# Project 2
# The Yalnix Kernel
### CS230/330 Operating Systems, Spring 1999

## 1 Introduction

For most of the quarter, you will be designing and implementing a simple kernel for an operating system. This document is intended to provide an overview of the entire kernel project. At first, this may appear very daunting. However, to make it easier for you and to ensure steady progress, the project has been divided into several sub-projects. Each of these sub-projects builds on the previous, starting with a relatively trivial initial project. Do not panic if you do not understand every bit of this document after a first reading—it should all become clear as you work on the projects and as more material is covered in class. Descriptions of each subproject will be handed out separately.

**Important Note:** You will not receive a grade higher than a C for this course unless your kernel passes all of the basic tests for the complete kernel in project 2.4. It is critical that you not fall behind and that you see us for assistance if you hit a snag. We *want* you finish the project and are here to help you do just that.

## 2 Overview

Not satisfied with his laptop, cell phone, pager, and Palm Pilot, Larry (a.k.a. "Geeky Larry") has decided that the development of a Portable Immersive Environment (PIE) is the only way that he will be able to simultaneously drive his minivan, surf the web, forward jokes to all of his friends via e-mail, check stock quotes, play net Quake, solve partial different equations, and watch the Jerry Springer show. Lacking the resources to build such a contraption and having been ridiculed by no fewer than 37 venture capitalists, Larry had only one place to turn–the federal government. Thus, in a stroke of genius, Larry was awarded a 5 million dollar research grant to build the PIE prototype. Not long thereafter, Larry found his office to be completely infested with university professors all wanting to work on pieces of the PIE project (unfortunately, by the time they had been discovered, they had chewed through all of the software in Larry's office except Powerpoint–but I digress). In any event, the Center of Wearable and Portable Immersive Environments (COWPIE) was formed to design and build the prototype system.

Now, to make a long sordid story short, it was finally decided that the most practical implementation of the PIE prototype would consist of three components: a helmet with a 180 degree field-of-view high resolution LCD display, a head-mounted satellite dish, and a cell-phone antenna, a 95-pound backpack containing Lithium-Ion batteries and a small liquid-cooled supercomputer, and a shoulder mounted force-feedback arm made out of cast-iron. DARPA also wanted to add a gun-turet to the design, but backed off of the idea when they discovered it would interfere with television reception.

The contract to build the prototype was put out for competitive bids and the contract was awarded to Bubba's Computer Shack and Welding Shop in Amarillo, Texas for a total cost of $937.23–narrowly beating the 2.7 billion dollar bid by Silicon Graphics (although there has been some speculation about the odd coincidence of Bubba being the half-nephew of the program manager's first cousin's boyfriend's dead sister-in-law). Unfortunately, spooked by Y2K, Bubba started babbling a bunch of nonsense like "Natas si Retrac" and ended up moving to the hills of Colorado (e.g., Aspen) to join a millenium survivalist cult. Before he left however, a prototype of the PIE system was developed in the form of a simulator for use on SPARC machines running Solaris. Currently, the computer hardware has a simple MMU, a single, small TLB, and support for terminal devices.

Your task is to implement an operating system kernel for the PIE computer system, which in deference to Bubba, you have called "Yalnix" as an awful Texas-based pun on "Unix." Yalnix includes the main process management features of Unix, plus a few miscellaneous kernel primitives to form a complete system. We don't require that Yalnix support signals or threads. Furthermore, the PIE hardware does not yet support disks, so you will not be implementing block I/O operations or demand paging. It will also be unnecessary to program any portions of your project in assembly language nor will you need to program any low-level machine-specific functionality. You will need to implement the concept of a *user process* as a unit of memory protection that shares a single CPU with other processes. Thus, you will need to implement process tables and scheduler and use the MMU and TLB judiciously to implement virtual memory. In addition, you will need to support a simple form of interprocess communication.

Your kernel will service user requests (Yalnix syscalls), traps (from the MMU and CPU), and interrupts (from the clock and other devices). These are the three main "interfaces" you will be implementing. The kernel also has an additional entry point used only at boot time.

This document describes the hardware environment in which your Yalnix kernel will run. The required kernel operations that you will implement over the course of the quarter are described in section 10.

# 3 The PIE hardware

The hardware features you will need to use fall in 5 categories:

- Memory management (description of the TLB and associated registers)
- Interrupts
- Context (where the state of the CPU can be found.)
- Devices (terminal I/O)
- Bootstrap entry point.

The hardware is controlled through several *privileged machine registers*, including the TLB. These special registers are read and written via several privileged CPU instructions.

The file `/u0/yalnix/include/hardware.h` defines constants and structures that describe the hardware. You might want to refer to it as you read the following sections.

### 3.1 Access to Privileged Machine Registers

The hardware contains several registers that can only be read or written via privileged CPU instructions. These registers are used to control the behavior of different hardware features. The hardware provides two instructions for reading and writing these privileged machine registers:

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

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

- `unsigned int ReadRegister(int which)`

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

All registers are readable and writable, but the result of reading some of them is undefined.

Each privileged machine register is identified by a unique integer constant passed as the `which` argument to the instructions. The file `hardware.h` defines symbolic names for these constants. In the rest of the document we will use only the symbolic names to refer to the privileged machine registers.

