The UtePC/Yalnix Memory System

This document describes the UtePC memory management hardware subsystem and the operations that your Yalnix kernel must perform to control it. Please refer to Handout 3 for a complete description of the rest of the UtePC hardware platform and Yalnix kernel.

1 Virtual Memory

The UtePCmachine 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, and region 1 is switched into the CPU on a context switch to map the address space of the process about to be run. 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 privileged mode, references to both regions are allowed, but when the CPU is not executing in privileged 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 and changed by the operating system. The regions in the UtePC hardware have locations and sizes fixed in the virtual address space by the MMU hardware.

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*.

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 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 (via a mechanism described below). The size of physical memory is unrelated to the size of the virtual address space of the machine.

Like the virtual address space, 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` happens to be equal to `VMEM_0_BASE` so that the kernel can run whether or not virtual memory is enabled.

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

3 Page Tables

The UtePC 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.

The UtePC hardware knows how to operate on simple arrays of 32-bit page table entries (i.e., it understands flat single-level page tables). Each page table entry contains information relevant to the mapping of a single virtual page. The kernel can allocate page tables wherever it wants, and the tell 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.

When context switching between two processes, you will typically want to manipulate the `REG_PTBR0` register at the very least. The MMU hardware never writes these registers: only the kernel can change virtual-to-physical address mappings. Recall that the values of privileged registers can be accessed using two "instructions" provided to your kernel by the hardware:

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

  Write `value` into the privileged machine register designated by `which`.
- `int ReadRegister(int which)`

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

All privileged machine registers are 32 bits wide.

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.

Because the hardware needs to be able to traverse and interpret page tables, the format of individual page table entries is dictated by the hardware. The file `hardware.h` defines this page table structure in terms of the C type (`struct pte`). 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[1].

To simplify your programming task, we allow you to specify page table locations as virtual addresses, *which should be mapped in region 0*.

4 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].

The UtePC chip-set also contains a TLB. Our 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, PTEs in the UtePC TLB are *not* tagged with a process or address space identifier. Thus, you need to flush the TLB whenever you change a page table entry in memory or change the active page table as part of a context switch; otherwise the TLB might contain stale mappings.

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`.

[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.

5 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 page table registers have been initialized. 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 as follows:

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

After virtual memory is enabled, *all* values used to address memory are interpreted *only* as *virtual* memory addresses by the MMU. Virtual memory cannot be disabled once enabled.

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 `PMEM_BASE` is the same as `VMEM_0_BASE`.

But how do you know how large the kernel is, i.e., what address represents the top of kernel memory within region 0?? The following may be useful for that purpose:

- `sbrk(0)` will return the lowest address not in use by the kernel.
- `&_data_start` is the lowest address used by the data segment. In building the initial page entries for your kernel, the page table entries covering up to (but not including) this point should be initialized with protection set to `PROT_READ|PROT_EXEC`.

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. However, once virtual memory is enabled and user processes are running, the kernel shares physical memory with user processes. Thus, you should use the provided versions of `malloc` and friends, which work correctly even before virtual memory is enabled. This frees you from having to manage physical memory while booting. If you use any mechanism for allocating kernel memory other than the provided `malloc` routines, `sbrk(0)` will not return the correct value, which will likely lead to serious problems..

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 in the form of a routine you must provide, `SetKernelBrk`, which we discuss in Section 6.

6 Process Memory Model and Allocation

Figure 1 illustrates the memory layout of both the kernel and user processes.

In this section we discuss how these address spaces are managed.

User processes

The stack on our architecture grows down from high to low virtual memory addresses. By convention the user stack starts at the top of the user virtual address space (`VMEM_1_LIMIT`) and grows down. We refer to the area containing the stack of a program as the *user stack area*. By convention, the user code, data, and heap is placed at the bottom of the user virtual address space, i.e., starting at `VMEM_1_BASE`. We refer to the area containing the program, data and heap of a program as the *user program area*. The space between the bottom of the stack and the top of the heap is unused, and should be marked as invalid (using the `valid` field of `struct pte`).

