no events are pending (in which case 0 is returned). The following error codes should be returned:

  - `ERROR_FAULT`. *buf* refers to an illegal address.
  - `ERROR_INVAL`. *maxevents* or *block* is invalid (a negative number perhaps).

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

  Transmits *len* bytes of data referenced by *buf* through the serial port. The call blocks until all of the data has been successfully sent. On successful completion, the return value should be *len*. Otherwise, one of the following error codes should be returned:

  - `ERROR_FAULT`. *buf* refers to an illegal address.
  - `ERROR_INVAL`. *len* is invalid (a negative number perhaps).

- `int SerialReceive(char *buf, int maxlen, int noblock)`

  Receive a maximum of *maxlen* bytes from the serial port and store them in *buf*. If *noblock* is set to zero, the call blocks until data is actually received. Otherwise, the call returns immediately even if no data is available. On success, the number of bytes received are returned. Otherwise, one of the following error codes are returned:

  - `ERROR_FAULT`. *buf* refers to an illegal address.
  - `ERROR_INVAL`. *len* is invalid (a negative number perhaps).

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

  Writes *len* bytes of data referenced by *buf* on the Yalnix debugging console. This call never blocks. On success, *len* is returned. Otherwise, one of the following error codes should be returned:

  - `ERROR_FAULT`. *buf* refers to an illegal address.
  - `ERROR_INVAL`. *len* is invalid (a negative number perhaps).

6.3.5 Interprocess Communication

Your kernel needs to support a simple form of interprocess communication in the form of message passing. The following system calls are used:

- `int Register(int serviceid)`

  Registers the calling process as providing a service identified by the integer *serviceid*. Only one process is allowed to register any given service identifier. The identifier exists in the kernel until the process exits. Return 0 on success or `ERROR_INVAL` if the identifier is an illegal value or already in use.

- `int Send(int target, void *buffer, int len)`

  Sends a message of *len* bytes to the target process or service indicated by *target*. If *target* is a positive number, it is taken to be a process ID. If it is negative, its absolute value is taken as a service identifier previously registered with the `Register()` syscall. Returns the number of bytes sent on success. On error, one of the following values should be returned:

  - `ERROR_FAULT`. *buffer* refers to an illegal address.
  - `ERROR_INVAL`. *len* is invalid (a negative number perhaps).
  - `ERROR_MSG`. *target* is an invalid destination (a non-existent PID or service identifier).

- `int Receive(int source, void *buffer, int len, int noblock)`

  Receives a message of up to *len* bytes. If *source* is 0, a message can be received from any sender. If *source* is nonzero, it indicates that a message is to be received from a specific sender. A positive *source* is used to specify a PID. A negative *source* is used to specify a service identifier. The *noblock* flag indicates whether or not the read operation should block. On success, it returns the number of bytes that were received. On error, the following codes should be returned:

  - `ERROR_FAULT`. *buffer* refers to an illegal address.
  - `ERROR_INVAL`. *len* is invalid (a negative number perhaps).
  - `ERROR_AGAIN`. No message is available. Only returned in non-blocking mode.
  - `ERROR_MSG`. *source* is an invalid destination (a non-existent PID or service identifier).

6.3.6 Synchronization

In order to synchronize threads created with the `RFork()` system call, your kernel provide a few system calls related to semaphores. The following system calls should be implemented:

- `int SemaCreate(int` *`semaid`*`, unsigned int` *`value`*`)` Create a new semaphore identified by the integer *semaid*. The initial value of the semaphore is *value* which must be a non-negative integer. Returns 0 on success or `ERROR_SEMA` if the semaphore identifier *semaid* is already in use.

- `int SemaDestroy(int` *`semaid`*`)`

  Destroys the semaphore identified by *semaid*. Returns 0 on success or `ERROR_SEMA` if the semaphore identifier doesn't exist. When the semaphore is destroyed, all processes waiting on the semaphore should be unblocked and the `SemaWait()` syscall should return `ERROR_SEMA`.

- `int SemaSignal(int` *`semaid`*`)`

  Signals the semaphore identified by *semaid*. This increments the value of the semaphore and awakens at most one process that might be waiting on the semaphore (if applicable). Returns the value of the semaphore (before the increment) on success or `ERROR_SEMA` if the semaphore identifier is invalid.

- `int SemaWait(int` *`semaid`*`, int` *`noblock`*`)`

  Waits on the semaphore identified by *semaid*. If the value of the semaphore is non zero, the value is decremented and the system call returns. If the value is zero, the calling process is blocked until it is awakened by a process calling `SemaSignal()`. If *noblock* is set to 1, the function returns an error instead of blocking when the semaphore is zero. Returns the value of the semaphore (before the decrement) on success or `ERROR_SEMA` if the semaphore identifier is invalid. When *noblock* is set, `ERROR_AGAIN` should be returned if waiting on the semaphore would cause the process to block.

