# CS 452 Kernel Assignment 3

Eric Woroshow (20165466)
Daniel Zapallow (20203316)

July 2, 2008

## 1 Executables and Instructions

Our kernel and module sources are contained in `/u8/eworoshow/cs452/k3`. The `dekernel`, `test_ee`, and `train` executables (available in the `kernel`, `modules/test`, and `modules/train` directories respectively) provide the functionality required by this assignment.

At boot time the operating system requests a process to run. The "test_ee" module implements the eater/ender test and acts as described in the assignment; its output is explained in Testing. The train control module, "train", acts identically to that implemented in Assignment 1; the same instructions (as outlined in the first assignment description) apply.

## 2 Kernel Design

This section describes the overall design of our kernel. The discussion encompasses three main topics: process management, system calls, and servers.

### 2.1 Processes

A process is the basic unit of multitasking in the operating system. One process executes a single module (ELF file). This section examines memory layout, the process descriptor, process creation, context switching, and scheduling.

#### 2.1.1 Memory Layout

The kernel is structured to take advantage of the full 512MB of memory available on the EOS systems. The kernel code and data is loaded at 1MB, extending upwards towards the top of memory. The module ELF files are loaded contiguously after the kernel. Each process is allocated a fixed-size 2MB block of memory. These blocks extend from 256MB to 512MB, allowing for a maximum of 128 processes. Inside each block the process' data occupies the bottom and its stack grows down from the top. (This configuration potentially allows the stack to corrupt the data section; nothing is done to prevent this error.)

The kernel manages the Global Descriptor Table (GDT) to provide the CPU with knowledge of the memory layout. The kernel code and data segments span the entire range of memory, allowing the kernel to read, write, and execute anywhere it desires. Each process has two associated GDT entries: one for its code segment and one for its data segment. Basic memory protection is enforced via the segment limits. GDT entries are allocated and freed inside the kernel using a simple stack-based free list (allowing for constant-time access).

The kernel never allocates memory for itself dynamically at runtime. Instead, it statically allocates space for its data structures (i.e., inside the data section of the ELF) or uses the stack. In this way the kernel maintains constant-time operations.

### 2.1.2 Descriptor

The process descriptor contains a small set of global data: the process' stack pointer, its code and data GDT descriptors, its state, its priority, its id and its parent's id.

The process state can be one of `Active`, `Ready`, `Blocked`, or `Dead`. An `Active` process is currently executing. A `Ready` process is available for selection by the scheduler and execution. A `Blocked` process is waiting on some event to ready it. A `Dead` process is permanently finished (and its descriptor may be reclaimed).

The process id is divided into two parts: the index and the generation. The index occupies the first 8 bits of the id and indicates the position of the descriptor in the set of descriptors maintained by the kernel. The generation consumes the remaining 24 bits and provides a mechanism to create ids that are unique over time. Every time a descriptor is reused the generation is incremented to ensure two processes that shared the same descriptor have unique ids.

The kernel contains a statically allocated array of 128 process descriptors. The descriptors, like the GDT entries, are allocated and freed using a stack-based free list which contains the indices of all the descriptors that are available for use. Adding and removing from the list is a constant-time operation hence the kernel can create and destroy processes in $O(1)$.

### 2.1.3 Creation

Inside the kernel, creating a process involves three steps: allocating a process descriptor, allocating GDT descriptors for the code and data segments, and initializing its stack. The stack is set up as if the process had trapped into the kernel via an interrupt. This way the process can be run via the common context switch kernel exit code.

Once the process begins executing, the pre-main code from `crt0` is responsible for copying the process' data from the code segment to the data segment. The kernel does not do this work because it is an expensive and non-constant-time operation.

### 2.1.4 Context Switch

The context switch provides the mechanism to enter and leave the kernel.

`KernelEnter`: First the code pushes the interrupt number on to the process stack. Next the contents of the general-purpose and segment registers are pushed on to the process stack.

Then the kernel data segment selectors are restored. The process stack pointer is stored in the active process descriptor. Finally the kernel stack pointer is loaded and the register values are popped off the stack to restore the kernel state. When `KernelEnter` returns, execution continues just after the call to `KernelLeave`.