The registers provided by the hardware are:

| Reg. Name | Readable | Writable |
|---|---|---|
| `REG_VECTOR_BASE` | Yes | Yes |
| `REG_VM_ENABLE` | Yes | Yes |
| `REG_TLB_FLUSH` | No | Yes |

| Reg. Name | Readable | |
|---|---|---|
| `REG_PTBR0` | Yes | Yes |
| `REG_PTLR0` | Yes | Yes |
| `REG_PTBR1` | Yes | Yes |
| `REG_PTLR1` | Yes | Yes |

he purpose of these registers will be described below or in a later document.

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

## 4 Memory Management Hardware

In managing physical memory, the operating system usually needs to be able to map different programs to different areas of physical memory. The Operating System can do this in many ways. For example, MS-DOS and older versions of the Mac OS allocated a contiguous chunk of physical memory to each program, and modified memory references by adding the base address onto which the program was loaded. This modification could be done at load time, in software, or at run-time, with the help of the hardware, e.g, through the use of base/limit registers. However, such contiguous allocation incurred a heavy fragmentation penalty, and in the 1970s a less onerous allocation scheme emerged in which a level of indirection was used to map the program's contiguous address space into many pieces of real memory that needn't be contiguous. The problem was that, for performance reasons, this mapping from virtual to physical addresses needed to be done entirely in hardware, as it would be prohibitive to invoke the operating system every time a program made a memory reference. This address translation was done in specialized hardware that came to be known as the MMU (Memory Management Unit.)

In a virtual memory system, then, the address translation for program memory references is done in hardware, but the allocation of memory – deciding which virtual addresses should map to which physical addresses – is done, by the operating system, in software. Because of this, the operating system and the MMU must collaborate on the implementation of the translation: typically, the MMU defines the format of translation tables, and forces the operating system to use this format in establishing the virtual-physical mapping. These tables are called "page" tables because the mapping is done at a certain granularity (the mapping is done in chunks of contiguous addresses,) and the unit of translation (the size of the chunks) is called a memory "page."

## 4.1 Virtual Memory

The PIE machine supports a paged memory mapping scheme using page tables. The virtual address space of the machine is divided into two *regions* called region 0 and region 1. By convention, region 0 is used by the operating system 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 switched by the OS whenever a context switch occurs. The hardware provides two regions so that it can protect kernel data structures from illegal tampering by the user applications. When the CPU is executing in priviledged mode, references to both regions are allowed, but when the CPU is not executing in priviledged mode, references to region 0 are not allowed. The hardware does this without intervention from your kernel.

These regions are not segments. Segments are typically associated with base and limit registers that are managed by the operating system. The regions in the PIE hardware have locations and sizes fixed in the virtual address space by the MMU hardware. Furthermore, the two memory segments only refer to the organization of the virtual address space. Physical memory is not divided into separate regions and can be arbitrarily assigned to either of the virtual memory regions.

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. The file `hardware.h` defines the following symbolic constants:

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

The size of pages in virtual memory is defined by the hardware. `hardware.h` defines the following symbolic constants and macros to make address and page manipulations easier:

- PAGESIZE: The size of a virtual memory (and as we will see, physical memory) page.
- PAGEOFFSET: A bit mask that can be used to extract the offset of an address into a page (`vaddress & PAGEOFFSET`).
- PAGEMASK: A bit mask that can be used to extract the base address of a page given an address (`vaddress & PAGEMASK`).
- PAGESHIFT: $\log_2$ PAGESIZE, the number of bits an address must be shifted right to obtain the page number, or the number of bits a page number can be shifted left to obtain its base address.
- UP_TO_PAGE($x$): Rounds address $x$ to the next highest page boundary. If $x$ is on a page boundary, it returns $x$.
- DOWN_TO_PAGE($x$):] Rounds address $x$ to the next lowest page boundary. If $x$ is on a page boundary, it returns $x$.

Page numbers in virtual memory are referred to as *virtual page numbers*.

## 4.2 Physical Memory

The machine supports a large physical space, into which may be loaded many concurrently executing processes. The physical memory of the machine begins at address PMEM_BASE (defined in `hardware.h`.) This value (and the values in 4.1) is determined by the machine's architecture specification. On the other hand, the total size of the physical memory is determined by how much RAM is installed on the machine. This value is determined by firmware at boot time and is supplied to your kernel (we tell you how later.) The size of physical memory is unrelated to the size of the virtual address space of the machine.

Like the virtual address space, the physical memory is divided into pages of size PAGESIZE. The macros PAGEOFFSET, PAGEMASK, PAGESHIFT, DOWN_TO_PAGE($x$) and UP_TO_PAGE($x$) may also be used with physical memory addresses.

PMEM_BASE is specified to be equal to VMEM_0_BASE.

Page numbers in physical memory are often referred to as *page frame numbers*.

## 4.3 Page Tables

The MMU hardware translates references to virtual memory addresses into their corresponding references to physical memory addresses. This happens without intervention from the kernel. However, the mapping that specifies the translation is controlled by the Yalnix kernel. This makes sense because memory references occur very frequently, while the virtual-to-physical map changes relatively infrequently and in accord with the process abstraction implemented by the kernel.

The Yalnix kernel defines the mapping from virtual to physical addresses via single-level, direct, page tables. Such a page table is an array of page table entries, laid contiguously in memory. Each page table entry contains information relevant to the mapping of a single virtual page.