6.3.7 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.

6.4 Device Driver Implementation

A critical part of your kernel is going to involve the use of the serial device and the flash memory card. In the past, students have experienced a considerable amount of unwarranted trouble supporting hardware devices. This section offers a few suggestions on how to structure your kernel to make it easier.

First of all, realize that your kernel is really nothing more than a glorified queuing system. For example, consider the handling of the `TRAP_CLOCK` interrupt. Inside your kernel, you are going to keep a queue of processes that are ready to run (the ready queue). Then, on each clock interrupt, your kernel is going to remove the currently running process and place it at the end of the ready queue. Then, the first process on the ready queue will be popped off the queue and placed on the CPU. This process will run until the next clock interrupt occurs at which point it will be removed from the CPU and replaced with another process. Conceptually, this procedure should be pretty obvious.

The handling of other hardware devices is very similar to the handling of the clock interrupt. The only difference is that unlike the clock interrupt (which occurs automatically), the I/O devices operate using a command-interrupt cycle. That is, you first give the device a command. Then, some time later, an interrupt is generated to indicate completion of the command. Furthermore, once a command has been issued, no other operations can be scheduled on the device until an interrupt is received to signal its completion.

The implementation of your I/O device drivers and demand paging is going to be very similar to that of the clock interrupt handler. First of all, you will need a queue that holds all of the processes that are waiting to use a particular device. Then, you will need to implement an interrupt handler that responds to the interrupt that is generated when a device operation has been completed (e.g., TRAP_SERIAL or TRAP_FLASH). In this interrupt handler, your kernel will take the process for which the operation was performed and move it to another queue (perhaps the ready queue). Then, the handler will pop off the next process from its device queue and issue a command to the hardware device on its behalf. The final piece of the puzzle is how processes actually move from the ready-queue to the device queue. This usually occurs when a process executes a system call such as `SerialWrite()`. Figure 5 shows the sequence of events.

Another way to view the overall control flow is as a large state machine. Specifically, at any given time, each user process is only allowed to be in one particular state. System calls and interrupts move the process between different states of execution as shown in Figure 6.

6.5 Demand Paging (Revisited)

Section 5.3 described the use of the flash card memory in general detail. Specifically, your kernel should try to use the memory on the flash card to extend the amount of available

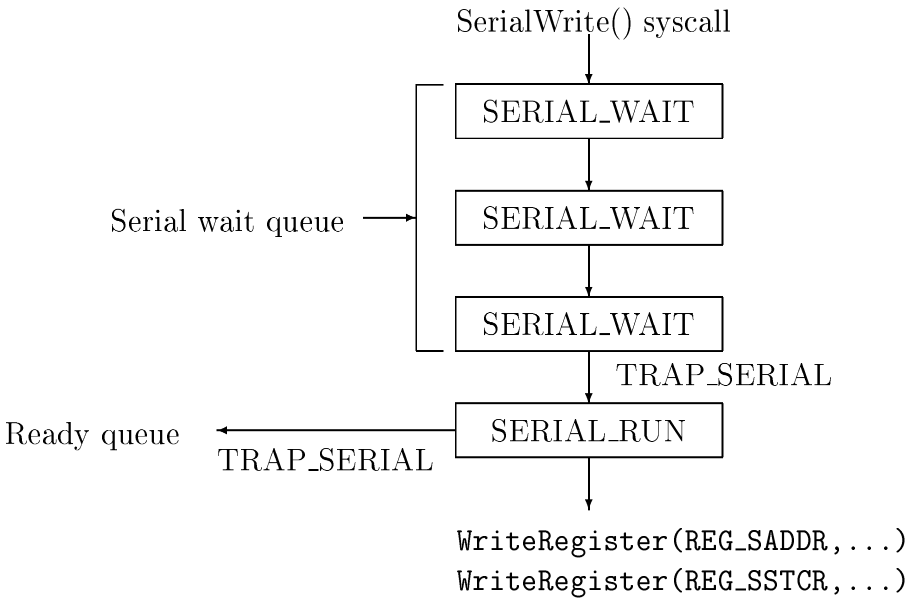

Figure 5: Control flow of serial device

memory for user processes. To do this, your kernel is going to have to know how to save and restore memory from the flash device as it is used. Most of this work will take place in your `TRAP_MEMORY` handler.

6.5.1 Undergraduate Requirements

This section does not apply to graduate students.

