| Date | Changes |
|---|---|
| May 30, 2008 | Initial skeleton (rewrite from old design). Discussion of basic abstractions. |
| June 3, 2008 | Filled-in HAL details, mostly copied from old design document. |
| June 17, 2008 | Basic discussion of the I/O Manager and messaging details. |
| June 18, 2008 | More discussion around scheduling + context switches. Filled-in the IDT contents. |
| June 20, 2008 | Filled-in Thread Manager details. |
| June 21, 2008 | Clarified some of the I/O Manager scheduling logic. Added section on Mailbox Manager. |
| August 26, 2008 | Various updates to reflect design changes. Removed support for multiple mailboxes per thread. Removed the Mailbox Manager subsystem. Added section on dedicated kernel threads. Miscellaneous other cleanup. |
| September 12, 2008 | Added some details on the different memory pools/heaps; and some basic discussion of the address space layout. |
| September 30, 2008 | Expanded discussion of address space management. Added section on memory allocation operators. |
| December 23, 2008 | Removed references to the boot heap. Corrected some memory management + address space details. Added discussion of TSS, page frame management, shared frames, boot sequence and ramdisk. |
| January 30, 2009 | Minor cleanup to the section on memory management. Clarified the requirement for kernel threads vs. dedicated address spaces. Clarified some shared-frame and copy-on-write behavior. |
| | Misc cleanup. Corrected some interrupt vectors. Added some details on system calls. Fixed some code references. Removed some unnecessary implementation details. |
| August 5, 2009 | Cleaned up section on interrupt management. Added discussion of Device Proxy subsystem. Corrected some of the details around address space layout. |
| August 11, 2009 | Added section on user libraries + system calls. Added various figures. Minor cleanup. |

# 1 Overview

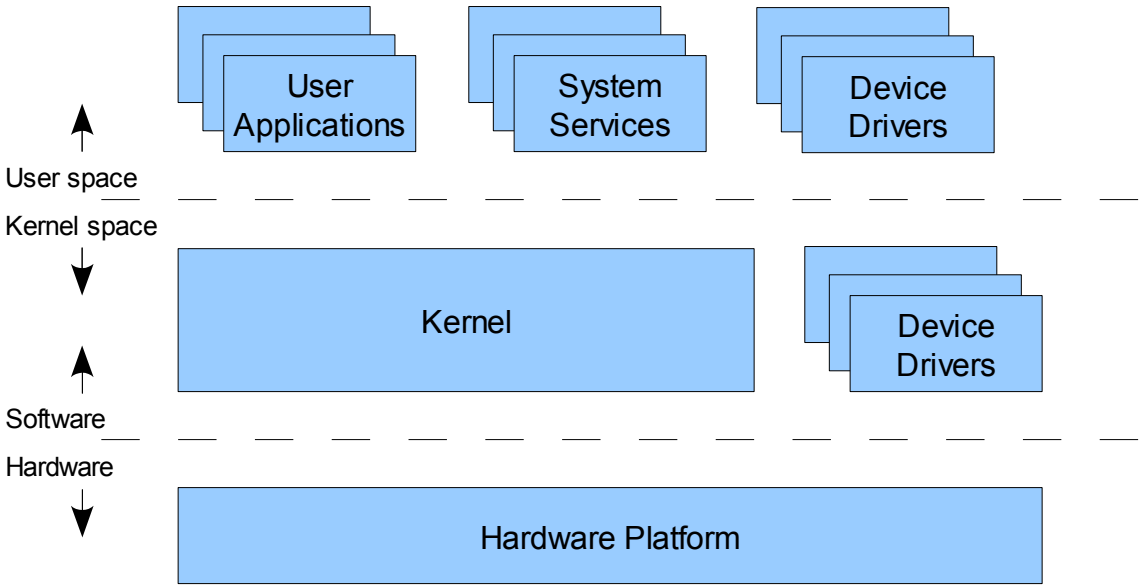
This document describes the design and implementation of the DX operating system. More specifically, it covers: the underlying OS kernel; basic hardware management; kernel-user interaction; and key user-level components.