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

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

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

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

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

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

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

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

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

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

The registers `REG_PTBR0` and `REG_PTBR1` contain *virtual* addresses. In a real operating system, these registers would contain *physical* addresses, as required by the MMU[1]. To simplify your programming task, in this project we instead make the MMU access these tables by their virtual addresses, which should be mapped in region 0.

#### 4.3.1 The Translation Lookaside Buffer (TLB)

Most architectures contain a TLB to speed up address translation. The TLB caches address translations so that subsequent references to the same virtual page do not have to retrieve the corresponding page table entry anew. This is important because it can take several memory accesses to retrieve a page table entry[2].

Our machine also contains a TLB. The TLB is managed by the hardware MMU, so you do not need to worry about caching page table entries yourself. Whenever a program accesses a virtual page, the mapping of that page to a physical frame number is automatically stored in the TLB. However, at times, your kernel will need to flush all or part of the TLB. In particular, after changing a page table entry in memory, the TLB will contain a stale mapping for the virtual addresses corresponding to the modified page table entry. Also,

[1]Otherwise the MMU would enter a loop

[2]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.

when carrying out a context switch from one process to another, the MMU will continue to map the cached entries of the extant process; unless the TLB is flushed, this will cause virtual addresses in the new address space to be mapped to the physical pages of the old address space.

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

- `TLB_FLUSH_ALL`: Flush the entire TLB.
- `TLB_FLUSH_0`: Flush all mappings for virtual addresses in region 0 from the TLB.
- `TLB_FLUSH_1`: Flush all mappings for virtual addresses in region 1 from the TLB.
- *addr*: Flush only the mapping for virtual address *addr* from the TLB, if present.

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

## 4.4 Initializing Virtual Memory

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

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

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

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

There is another problem caused by the need to bootstrap virtual memory: the kernel, like any other program, references data by their addresses. If the address of the location of the kernel changes, the addresses of the data references within it also need to change. But these references are generated by the compiler, and are very difficult to change while the kernel itself is running. You could *relocate* the kernel before enabling virtual memory, but there is an easier solution: you can make sure that both before and after virtual memory is enabled, your kernel is loaded into the same range of addresses. Your kernel should, before enabling virtual memory, map its own physical frames so that $vpn == pfn$ in that range at the bottom of region 0. This works because the hardware guarantees that `PMEM_BASE` is the same as `VMEM_0_BASE`.

One minor problems is how do you know where the memory range ends that contains the kernel at the bottom of region 0? The following may be useful for this purpose:

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

Before virtual memory is enabled, the kernel has access to the full extent of physical memory. If it needs to allocate storage dynamically, it can just grab some memory without further ado. However, to help you write the kernel, we provide versions of `malloc` and friends that work correctly even before virtual memory is enabled. This frees you from having to manage physical memory while bootstrapping, and from having to make sure your kernel's dynamic storage remains accessible afterwards. In particular, the `sbrk(0)` of the previous paragraph does not work if your kernel allocates its own dynamic storage without the use of `malloc` and friends.

We also provide versions of `malloc` and friends that work after virtual memory is enabled, but since the virtual memory space is managed by your kernel, those versions require assistance from your kernel. This is discussed later.

## 4.5 Process Memory Model and Allocation

**User processes**

The stack on our architecture grows from higher to lower virtual memory addresses. Thus, the convention for user processes is to place the stack at the top of the user virtual address space (`VMEM_1_LIMIT`,) allowing it to grow downward toward the program and data of the process. The convention is also for the program and data of the user process to be placed at the bottom of the user virtual address space (`VMEM_1_BASE`,) to leave as much room as possible for growth in between. Programs often need to dynamically allocate memory (e.g, using `malloc`,) and the convention is for this heap area to grow upward towards higher addresses. The space between the bottom of the stack and the top of the heap is unused. At any given point, these unused pages are marked invalid using the `valid` field of `struct pte`.

The area containing the program, data and heap of a program is called the process' *program area*, and the area containing the stack of a program is called the process' *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 by the `Brk` kernel call (system call) which your kernel will implement (see section 10.2.)

In the library with which your user programs are linked by the provided `Makefile`, we provide versions of the familiar `malloc()` family of procedures which call your kernel's `Brk()` syscall. Thus, while your kernel must manage user processes' break via the `Brk()` syscall, your user programs can call `malloc()`, `realloc()`, etc, as if you were writing regular UNIX programs.

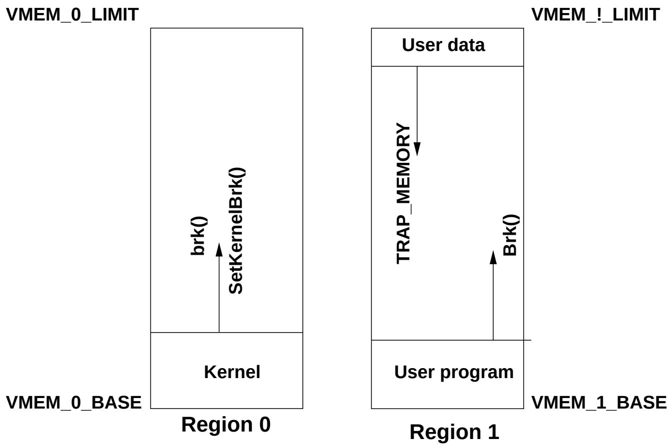


Figure 1: Virtual address space

The stack area of a process will be 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. The correct way to enlarge a process' stack area is described in section 5.