To support the flash card device, your kernel should implement a memory management scheme in which all of the runnable processes should be able to fit entirely within the physical memory of the machine. To do this, your kernel should enforce strict rules about memory utilization—specifically, when physical memory is exhausted, future memory allocations by user processes should result in errors (e.g., attempts to call RFork, Exec or Brk should fail with error codes).

Although your kernel must make sure that all of the running programs fit in physical memory, your kernel still must provide support for swapping using two special system calls (described in the previous section):

- `int Suspend(int` *`pid`*`)`

  This call puts a process to sleep–effectively removing it from all of the queues in the system. When a process is put to sleep, all of its memory should be paged out to the

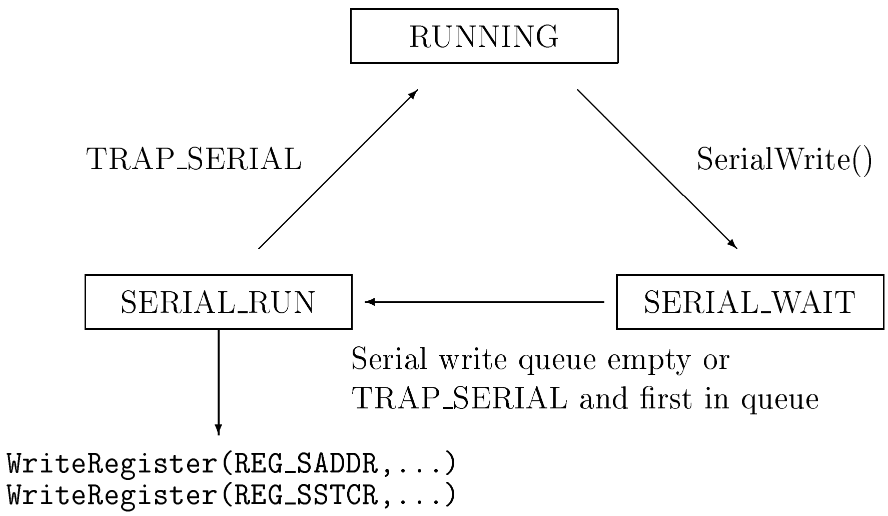

Figure 6: State transition diagram for a process calling SerialWrite()

flash card and its physical memory pages released back to the kernel. The caller of this system call should be blocked until all of the memory of the target process *pid* has been released. Upon the successful completion of this call, the newly released memory can be allocated to other tasks.

- `int Resume(int` *`pid`*`)`

  This call restores a process that was previously put to sleep. To do this, your kernel first needs to check if enough physical memory pages are available to bring the entire process back from the flash card. If not, nothing happens and an error `ERROR_NOMEM` should be returned. If enough memory is available, your kernel should reserve enough physical pages to hold the resumed process and arrange to have the process restored from the flash card. If the process can be restored, this function should return to the caller immediately without having to wait for the target process to be loaded from the card.

When restoring a previously suspended process, your kernel should use demand-paging to load stored memory pages into physical memory as they are actually used. To do this, the `Resume()` syscall should put the process back onto the ready queue with all of the pages still swapped out. Then, the `TRAP_MEMORY` handler should be modified to restore pages as they are actually accessed. The advantage to this approach is that a resumed process may only use a small fraction of its memory pages. By only restoring the pages that are needed, your kernel will extend battery life by reducing the number of flash card operations that are performed.

When a resumed process is later suspended again, your kernel should take steps to avoid unnecessary stores to the flash card. For instance, read-only memory pages should not be saved to the flash card more than once. Similarly, your kernel only needs to save memory pages that were actually modified (you can look at the dirty bits in page table entries to determine this).

If you are bored and would like to make your kernel a little more efficient, you can modify the implementation of `Suspend()` so that the memory pages of the suspended process are not actually written to the flash card until they are needed for other purpose.

6.5.2 Graduate Requirements

Extra credit for undergraduates

Your kernel needs to automatically support demand paging for all user processes and must be able to support swapping for running programs. Specifically:

- When physical memory is exhausted, memory allocations and system calls such as RFork() and Exec() must continue to operate as long as enough flash card memory is available. To do this, your kernel will have to free up physical memory pages by identifying victim pages and automatically writing them out to the flash card.
- User processes are allowed to grow beyond the size of physical memory as long as flash memory is available. For instance, since there are 8 Mbytes of flash memory, your kernel should theoretically allow an 8 Mbyte user process to run.

To implement this, your kernel will need to implement some kind of page-replacement algorithm. This algorithm should monitor physical memory pages and make decisions about the most appropriate pages to write to the flash device. For example, you might implement an approximated LRU scheme by periodically examining the access and dirty bits of page table entries.