## *1.1 System Overview*

The DX OS implements a fairly "typical" design:

- A micro-kernel in kernel space.
- A variety of system services executing in user space.
- Various user applications also executing in user space.

The kernel includes a few device drivers for configuring some of the key hardware components; but the majority of drivers live in user space.



*Drawing 1: Basic system decomposition*

# 2 Kernel

## *2.1 Overview*

DX is built around a message-passing micro-kernel. Different threads of execution communicate with one another by exchanging "messages". This messaging facility is the core of the OS; and simultaneously provides the mechanisms for communication, scheduling and synchronization.

The kernel provides only message-passing and basic hardware management; all other system functionality is pushed out to user-level daemons. More specifically, the kernel provides:

- Low-level hardware management
- Message-passing facilities
- Scheduling
- Address space management + protection

## *2.2 Basic Abstractions*

The DX kernel supports three basic abstractions:

- Address spaces
- Messages
- Threads

### 2.2.1 Address Spaces

An address space (class `address_space_c`) is the basic memory container and the basic memory protection mechanism. Threads, messages and other resources are all inherently "contained" within an address space. Messages are the only supported form of interaction/communication between address spaces.

An address space has the following attributes:

- A unique, 32-bit id. An address space id uniquely identifies one specific address space within the system. Address space ids are globally visible (i.e., visible to all threads in all address spaces). No two address spaces ever simultaneously share the same id.
- A hardware-defined Page Directory/Table hierarchy.
- Possibly, a hardware-defined I/O bitmap. This bitmap, if present, specifies which I/O ports are available to threads within this address space (see section 2.3.1.4). This bitmap is typically only required for device drivers running in user space..
- A reference count.
- A list of shared frames – physical page frames, mapped into this address space, that are also visible (shared) from other address spaces.
- A pool of blocks for mapping the payloads of incoming messages.

### 2.2.2 Messages

A message (class `message_c`) is the primary unit of communication. Threads within the same address space may communicate either via messages or via shared-memory, but all cross-address space communication is built on top of the message facility.

Each message is comprised of:

- A handle to source thread. This is essentially the "return address" of the thread that sent the message; and allows the recipient to reply directly back to the sender, if necessary.
- A handle to the destination thread. This is the thread to which the message should be delivered.
- The message type. This allows the recipient to determine the format of the payload, if any.
- Some control information. The control data indicates:
    - Whether the sender is waiting for a response to this message
    - An arbitrary sequence number for matching requests + responses

The kernel supports three message subtypes:

- "Small messages" (class `small_message_c`) contain only a single word of payload data.
- "Medium messages" (class `medium_message_c`) contain up to 128 bytes of payload data.
- "Large messages" (class `large_message_c`) contain one or more pages of payload data.

### 2.2.3 Threads

A thread (class `thread_c`) is the basic unit of execution and scheduling. A thread may execute exclusively in kernel-space; exclusively in user-space; or a combination of both.

Each thread has the following attributes:

- An address space. Each thread is contained within a specific address space which provides the memory context for that thread. An address space may contain more than one thread; but each thread has exactly one parent address space.
- A unique, 32-bit id. A thread id uniquely identifies one specific thread within the system. Thread ids are globally visible (i.e., visible to all threads in all address spaces). No two threads ever simultaneously share the same thread id.
- A queue of incoming messages (the thread's "mailbox").
- A mask of capabilities (permissions).
- A kernel-mode stack. This is the stack used when the thread is operating in kernel-space.
- A reference count.

- A pre-allocated page for supporting copy-on-write. If/whenever the thread takes a copy-on-write fault, it can use this page to temporarily map the contents of the faulting page.