**Kernel**

In section 4.4 we explained that at boot time, the hardware loads the kernel into a range of *physical* addresses starting at `PMEM_BASE`. In that section we also explained that, when virtual memory is initialized, the kernel should remain mapped to the similar range of *virtual* addresses starting at the address `PMEM_BASE` (which is the same as `VMEM_0_BASE`.) After virtual memory is enabled by the kernel, the kernel can grow by calling `malloc`, etc, and the new pages need not be mapped to any particular physical memory location.

The kernel grows when it needs to allocate dynamic memory, e.g, using `malloc()`. We saw in the previous section that user programs call the `Brk()` syscall to effect this growth. Who implements `Brk` for the kernel? The kernel itself must.

In the library with which your kernel is linked by the provided `Makefile`, we provide versions of the familiar `malloc()` family of procedures which call the procedure `SetKernelBrk()` when they need to augment their heap of available memory. `SetKernelBrk()` must be provided by you. The C function prototype of this procedure is:

- `void SetKernelBrk(caddr_t addr)`

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

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

If running directly on 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 (another feat of the PIE design committee).

# 5 Traps

Your kernel runs only when its services are required. Its services are requested via interrupts, calls and exceptions managed by the hardware. In the case of interrupts, it is the hardware (clock or terminals) that requests the service. In the case of kernel calls, the hardware acts on behalf of the currently scheduled user program. In this case the hardware must act as an intermediary in order to put itself in privileged mode and restrict what code the CPU executes in that mode. In the case of exceptions, the hardware notifies the kernel that something has gone hay-wire, somewhere. The actual cause of the problem may be something that the kernel expects and can fix, such as the exception raised when a user program tries to access unmapped virtual memory.

Interrupts, kernel calls and exceptions are collectively referred to as "traps".

When requesting the kernel's service, the hardware transfers execution to an address obtained from a *vector table*. The vector table is an array of procedure addresses (pointers to procedures, in C nomenclature) indexed by the type of exception, call, or exception raised by the hardware. The following interrupts, calls, and exceptions have entries in the vector table. Their symbolic names are defined in `hardware.h`:

- `TRAP_KERNEL`

  This trap results from a "kernel call" ("syscall") trap instruction executed by the current user processes. All syscalls (such as `Brk`) enter the kernel through this trap.

- `TRAP_CLOCK`

  This interrupt results from the machine's hardware clock, which generates periodic clock interrupts. Your kernel should do round-robin scheduling of the runnable processes. Each time you get a `TRAP_CLOCK`, you should do a context switch to the next process in the ready queue. That is, the scheduling quantum should be one clock tick.

- `TRAP_ILLEGAL`

  This exception results from the execution of an illegal instruction by the currently executing user process. An illegal instruction can be an undefined machine language opcode, an illegal addressing mode, or a privileged instruction when not in privileged mode. When you receive a `TRAP_ILLEGAL`, your kernel should abort the current Yalnix user process, but should continue running other processes.

- `TRAP_MEMORY`

  This exception results from a disallowed memory access by the current user process. The access may be disallowed because the address is not a valid virtual address (neither in region 0 nor 1,) because the address is not mapped in the page tables, or because, even though the address is mapped, the access requires permissions on the corresponding page table entry which were not permitted by the `prot` field of the page table entry. You will receive a `TRAP_MEMORY` exception when a Yalnix user process attempts to grow its stack beyond its currently allocated stack. If the virtual address being referenced that caused the trap is in region 1, is below the currently allocated memory for the stack, and is above the current brk for the executing process, your Yalnix kernel should attempt to grow the stack to "cover" this address, if possible. In all other cases, you should abort the currently running Yalnix user process, but should continue running any other user processes.

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

- `TRAP_MATH`

  This exception results from any arithmetic error from an instruction executed by the current user process, such as division by zero or an overflow. When you receive `TRAP_MATH`, you should abort the currently Yalnix user process, but should continue running any other user processes.

- `TRAP_TTY_RECEIVE`

  This interrupt is issued from the terminal device controller hardware, when a complete line of input is available from one of the terminals attached to the system.

- `TRAP_TTY_TRANSMIT`

  This interrupt is issued from the terminal device controller hardware, when the current buffer of data previously given to the controller has been completely sent to the terminal.

When you abort a process, print a message from your kernel with the process' pid and some explanation of the problem. The exit status reported to the parent process of the aborted process when the parent calls `Wait()` (see 10.2) should be `ERROR`. You can print the message from your kernel with `printf` or `fprintf` in which case it will print out on the UNIX window from which you started your kernel, or (better but not required,) you can use the hardware-provided `TtyTransmit` procedure to print it on the Yalnix console.

The interrupt vector table is stored in memory as an array of pointers to functions, each of which handles the corresponding trap. For each, the name given above is a symbolic index into the corresponding vector (vector == pointer to function) in the vector table. Thus, the vector number should be used as a subscript into the vector table array to set the trap handlers when initializing your kernel. The privileged machine register

`REG_VECTOR_BASE`

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

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

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

See section 6 for a description of the type `struct cpu_context_frame`.