Like the undergraduate kernel, your implementation must also support the `Suspend()` and `Resume()` system calls. However, your implementation of these system calls is as follows:

- `int Suspend(int` *pid*`)`

  This call puts a process to sleep–effectively removing it from all of the queues in the system. This call returns immediately to the caller regardless of whether or not the memory of the target process has been saved to the flash device. The memory of the target process should only be written to the flash card if it is actually needed for other purposes. You might implement your page replacement algorithm so that it preferentially tries to victimize suspended processes before stealing memory from running processes.

- `int Resume(int` *pid*`)`

  This call restores a previously suspended process. The suspended process is simply put back on the ready queue and allowed to run. If any of its memory was swapped out to the flash card, it should be restored as the process causes generates `TRAP_MEMORY` exceptions. Unlike the undergraduate requirement, this function always succeeds–even if there is not enough physical memory to bring back all of the suspended process.

Your kernel should try to be somewhat intelligent about the pages it saves to the flash card. For example, read-only pages should only be written to the flash card once. Similarly, previously saved, but unmodified data pages can simply be discarded rather than repeatedly saving their contents.

6.6 Power Management (EXTRA CREDIT)

As previously mentioned, the hardware is powered by a set of batteries. Furthermore, the use of various devices such as the flash card drain the batteries at different rates. For instance, when idling, the hardware will run for about 20 minutes. However, under constant use of the flash card, the hardware only runs for about 5-6 minutes.

To prolong battery life, your kernel should implement the following features:

- Use copy-on-write page sharing for RFork() and related system calls. This will reduce the memory requirements for all running programs.
- Use LRU page replacement to reduce the amount of swapping.
- Support for the `TRAP_POWER` interrupt. When received by your kernel, your kernel should go into suspend mode in which *all* processes and associated kernel state are dumped to the flash storage device before the machine is halted. On subsequent power-up of the hardware, your kernel should restore all of its data and return to the same execution state as when the `TRAP_POWER` interrupt was received.

The first few requirements are relatively straightforward enhancements to existing parts of the kernel. The last requirement (kernel suspend) will require you to figure out how to swap the kernel itself out to the flash storage card and how to restore it on power up. This will probably require modifications to the kernel boot-loader and your kernel startup procedure.

7 Implementation hints

7.1 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.

RFork

On a RFork, 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

7.2 Kernel Initialization

Your kernel begins execution at the procedure main. As already shown in section 4.8, this function is called from the boot firmware, which expects it to conform to the following prototype:

```
int
main(char *cmd_args[], cpu_context_frame *context)
```

The procedure arguments above are built by the boot ROM and passed to your main 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 context argument is a pointer to the initial cpu context frame (see section 5.5). 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 `RFork`, 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 `RFork`.)

Before allowing the executing of user processes, the `main` 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 value of the `REG_PHYSM` register which should be read in your `main()` procedure. The list should of course not include any memory that is already in use by your kernel.
- Enable virtual memory
- Create the idle process based on the cpu context passed to your `main` routine. The idle process is the process that gets scheduled onto the CPU when there is no other process ready to run. See `Pause` in section 4.9.
- 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 `main` as one of the `cmd_args` strings.
- Return from your `main` 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.

7.3 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.

[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.

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. Think about all of the queues and process states that your kernel will need. When in doubt, remember that your kernel is really nothing more than a huge queueing system.
3. 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.
4. Implement a skeleton of your kernel in which there are simple functions for every trap and system call.
5. Write some debugging support and utility functions to help you with the development of the kernel. For example, you might implement some function to write debugging information to the hardware console.
6. Write a simple in-kernel idle "process" that prints a message whenever it runs. Modify your kernel so that it boots up and starts running the idle process. Then you should experiment with generating different types of interrupts as the idle process runs (`TRAP_CLOCK`, `TRAP_PEN`, `TRAP_SERIAL`, etc.).
7. 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 "`===>>>`".
8. Implement kernel memory management and the `TRAP_MEMORY` handler.
9. Create some functions for creating processes and define your PCB structure.
10. Implement the `ConsoleWrite()` system call so that programs can generate some output.
11. Write an idle program (a single file that simply loops forever and prints some messages using `ConsoleWrite()`). Modify your kernel so that it loads the idle program into memory and starts its execution. If you can get the idle process to run, you have successfully bootstrapped virtual memory.
12. Write an "init" program. The simplest init program would just loop forever and print messages. Modify your kernel `main()` function to load "init" in addition to the idle program. You will now have two separate programs loaded at the same time.
13. Modify the `TRAP_CLOCK` handler to context switch between the idle and init programs.
14. 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.