We refer to the limit of the program area (i.e., the lowest address not part of the program area) as the user *break*. User process' control the location of the user break using the `Brk` system call that your kernel will implement. *This*

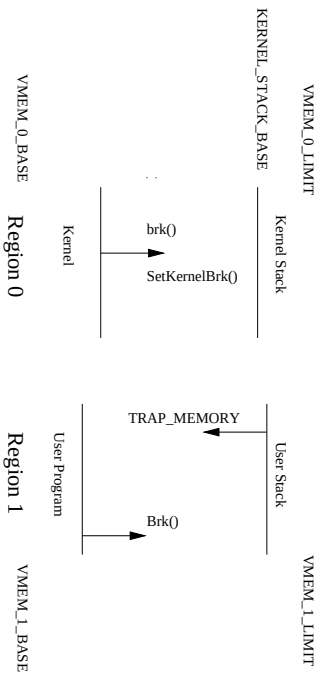

Figure 1 Virtual address space

is the only means by which a user process can cause memory to be allocated or deallocated at the top of the program area. Most user applications do not directly use the `Brk` system call. Rather, they use memory allocation routines from the `malloc` family, which manages a private memory pool of its own, calling `Brk` when it needs additional storage to satisfy a request. We provide versions of `malloc()` (and related functions) that invoke the Yalnix `Brk()` system call in the library with which your user programs are linked by the provided `Makefile`. You do not need to worry about the details of these routines; you need only ensure that `Brk` does the right thing. In summary, a program's heap grows in response to *explicit* requests by user programs, via the `Brk` system call.

In contrast, stack growth occurs in response memory exceptions when the user process accesses addresses below the bottom of the currently allocated stack space. When your kernel receives a TRAP_MEMORY for an address between the bottom of the stack and top of the heap, your kernel should allocate pages between the bottom of the stack and the fault address. The correct way to enlarge a process' stack area is described in more deail in Handout 3.

Kernel

By convention the kernel stack starts at the top of the kernel virtual address space (`VMEM_0_LIMIT`) and grows down. We refer to the area containing the stack of a program as the *kernel stack area*. By convention, the kernel code, data, and heap is placed at the bottom of the kernel virtual address space, i.e., starting at `VMEM_0_BASE` (which is the same as PMEM_BASE). We refer to the area containing the kernel program, data, and heap of a program as the *kernel program area*. The space between the bottom of the stack and the top of the heap is unused, and should be marked as invalid (using the `valid` field of `struct pte`).

As described in Section 5, at boot time the hardware loads the kernel into the kernel program area starting at physical address `PMEM_BASE`. When virtual memory is initialized, the kernel should remain mapped to the same range of virtual addresses, which works because `VMEM_0_BASE` equals `PMEM_BASE`).

Just like user programs, the kernel can allocate dynamic memory using `malloc` and family. In the previous section we noted that the user library version of `malloc` invokes the `Brk` system call when it needs new physical pages at the top of its heap.

However, the kernel version of `malloc` cannot itself invoke a system call. Instead, the kernel needs to implement its own version of `Brk`, called `SetKernelBrk`, which is invoked via a in-kernel function call rather than a system call.

In the library with which your kernel is linked by the provided `Makefile`, we provide versions of the familiar `malloc()` family of procedures that call `SetKernelBrk()` when necessary. `SetKernelBrk()` must be provided by you. The C function prototype of this procedure is:

- `int SetKernelBrk(void * addr)`

This procedure should grow or shrink the space allocated to your kernel so that all addresses between `VMEM_0_BASE` and `addr` are valid kernel addresses. It must allocate and map or unmap pages, as required, to make `addr` the new kernel break. Return 0 on success and -1 on failure. `SetKernelBrk` need not do anything before virtual memory is enabled.

Kernel Stack

In addition to the stack in Region 1 of each user process, the kernel needs a stack *for each process executing on the system*. The kernel uses its stack to store local variables, procedure call linkage state (e.g., return PC), and such not on its stack. By separating the kernel stack for a process from the user stack, the job of the kernel becomes much less confusing, and also much more flexible. This separation also improves the security of the system, since values that may be left in the memory used for the kernel stack for a process are not visible to the user process once executing back in user mode.

As described earlier, the user stack grows by trespassing into unmapped memory, which causes a `TRAP_MEMORY`, which in turns causes the CPU to enter kernel mode. When entering kernel mode, the CPU also switches automatically to the current kernel stack, and stops modifying the user stack for the duration of the trap handler. At this point, the kernel is executing its own instructions on its own stack and is thus free to manipulate the user stack as it sees fit, without risking interfering with the execution of the user process or without confusing its own execution as the kernel.

By having a separate kernel stack for each process, it is possible for he kernel to block the current process while inside the kernel. When performing a context switch, the kernel can then switch to the kernel stack of the process being context switched in, leaving the state of the old process's kernel stack unmodified until that process is run again. That process can thus begin execution again at exactly the point it was at inside the kernel when it last ran. Unlike the user stack for a process, which can grow dynamically, the kernel stack for each process is a fixed maximum size. This avoids the extreme complexity (and contortions) necessary for the kernel to handle its own `TRAP_MEMORY` exceptions and grow its own kernel stack while executing on its own kernel stack. The kernel stack for each process is a fixed `KERNEL_STACK_MAXSIZE` bytes in size.