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

# 6 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 (see previous section.)

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

- `int frame_len`: The length (in bytes) of the entire exception frame. The fields defined here form a common header on the exception frame, and are followed by a variable number of bytes of additional hardware information defining the state at the time of the exception. The `frame_len` field defines the total length (including this header) of the cpu context frame. The maximum length of a cpu context frame is defined in `hardware.h` as `MAX_CPU_CONTEXT_FRAME`.
- `int vector`: The vector number of the particular trap, e.g, `TRAP_ILLEGAL`.
- `int code`: A code value giving more information on the particular trap. Its meaning varies depending on the type of trap. In particular, for `TRAP_KERNEL`, it specifies the type of syscall that produced the current trap (see section 10.2,) and for `TRAP_TTY_TRANSMIT` and `TRAP_TTY_RECEIVE` it specifies the terminal number that caused the interrupt.

- `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 10.2.

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

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

# 7 Console and Terminal I/O Handling

The system is equipped with several terminals for use by the user processes executing on the Yalnix kernel. The Yalnix terminals are emulated inside their own X windows unless the `-n` command line switch is passed to the executable (see 12.2)

When reading from a terminal, an interrupt is not generated until a complete line has been typed at the terminal. For writing to a terminal, a buffer (which may actually be only part of a line or may be several lines of output) is given by the process to the hardware terminal controller, and an interrupt is not generated until the entire buffer has been sent to the terminal. The constant `TERMINAL_MAX_LINE` defines the maximum line length supported for either input or output on the terminals.

The current machine configuration supports `NUM_TERMINALS` (constant defined in `hardware.h`), numbered from 0 to `NUM_TERMINALS -1`, with terminal 0 serving as the Yalnix system console, and the others serving as regular terminals. In fact, though, this use is only a convention, and all terminals actually behave in exactly the same way.

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

To write to a terminal from within your kernel (on a `TtyWrite` syscall triggered by a user process) you can use the hardware operation

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

This operation begins the transmission of `len` characters from memory, starting at address `buf`. The address `buf` must be in your kernel's memory, not in the Yalnix user address space (i.e, it must be in virtual memory region 0.) This is to allow context switches to unmap the user process' buffer used in the `TtyWrite` call, even while the `TtyTransmit` proceeds. When the data has been completely written out, you will get a `TRAP_TTY_TRANSMIT` interrupt. Since your kernel is not interruptible, you will not get this interrupt at least until you return out of the kernel into some user process (perhaps the "idle" process.) When the `TRAP_TTY_TRANSMIT` interrupt occurs, the `code` in the cpu context will be set to the number of the terminal that caused the interrupt. You cannot do a second `TtyTransmit()` until you get the `TRAP_TTY_TRANSMIT` interrupt signalling completion of the first.

Input from the terminal works similarly. You will not get a `TRAP_TTY_RECEIVE` interrupt until the user at the Yalnix terminal types a complete line of input (with a `'\n'` at the end). When the interrupt occurs, the `code` field of the cpu context will be set to the number of the terminal that caused the interrupt. You can then get the data line by using the hardware operation:

- `int TtyReceive(int tty_id, char * buf, int len)`

to tell the terminal interface to copy the data of the new input line into the buffer at virtual address `buf`, which must be in the kernel's memory (region 0.) The parameter `len` specifies the length of the buffer. You may always specify `len` to be `TERMINAL_MAX_LINE` in order to simplify your programming task. The actual length of the input line (including the '\n') is returned as the return value of `TtyReceive()`. Thus when a blank line is typed, `TtyReceive` will return a 1. When an end of file character (control-D) is typed, `TtyReceive` returns 0. End of file behaves just like any other line of input, however; in particular, you can continue to read more lines after an end of file. The data copied into your buffer is *not* terminated with a null character, you must use the length returned by `TtyReceive`.

When you receive a `TRAP_TTY_RECEIVE`, you must do a `TtyReceive()` and save the new input line in a buffer inside your kernel, until a user process performs a `TtyRead` syscall (defined in section 10.2.)

You can use `malloc()` in your kernel to manage a queue of terminal input lines that have been received by your kernel but not yet read by a user process. On a `TtyRead()`, if the queue is not empty, just return the next line to the calling user process immediately; otherwise, the calling process should block until the next `TRAP_TTY_RECEIVE` interrupt. For a `TtyWrite()` syscall, you must keep a queue of processes waiting to write to the terminal, and call `TtyTransmit()` for each of them in order. Each of these processes should then be blocked by your kernel until the matching `TRAP_TTY_TRANSMIT` is received by your kernel.

# 8 Bootstrapping Entry Point

In the preceding pages we have talked about how to bootstrap virtual memory, but, where is that done from?

On boot, the hardware starts executing the code in the *Boot ROM.* This firmware knows just enough to read the first sector of the disk, where the kernel is written, and loads the kernel into the bottom of physical memory. It then jumps to the routine:

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

This routine is part of your kernel. It must be written by you, and it is the routine that carries out all the bootstrapping operations described above. This function is discussed further in section 10.3.2.

# 9 Miscellaneous Hardware Operations

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

- `void Pause(void)`

that temporarily stops the emulated CPU until the next trap occurs.

The hardware also provides another instruction to stop the CPU. By executing the

- `void Halt(void)`

hardware instruction, the CPU is completely halted and does not begin execution again until rebooted (i.e, until the Yalnix process is started again from your UNIX terminal.) You should use this instruction to end the emulation and exit from your kernel.