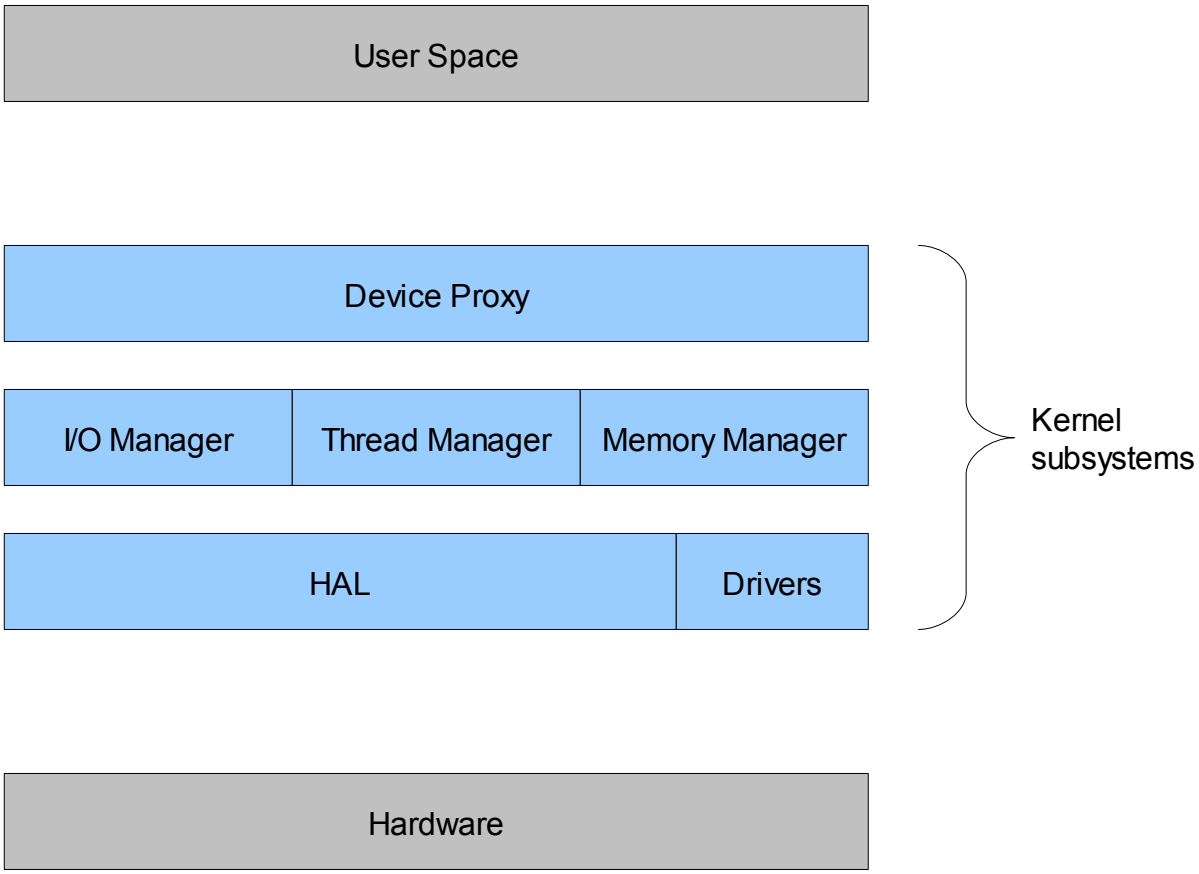
## 2.3 Kernel Subsystems

The kernel is composed of five subsystems:

- The HAL provides basic hardware management + hardware-specific primitives.
- The I/O Manager provides message-passing and scheduling.
- The Thread Manager creates + destroys threads.
- The Memory Manager provides basic in-kernel memory allocation; and address space management.
- The Device Proxy exposes hardware resources out to user mode drivers.

The subsystems are organized into three layers of functionality, mainly for simplicity and clear separation of concerns. A kernel subsystem may invoke any other subsystem at its own level; any subsystem "beneath" it; but may not invoke subsystems "above" it. For example, the I/O Manager can invoke any other subsystem except the Device Proxy.

.