`KernelLeave`: The enter sequence is performed in reverse to save the kernel state and restore that of the active process. The register contents are pushed on to the kernel stack and the kernel stack pointer is saved. The process stack pointer is then loaded and the process register values are popped off. The `iret` instruction continues execution where it left off prior to the interrupt occurring.

The context switch relies on several assumptions. First, it assumes the machine uses the x86 architecture. Second, it assumes the kernel is always entered via an interrupt. Third, it assumes the kernel data segment selector is constant, allowing it load the segment in order to retrieve the kernel stack pointer.

### 2.1.5 Scheduling

The kernel uses a priority-based round-robin scheduler. Higher-priority processes always execute before those of lower priority. Processes sharing the same priority execute one after another in round-robin fashion.

The scheduler has a mechanism to force a given process to execute next regardless of priority. This facility is used in interrupt handling.

Processes are not time sliced; a process only yields when an interrupt occurs. Both software interrupts (i.e., system calls) and hardware interrupts (e.g., timer) will cause a process to trap into the kernel.

Internally, the scheduler uses a bit map to determine the highest-priority round-robin queue containing processes ready to execute. The queues are implemented using circular arrays. These data structures guarantee the scheduler will execute in constant time.

## 2.2 System Calls

System calls, triggered by software interrupts, provide a mechanism to request services from the kernel.

The system call number and arguments are passed on the process stack. The kernel uses the saved stack pointer in the process descriptor to examine the stack and determine the type of service requested. The return value is given by overwriting the system call number. When what was the call number is popped off the stack its new value is used as the return value from the system call. (These design decisions introduce a tight coupling with the context switch code because the system call module must know exactly where on the stack the arguments are placed relative to the stack pointer.)

Every system call executes in constant time. The overhead of both performing a context switch in to and out of the kernel and scheduling a new process is constant. Moreover, each call has been implemented to execute in constant time.

All system calls potentially yield the CPU to higher-priority processes. Once inside the kernel, the scheduler determines the next process to run and it may not be the process requesting the system call.

### 2.2.1 Create

Creating a new process involves three steps. First, the process module information is looked up by name (the ELF files are preprocessed at boot time). Second, a new process is created using the module information. Third, the new process is scheduled to execute (and thus may preempt the process which called `Create`).

### 2.2.2 MyPid/MyParentPid/MyPriority

These system calls simply return the appropriate value from the active process descriptor.

### 2.2.3 ValidPid

This system call looks up a descriptor based on the index portion of the given process id then verifies the process specified by the descriptor is not dead.

### 2.2.4 Send/Receive/Reply

The `Send`, `Receive`, and `Reply` system calls provide a (blocking) message passing system.

Each receiving process owns a send queue, implemented as a circular array. This queue contains the set of processes in which a `Send` has occurred without a corresponding `Receive`. (That is, the destination process has not yet called `Receive` or has other senders waiting.) A call to `Receive` will pick up the sender's information from the queue and perform the transfer. The kernel services processes on this queue in first-come first-served order so as to avoid starvation.

If a `Receive` occurs before any messages are sent, the receive buffer and its size are stored in an array (with one slot for each possibly receiving process) and the process becomes receive-blocked. A subsequent `Send` will pick up the receive information and process the transfer.

When a `Send` occurs the reply buffer information is also stored in an array (again with one slot per process) so a call to `Reply` knows where to send the reply message.

The kernel imposes a maximum message size restriction of 128 bytes to guarantee that message passing has a known worst-case upper bound and can be said to execute in constant time.

When a process dies we unblock all the processes waiting on its send queue. We also unblock all the processes waiting for its reply. Otherwise these processes would wait indefinitely for a `Receive` or `Reply` that would never arrive. We are also careful to unblock processes should any stage of the S/R/R process fail.

### 2.2.5 AwaitEvent

The `AwaitEvent` system call provides the ability for user processes to respond to interrupts.