# 10 The Yalnix Kernel

Yalnix user processes call the kernel by executing a trap instruction. To simplify the interface, we provide a library of assembly routines that perform this trap from the user process. This library provides a standard C procedure call interface for the syscalls as described below. The trap instruction generates a trap to the hardware, which invokes your kernel using the

`TRAP_KERNEL` vector from the interrupt vector table. The include file `yalnix.h` defines the interface for all Yalnix syscalls.

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 `yalnix.h`.) The arguments to this call, supplied to the library procedure call interface in C, 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, the value `ERROR` defined in `yalnix.h` should be returned in `regs[0]`.

Yalnix supports the basic process creation and control operations of most flavors of UNIX, such as the syscalls you used for project 1: `Fork()`, `Exec()`, `Wait()`, etc. Yalnix also supports a `Delay()` function and terminal i/o syscalls. Furthermore, Yalnix supports inter-process communication that is very similar to the IPC provided in the V-System, an operating systems research project from Stanford University.

## 10.1 Inter-process Communication

Memory protection is great to protect processes against the bugs of other processes: it isolates each process from each other, affording them the illusion of being the only process on the machine. But it is this very isolation that makes it somewhat difficult for processes to collaborate and/or offer services to each other. When programs trust each other, and collaborate closely, they may be run as different threads within one process: this affords them good communication but requires that they trust each other to follow the rules of usage to synchronize access to their shared memory. It is also possible for programs that only trust each other partially to collaborate while running in separate, memory-protected processes. This is done through what is commonly called "inter-process communication", in which processes declare certain small portions of their address space to be open to other processes through restricted, well-defined mechanisms whose rules are enforced by the operating system. The overhead in this case is higher than the overhead of communicating threads, but allows communication without surrendering most of the benefits of memory protection.

In Yalnix, you will implement, if you have time at the end, a form of inter-process communication (IPC) similar to that implemented by the V-System. IPC is done through request-response messages, in which a process may send a short message to a willing receiver process. The sender then blocks until the receiver sends a short reply message. In Yalnix IPC, the receiver process (and only the receiver) can also copy pieces of its memory space to the memory space of the requester.

An apparently small but thorny issue in all process communication is how to establish the initial connection with another process providing a service. How does a process requiring a service (e.g, a mail handler) know what the process id is of the process providing the service? There are several ways of resolving this issue: for instance, well-known port numbers and mailboxes with well-known addresses. Yalnix allows servers to register themselves with a service number, and it allows IPC to be directed to service numbers rather than to process ids.

We would like to emphasize that you should not implement Yalnix IPC until you have gotten the rest of the kernel to work. The Yalnix IPC syscalls are `Register`, `Send`, `Receive`, `Reply`, `CopyTo` and `CopyFrom`.

## 10.2 The Yalnix syscalls

The following syscalls should be implemented by your kernel:

- `int Fork(void)`

  Create a new child process as a copy of the parent (calling) process. On success, in the parent, the new process ID of the child is returned. In the child, the return value should be 0.

- `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. On failure, this call returns an error code.

  We provide a template of the `LoadProgram` procedure. This procedure reads a program image from an executable file (the file must have been compiled as indicated in the provided `Makefile`,) allocates memory for the new process, and creates its initial stack with the arguments given. This template can be found in `/u0/yalnix/lib/load.template`.

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

  Terminate execution of the current program and process, and save the integer *status* for possible later collection by the parent process on a call to `Wait`. All resources used by the calling process are freed, except for the saved *status*. On success, there is no return from this call, since the calling process is terminated. On failure, this call returns an `ERROR`. When a process exits or is aborted, if it has children, they should continue to run normally, but they will no longer have a parent. When the orphans exit, at a later time, you need not save or report their exit status since there is no longer anybody to care.

- `int Wait(int * status_ptr)`

  Collect the process ID and exit status returned by a child process of the calling program. When a child process `Exits`, it is added to a FIFO queue of child processes not yet collected by its specific parent. After the `Wait` call, the child process is removed from this queue. If the calling process has no child processes (`Exited` or running,) `ERROR` is returned. Otherwise, if there are no `Exited` child processes waiting for collection by this process, 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.

- `int GetPid(void)` Returns the process ID of the calling process.

- `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.

- `int Delay(int clock_ticks)`

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

- `int TtyRead(int tty_id, caddr_t buf, int len)`

  Read the next line from terminal *tty_id*, copying it into the buffer referenced by *buf*. The maximum length of the line returned is given by *len*. The line returned in the buffer is *not* null-terminated. On success, the length of the line in bytes is returned. The calling process is blocked until a line is available to be returned. If the length of the next available input line is longer than *len* bytes, only the first *len* bytes of the line are copied to the calling process, and the remaining bytes of the line are saved by the kernel for the next `TtyRead`. If the length of the next available input line is shorter than *len* bytes, only as many bytes are copied to the calling process as are available in the input line; the number of bytes copied is indicated to the caller by the return value of the syscall.

- `TtyWrite(int tty_id, caddr_t buf, int len)`

  Write the contents of the buffer referenced by *buf* to the terminal *tty_id*. The length of the buffer in characters is given by *len*. The calling process is blocked until all characters from the buffer have been written on the terminal. The return value should be *len* if the `TtyWrite` is successful, or `ERROR` if it is not.