15. 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.

16. Implement a few additional I/O syscalls so that user programs can start to generate some output. For example, `DisplayWrite()`, `SerialWrite()`.

17. Implement the `PenRead()` and `SerialRead()` syscalls. Now your kernel should have fairly good control over of the I/O devices.

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

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

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

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

22. By now, you should be able to run a simple shell proocess on your kernel.

23. Implement threads and semaphores.

24. Implement the `Register()`, `Send()`, and `Receive()` syscalls.

25. Implement the `Suspend()` and `Resume()` syscalls along with support for demand paging.

26. Implement and test anything else that is remaining. This should be a piece of cake unless you are still working on process management the night before the deadline.

27. Look at your work and wonder in amazement.

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!

8 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 main 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.

Minimally, to receive a passing grade for the project, your kernel must be able to boot up to a simple shell and execute simple commands. In addition, your kernel must support the basic I/O devices (display, pen, serial).

9 Logistics

9.1 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 GNU C compiler (gcc) can be used to compile your kernel and user programs. Although `g++` has been installed on the machine, your kernel must be implemented in ANSI C. In part this is due to the fact that your kernel does not have access to the standard C or C++ libraries—making it extremely difficult to utilize basic languages features like the "new" and "delete" operators.

9.2 Running your kernel

Your kernel will be compiled and linked as an executable image 'yalnix' that will be loaded onto the simulator hardware. To run your kernel, a special boot loader is used. This loader takes your program 'yalnix', loads it into physical memory and starts execution at the `main()` function defined in your kernel. The boot loader is invoked as follows:

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

For example,

```
yboot -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 `main` is called. The remaining arguments in the command line are passed to `main` in its `cmd_args` argument. For example, when run with the sample command line above, `main` 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 `main`, 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.

9.3 Debugging Your Kernel

Due to the restricted execution environment, your kernel only has access to a limited amount of functionality. To produce output to a trace file, the `TracePrintf()` function may be used. Otherwise, all other output must be produced using hardware registers. One way to produce debugging output is to write a collection of debugging functions that write characters to the `REG_CONSOLE` register. In the first part of the project, you will be writing a collection of debugging functions to help you with this.

If you are feeling somewhat masochistic, the gdb debugger can be used in a limited sense. However, due to the emulator's heavy use of signals and special memory mappings, it is almost impossible to obtain useful information. If you do choose to use gdb, you may want to enter the following line so that certain signals are ignored:

`handle SIGILL SIGFPE SIGSEGV SIGUSR1 pass nostop noprint.`

As an alternative to gdb, your instructor and TAs have developed a customized set of debugging tools that are specifically designed for the Yalnix project. When your kernel fails, it dumps a special 'ycore' file in the current directory. The program `ydb` (located in `/yalnix/bin`) can then be used to examine this file and produce a traceback. To use it, simply type 'ydb' after your kernel has crashed. You will get output similar to the following:

```
stonecrusher % ydb
ydb starting.
Reading ycore...
8 stack frames read.
18996 stack bytes read.
Reading symbols from 'yalnix'
Reading symbols from '/yalnix/etc/hardware'
Reading symbols from 'yboot'
Kernel stack trace:
#0 0x01000cac in TrapClock(ffbdd858,411f6ec,1000bf4,10080a8,40f4fe8,1d)
#1 0x0402835c in RealSignalHandler(e,0,ffbddd40,ffbdd898,ffbdd898,6)
#2 0x0402883c in SignalHandler(4,ffbddf78,ffbddcc0,0,0,0)
#3 0x040f51c0 in sigacthandler(4,ffbddf78,ffbddcc0,0,0,40f5400)
#4 0x04026b6c in _InitExceptions(411e0ac,410384c,4103844,3f8,2,40f5400)
#5 0x04023b2c in main(1,ffbefc8c,ffbefc94,80000012,4,0)
#6 0x000115f0 in main(1,ffbefc8c,ffbefc94,22000,0,0)
#7 0x00010e40 in _start(0,0,0,0,0,0)
stonecrusher %
```

10 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).

11 Approximate schedule

Please refer to the following dates when planning your time:

- Tuesday, January 9. Project handed out.
- Friday, January 19. You need to have formed a project group. Yalnix "Hello World" due.
- Friday, February 2. Basic memory management, virtual memory, context switching, and simple I/O devices.
- Friday, February 9. Process management.
- Friday, February 23. Threads, semaphores, and IPC.
- Friday, March 1. Everything else.

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

12 Credits

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