When a process calls `AwaitEvent` with a valid IRQ number, first the given IRQ is unmasked in the programmable interrupt controller (PIC). Second a handler inside the kernel is hooked up to the interrupt. The handler uses an array to map IRQs to waiting processes. Finally, the process is blocked until the interrupt occurs.

When the interrupt fires the handler is called to wake up the waiting process. The kernel forces the scheduler to execute the newly-woken process next (ensuring a timely response the interrupt source). The IRQ is then re-masked and the handler unhooked. When the interrupt is handled the process will signal the end-of-interrupt (EOI) to the PIC.

Attempting to call `AwaitEvent` for an interrupt that already has a waiting process is an error; only one process may respond to an interrupt.

## 2.3 Servers

Servers provide services to processes. These services are not accessed via specific system calls but rather via Send/Receive/Reply message passing. A call to `Delay`, for example, is transformed by the system call library to `Send` to the clock server. It follows, then, that servers execute as user processes.

### 2.3.1 Name Server: RegisterAs/WhoIs

The name server provides a process name to process id mapping.

The server uses a coalesced hash map to store the name to id mapping. On average, this data structure provides $O(1)$ insertion and lookup time but the worst-case execution time is $O(n)$ (where $n$ is the number of names in the map). It uses Knuth's hash from the Art of Computer Programming which is, in general, a poor hash but is quick to implement and quick to execute.

User processes communicate with the name server via message passing. A complication arises with this configuration because, in order to `Send` to the name server, a process must know the server's id but `WhoIs` is not available. To compensate the kernel provides two additional system calls: `WhoIsNameserver` and `RegisterAsNameserver`. These calls allow processes to initially bypass the S/R/R mechanism for communicating with the name sever.

### 2.3.2 Clock Server: Delay, GetTime

The clock server provides a delay function to block processes for some number of ticks. The clock server uses a three-process structure: a server, notifier, and courier.

The notifier configures the 8253 timer (the kernel requests a tick frequency of 20 Hz) then calls `AwaitEvent` on the timer IRQ. Upon receiving an interrupt it sends a tick message to the server via the courier. The courier provides a one-tick buffer for the notifier so the notifier is never blocked in the server's send queue.

The server accepts delay and time requests from clients and tick messages from the courier. When the server receives a delay request it inserts the requester's process id and delay time into an array (where open slots are tracked via yet another stack-based free list). For every tick the server receives it iterates through the list of waiting processes and decrements the remaining delay time. When this time reaches zero the server calls `Reply` to wake the process. Every tick also tracks the elapsed milliseconds, minutes, and seconds since the server was started. The `GetTime` call returns these values in a structure.

(A better way to implement the delay would keep the process ids in a sorted order relative to the time they will need to wait. This would allow for insertion in $O(n)$ time but it would mean that each tick the work would be $O(1)$ (i.e., the only work that would need to be done is check the current time against the head of the free list). This has the oposite run times as the method we implemented however since looking at the list is done many more times then the adding to it this method would be more efficent. We did not have time to implement this.)

It should be noted that, with the priority-based scheduler, a call to `Delay` may block for longer than the number of ticks request. The context switch overhead also inflates the amount of time to delay.

### 2.3.3 Serial Server: Get/Put/Read/Write/Config

The serial server provides a mechanism to send and receive data from the serial ports. The serial server uses a six-process structure: a server, notifier, two couriers and two buffers. The server is responsible for constructing all the worker processes.

At the highest level, the server receives `Get` and `Put` requests from clients and internal get and put requests from the couriers. The server maintains both an input and output queue (implemented as circular arrays) to buffer data that has not been read or written yet, respectively. Because `Get` calls block until data is available the server also maintains a queue of getters. As data arrives via the serial port the getters are served in first-come first-served order.

The input courier is responsible for feeding data from the input buffer to the server. Conversely, the output buffer takes data from the server and forwards it to the output buffer. The couriers provide a layer of indirection between the buffers and the server to ensure the buffers are never blocked in the server's send queue (thus blocking the notifier).