- `int Register(unsigned int` ***`serviceid`***`)`

  Registers the calling process as providing the service *serviceid*. The Yalnix kernel does not enforce that the process actually provides the service, and does not know what service a particular *serviceid* refers to. After a server is registered with `Register`, processes can use `Send` to send messages to it by specifying the service id instead of the process id. `Receive` returns `ERROR` if a service is currently registered by a running process, or if any other error condition occurs.

- `int Send(caddr_t` *`msg`*`, int` ***`pid`***`)`

  This call sends the 32-byte message from address *msg* to the process indicated by *pid*. The calling process is blocked until a reply message sent with `Reply` is received. If *pid* is negative, the message is instead sent to the process currently registered as providing the service with service id $-pid$. On success, the call returns 0; on any error, the value `ERROR` is returned.

- `int Receive(caddr_t` *`msg`*`)`

  This call receives the next message sent to this process with `Send`. If no such message has been sent yet, the calling process is blocked until a suitable message is sent. On success, the call returns the pid of the sending process; on any error, the value `ERROR` is returned.

- `int Reply(caddr_t` *`msg`*`, int` ***`pid`***`)`

  This call sends the reply message *msg* to the process *pid*, which must currently be blocked awaiting a reply from an earlier `Send` to this process.. The reply message overwrites the original message sent (indicated by the *msg* on the sender's `Send` call) in the sender's address space. On success, the call returns 0; on any error, the value `ERROR` is returned.

- `int CopyFrom(int` ***`srcpid`***`, caddr_t` ***`dest`***`, caddr_t` ***`src`***`, int` ***`len`***`)`

  This call copies *len* bytes beginning at address *src* in the address space of process *srcpid*, to the calling process' address space beginning at address *dest*. The process *srcpid* must currently be blocked awaiting a reply from an earlier `Send` to the calling process.. On success, the call returns 0; on any error, the value `ERROR` is returned.

- `int CopyTo(int` ***`destpid`***`, caddr_t` ***`dest`***`, caddr_t` ***`src`***`, int` ***`len`***`)`

  This call copies *len* bytes beginning at address *src* in the address space of the calling process, to the address *dest* in the address space of process *destpid*. The process *destpid* must currently be blocked awaiting a reply from an earlier `Send` to the calling thread. On success, the call returns 0; on any error, the value `ERROR` is returned.

Your kernel should **verify all arguments passed to a syscall**, and should return `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 `TtyRead`, for instance, might well overwrite crucial parts of the operating system which might, in some clever way, gain an intruder access as a privileged user. Also, a pointer to memory that is not correctly mapped or not mapped at all would generate a `TRAP_MEMORY` which would never be serviced because the kernel is not interruptible.

## 10.3 Implementation hints

### 10.3.1 Process Initialization

#### Process IDs

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

**Fork**

On a `Fork`, a new process ID is assigned, the cpu context from the running parent is copied to the child, and new physical memory is allocated into which to copy the parent's memory space contents. Since the CPU can only access memory by virtual addresses (using the page tables) you must map both the source and the destination of this copy into the virtual address space at the same time. You need not map all of both address spaces at the same time, however: the copy may be done piece-meal, since the address spaces are already naturally divided into pages.

**Exec**

On an `Exec`, you must load the new program from the specified file into the program region of the calling process, which will in general also require changing the process' page tables. The program counter in the `pc` field of the cpu context frame must also be initialized to the new program's entry point. A sample C procedure that can be used to do all this is available in `/u0/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`

### 10.3.2 Kernel Initialization

Your kernel begins execution at the procedure `KernelStart`. As already shown in section 8, this function is called from the boot firmware, which expects it to conform to the following prototype:

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

The procedure arguments above are built by the boot ROM and passed to your `KernelStart` routine at boot time. The `cmd_args` argument is a vector (in the same format as `argv` for normal UNIX `main` programs), containing a pointer to each argument from the boot command line (what you typed at your UNIX terminal.) The `cmd_args` vector is terminated by a `NULL` pointer. The `pmem_size` argument is the size of the physical memory in the machine you are running, as determined by the boot ROM. The size of physical memory is given in units of bytes. Finally, the `context` argument is in the format of a cpu context frame (see section 6) although it is built by the boot ROM rather than by the hardware in response to a trap. Your kernel should use this cpu context frame as the basis for other cpu context frames, notably the cpu context frame that starts the initial process at boot time. Processes created from a `Fork`, instead, copy the cpu context frame from the parent process. (Note that in Yalnix, all processes except for the first process started at boot time are created by `Fork`.)

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

- Initialize the interrupt vector entries for each trap, by making them point to the correct subroutines in your kernel.
- Initialize the `REG_VECTOR_BASE` privileged machine register to point to your interrupt vector array.
- Build a structure to keep track of what page frames are free in physical memory. For this purpose, you might be able to use a linked list of physical frames, implemented in the frames themselves. Or you can have a separate structure, which is probably easier, though slightly less efficient. This list of free pages should be based on the `pmem_size` argument passed to your `KernelStart`. The list should of course not include any memory that is already in use by your kernel. The list may or may not include some of the physical frames targetted by the direct hardware translation of addresses in region 0.
- Enable virtual memory
- Create the idle process based on the cpu context passed to your `KernelStart` routine. The idle process is the process that gets scheduled onto the CPU when there is no other process ready. See `Pause` in section 9.
- Create the first process and load the initial program into it. The process will serve the role of the *init* process in UNIX[3]. To run your initial program you should put the file name of the *init* program on your shell command line when you start Yalnix. This program name will then be passed to your `KernelStart` as one of the `cmd_args` strings.
- Return from your `KernelStart` routine. The machine will begin running the program defined by the current page tables and by the values returned in the cpu context frame (values which you have presumably modified to point to the initial context of the initial process.