The kernel stack is located at the same virtual memory addresses for all processes. The kernel stack always begins at virtual address `KERNEL_STACK_BASE` and has a limit address of `KERNEL_STACK_LIMIT`, which is the top of Region 0. Each kernel stack must be stored in different pages of physical memory, but at any given time only one set of these physical memory pages is mapped into this range of virtual addresses. Your kernel must change the virtual memory mappings for the kernel stack on each context switch to map in the new process's kernel stack. Since the kernel stack for each process is located in Region 0, you must change the corresponding page table entries in Region 0's page table as part of the context switch operation.

7 Kernel Context Switch

As mentioned Handout 3, we provide a simple interface (`KernelContextSwitch`) to hide much of the complexity of context switching (register saving and stack manipulation). Using this interface, it is possible to force a context switch from the current process to another process while inside the kernel. For example, suppose some process calls the `Delay` kernel call to delay itself for 5 clock ticks. Using `KernelContextSwitch`, you can block this process inside the kernel (in the middle of `Delay`) and switch to another process. After 5 clock ticks have occurred, you can unblock the process and it will resume exactly where it left off (in the middle of `Delay`), after which it return from the kernel. In the intervening 5 clock ticks, you can run other processes.

To enable this functionality, we provide the function:

```
int KernelContextSwitch(KCSFunc_t *, void *, void *)
```

The type `KCSFunc_t` (kernel context switch function type) is a C typedef of a special kind of function. `KernelContextSwitch` and the type `KCSFunc_t` are defined in hardware.h. The `KernelContextSwitch` function

temporarily stops using the standard kernel context (registers and stack) and calls a function provided by you. For example, suppose you called this function for `KernelContextSwitch` to call in your kernel `MyKCS`. Then you would declare `MyKCS` as:

```
KernelContext *MyKCS(KernelContext *, void *, void *)
```

Since `MyKCS` is called while not using the normal kernel registers or stack, `MyKCS` can safely do what it needs to do to complete the context switch. The two "void *" arguments to `KernelContextSwitch` are passed unmodified to `MyKCS`. You should use them, for example, to point to the current process's PCB and to the PCB of the new process to be context switched in, respectively. In `MyKCS`, you should do whatever you need to do to switch kernel contexts between these two processes. The "`KernelContext *`" argument passed to `MyKCS` is a pointer to a copy of the current kernel context that has been tem- porarily saved somewhere in memory. You should copy this into the old process's PCB. The "`KernelContext *`" pointer returned by MyKCS should point to a copy of the kernel context of the new process to run; this can point to the copy that you earlier saved in that other process's PCB.

The actual `KernelContext` data structure definition is provided in hardware.h. You need not worry what is in a `KernelContext`; just copy the whole thing into the PCB. On a real machine, this structure would be full of low-level hardware-specific stuff; on the UtePC emulator, it is full of low-level Linux and x86-specific stuff.

When you call `KernelContextSwitch`, the current process will be blocked inside the kernel exactly where it called `KernelContextSwitch`, with exactly the state (register contents and stack contents) that it had at that time. (Note: Except for this state, the rest of the system may have changed!) `KernelContextSwitch` and `MyKCS` do the context switch. When this process is later reloaded, its earlier call to `KernelContextSwitch` returns. The return value of `KernelContextSwitch` in this case is 0. If any error occurs, `KernelContextSwitch` does not switch contexts and instead returns -1; in this case, you should print an error message and exit, as this generally indicates a programming error in your kernel.

Note that `KernelContextSwitch` only helps you with actually saving the state of the process and later restoring it. Your kernel must to everything else related to context switching, such as moving a PCB from one queue to another, tracking why the process was blocked, or tracking when it should be reloaded.

Also, note that each process has its own kernel stack in the same place in virtual memory (as shown in Figure 1). You do not need to save or restore the *content* of the kernel stack, but you do need to change the page table entries for the kernel stack to point to the physical pages allocated for the new process's kernel stack.

You can also use `KernelContextSwitch` to just get a copy of the current kernel context. You might find this useful in your implementation of `KernelStart` as part of creating the idle or init process, and in your implementation of the `Fork` system call handler when creating the new child process.

8 Bringing It All Together

This document, in conjunction with Handout 3, provides all of the information you need to create addresses spaces for processes, handle stack growth, and support heap allocation (via `Brk()`). Project 3 (Yalnix Part II) involves adding basic memory management support to your kernel to allow you to dynamically load a second user process (in addition to the idle process) that can perform basic memory operations (stack growth and heap allocation). Project 4 (Yalnix Part III) will involve adding basic process management (`Fork()`, `Exec()`, and `Wait()` and interprocess communication.