The buffers, as suggested by their name, buffer data. The input buffer receives data from the notifier and buffers it until passing it off to the courier (who then passes it to the server). The output buffer works similarly, taking data from the courier and passing it to the notifier. (There is one quirk, however: the output buffer is responsible for sending the data to the serial USART. Because the notifier only signals when the serial hardware is ready to transmit, if no data is available the notifier has no way to later pick up data to transmit.) The serial server uses buffers in addition to couriers to provide a larger event buffer should the serial port be particularly active.

At the lowest level, the notifier interacts with the serial hardware. It initializes the USART then calls `AwaitEvent` to respond to its interrupts. Based on the contents of the

IIR the notifier either signals the output buffer to send data or reads data and passes it to the input buffer. The notifier polls the IIR until all the interrupts are handled to avoid a race condition ( discussed in detail on the newsgroup). Also, the notifier operates with interrupts disabled to prevent stuff.

# 3 Testing

A variety of modules test the kernel:

- The `test_ns` module tests the name server. It tests registration and querying both in a single process and across processes. If the output indicates the values returned from `Create`, `MyPid`, and `WhoIs` match then the test has passed.
- The `test_srr` module tests message passing. It recursively creates worker tasks to calculate the factorial of a number. If the output shows correct factorial calculations then the test has passed.
- The `test_time` module tests the clock server. It calls `Delay` for increasing intervals of time. If the stated delay matches wall-clock time then the test has passed.
- The `test_serial` module tests the serial server. It prints a message to the screen then reads characters from the terminal and echoes them to the screen. If the echo works correctly then the test has passed.
- The `test_ee` module performs a high-level test of the kernel. It tests all of its facilities: process management, message passing, serial communication, names, and timing.

  Its output shows shows the eater processes end in the order 6, 8, 7, 9, 3, 5, 2, 4. The eaters end in this order because of the varying amounts of time they spend executing. The last number eaten by each eater (i.e., $100 - i \mod 100$) indicates the number of seconds spent running (because the delay between each multiple is the multiple itself). The 6 and 8 eaters last eat 96; the 7 eats 98; 9 and 3 eat 99; 5, 2, and 4 eat 100. It follows that the eaters end in the order they do (with the caveat that, within each group of eaters taking the same amount of time to finish, a non-deterministic scheduler may randomize the order within the group).

These modules also indirectly test all the other services provided by the kernel. Should any system call or other internal mechanism fail this would be reflected in a failing test. Hence even if there are no explicit tests for, as an example, process creation, the thoroughness of the other tests compensates.

# 4 Timings

The timings below were obtained using the `test_timings` module. This module uses `GetTime` (from the time server) to track the elapsed time for each of the operations. Because the resolution of the timer is very coarse (one tick every tenth of a second) the module looks at the total elapsed time for running each operation one million times then calculates the average per-operation time. Each timing was then performed multiple times to provide additional data on which to base the following results.

a. The fastest 16-byte Send/Receive/Reply took two (2) microseconds.

b. The fastest 0-byte Send/Receive/Reply took two (2) microseconds. (This value is identical to that of the 16-byte S/R/R because the additional overhead of sending 16 bytes is less than the resolution of the tests.)

c. The `Create` system calls takes five (5) microseconds on average. (We measure the time for a `Create` as the time taken for a child process to be instantiated, run, exit immediately, then be reclaimed. Of course, the time between the system call and the process starting to run is slightly less; an educated guess based on the context switch timings is three (3) microseconds.)

d. The worst-case scenario where interrupts are disabled for an extended period occurs when all the possible interrupts occur virtually simultaneously while the operating system is already executing an expensive operation with interrupts off. In this situation the various server notifiers, with interrupts disabled, must service all the interrupts one by one. The total time with interrupts disabled, then, is the sum of all the service times (plus whatever time was spent with them disabled prior to them firing). The kernel's context switch takes less than one (1) microsecond to execute. The serial notifer may service at most four interrupts each taking (as an extreme overestimate) 10 microseconds. The time notifier may take up to 5 microseconds. With two serial notifiers running, one for the terminal and one for the trains, and one time notifier, we can establish an upper bound of 100 microseconds with interrupts disabled.