### 10.3.3 Plan of Attack

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

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

[3]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.

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

3. Write a simple in-kernel idle "process" that prints a message whenever it runs and the `Pause()`'s, which should be once every second as it gets woken up by the clock interrupt. *Getting to here roughly represents completion of Project 2.1.*

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

5. Write the `TRAP_MEMORY` handler.

6. Write an idle program (a single file with a main that loops forever calling `Pause()`.) Write `KernelStart` and `LoadProgram` and use them to get the idle process running. If the idle process runs, you have successfully bootstrapped virtual memory! *Getting to here roughly represents completion of project 2.2.*

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

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

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

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

11. Implement the `Delay` syscall and call it from the init program. Make sure your idle program then runs for several clock ticks uninterrupted. At this point you have your process queues working smoothly and can context switch in at least two places. *Getting to here roughly represents completion of project 2.3.*

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

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

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

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

16. Implement and test the remaining syscalls. The terminal operations and inter-process communication will not be difficult at this point, either, unless you are working on them the night before the deadline.

17. Look at your work and wonder in amazement.

# 11 Your Assignment

To complete this assignment, you need to implement a Yalnix kernel that runs on the hardware defined in this document, and provides the traps defined here. Specifically, you need to provide:

- a `KernelStart` routine to perform kernel initialization,
- a procedure to handle each defined trap.
- a procedure (called by your `TRAP_KERNEL` handler) to implement each defined Yalnix syscall.

# 12 Logistics

## 12.1 Compiling your kernel

The file `/u0/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.

Solaris has several compilers with unclear boundaries. One of them, `/usr/ucb/cc`, is a pitiful attempt to provide a BSD-compatible programming interface (Sun OS was BSD-based) in this most un-BSD-like of systems. Avoid like the plague.

The real Solaris compiler and tools can be found in `/opt/SUNWspro/bin`. Yup, that's the only place these tools are installed. This directory should already be on your default path on `cs230.cs.uchicago.edu`.

The project may also work with `gcc` and `gdb`. However, these tools have not yet been installed (although this situation may change in due course).

You may use, `CC` and/or `g++`, at your peril: the hardware emulation has not been tested with C++. Furthermore, some of the high level features of C++ can actually make life more difficult for you since operating systems tend to operate at a very low level. You may also wish to utilize the following C++ portability guide:

- Don't use templates.

- Don't use exceptions.
- Don't use run time type information (RTTI).
- Don't use namespaces.
- Don't use staticly constructed objects.
- Don't use the C++ library.

That said, you should also know that similar emulators have been used successfully in the past with kernels written in C++.

## 12.2 Running your kernel

Your kernel will be compiled and linked as an ordinary UNIX executable program, and can thus be run as a command from the shell prompt. When you run your kernel, you can put a number of UNIX-style switches on the command line to control some aspects of execution. Suppose your kernel is in the executable file `yalnix`. Then you run your kernel as:

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

For example,

```
yalnix -t -lk 5 -lh 3 -n 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.

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

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

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

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

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

## 12.3 Debugging Your Kernel

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

You may also use the reasonable version of `dbx` that comes with Solaris. I have found that it works satisfactorily. However, because of our emulator's heavy use of signals, it is necessary to tell `dbx` to ignore certain events. You do this by typing

```
ignore 4 11 14 30
```

at the dbx prompt.

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

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

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

## 12.4 Controlling your Yalnix Terminals

By default (unless you specify the option `-n` as specified in section 12.2, or unless you are not running X,) Yalnix terminals are emulated as `xterm`s on your X windows display.

The shell environment variables `YALNIX_TERM`$N$, with $N$ replaced by a number from 0 to `NUM_TERMINALS - 1`, can be set to additional `xterm` arguments so that, for instance, you can specify a Yalnix terminal geometry, font or color different from your X defaults.

The terminal windows keep log files of all input and output operations on each terminal. The files `TTYLOG.`$N$ record all input and output from each terminal separately, and the file `TTYLOG` collectively records all input and output from the four terminals together. The terminal logs show the terminal number of the terminal, either `>` for output or `<` for input, and the line output or input.

### 12.5 Odds and ends

This project will be graded primarily on correctness and efficiency. However, significant lack of documentation or elegance within your code will adversely affect your grade. You should use reasonable variable names and include comments in particularly tricky parts of the code, but you need not document code that does what it looks like.

## 13 Approximate schedule

Please refer to the following dates when planning your time:

- Monday, April 5th. Project 2 handed out.
- Friday, April 16th. You should have formed a project group.
- Friday, April 23rd. Project 2.1 due ("Hello World").
- Friday, May 7th. Project 2.2 due (basic process management, device drivers, context switching, virtual memory).
- Friday, May 21st. Project 2.3 due (system calls, booting to a shell).
- Friday, May 28th. Project 2.4 due (interprocess communication, complete kernel).

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