*Drawing 2: Basic kernel decomposition + subsystems*

### 2.3.1 Hardware Abstraction Layer (HAL)

| | |
|---|---|
| Subsystem | HAL |
| Purpose | Lowest layer of the kernel. Hides low-level hardware, processor and system details. Encapsulates various x86 quirks and x86 registers/structures. Provides portability between uni- and multi-processor systems. |
| Handles interrupts | NMI (x86 vector 2) |
| | Device not available (x86 vector 7) |
| | Co-processor overrun (x86 vector 9) |
| Handle system calls | None |
| Kernel interfaces | Management of x86-specific structures (e.g., the GDT, IDT, TSS, etc). |
| | Low-level thread initialization and context switching |
| | System/CPU management (e.g., reboot, shutdown, suspension, etc.) |
| | Low-level memory management (e.g., manipulation of page tables, etc). |
| | Synchronization primitives (e.g., spin-locks) |
| Spinlocks | None |

#### 2.3.1.1 Global Descriptor Table

The HAL is responsible for initializing the Global Descriptor Table (GDT). The kernel uses a "flat" memory model and stack-based context switching (see section 2.3.1.5), and therefore the GDT only contains a handful of entries. The HAL does not provide or define any LDT.

The contents of the GDT are described below. Unused GDT entries are marked as "not-present" and are omitted from the table below. All code descriptors are marked as non-conforming, execute-only, 32-bit segments with a base address of zero and a limit of 4GB (0xFFFFFFFF). All data descriptors are marked as read-write, 32-bit segments with a base address of zero and a limit of 4GB (0xFFFFFFFF).

| GDT Index | Purpose | Descriptor Privilege Level | Associated selector |
|---|---|---|---|
| 0 | NULL | N/A | 0x00 |
| 1 | Kernel code | 0 | 0x08 |
| 2 | Kernel data | 0 | 0x10 |
| 3 | User code | 3 | 0x18 @RPL? |
| 4 | User data | 3 | 0x20 @RPL? |
| 5-7 | Unused | | |
| 8 | Task/TSS for CPU0 | 0? @@ | N/A |
| 9-15 | Unused | | |

#### 2.3.1.2 Interrupt Descriptor Table

The HAL is also responsible for initializing the Interrupt Descriptor Table (IDT). The contents of the IDT are listed below. Unused IDT entries are marked as "not-present" and are omitted from the table below. The IDT contains only interrupt and trap gates; task gates are not used. All descriptors are 32-bit gates and refer to the kernel code selector. All interrupt gates have a Descriptor Privilege Level (DPL) of zero; all trap gates have a DPL of 3.

| Entry | Source/Cause | Gate type | Handler on this entry |
|---|---|---|---|
| 0 | 80x86 Divide Error | Interrupt | |
| 1 | 80x86 Debug Step | Trap | |

| | | | |
|---|---|---|---|
| 2 | 80x86 NMI | Interrupt | HAL |
| 3 | 80x86 Breakpoint | Trap | |
| 4 | 80x86 Overflow (INTO) | Trap | |
| 5 | 80x86 Bounds Check | Trap | Memory Manager |
| 6 | 80x86 Invalid Opcode | Interrupt | HAL |
| 7 | 80x86 Device not available | Interrupt | HAL |
| 8 | 80x86 Double Fault | Interrupt | HAL |
| 9 | 80x86 Coprocessor overrun | Interrupt | HAL |
| 10 | 80x86 Invalid TSS | Interrupt | HAL |
| 11 | 80x86 Segment Not Present | Interrupt | Memory Manager |
| 12 | 80x86 Stack Fault | Interrupt | Memory Manager |
| 13 | 80x86 General Protection Fault | Interrupt | Memory Manager |
| 14 | 80x86 Page Fault | Interrupt | Memory Manager |
| 15 | Reserved | N/A | |
| 16 | 80x86 Floating Point Error | Interrupt | HAL |
| 17 | 80x86 Alignment Check | Interrupt | |
| 18 | 80x86 Machine Check | Interrupt | HAL |
| 19 | 80x86 SIMD Floating Point Error@@@ | Interrupt | |
| | | | |
| 32 | PIC IRQ0 (timer) | Interrupt | I/O Manager |
| 33 | PIC IRQ1 | Interrupt | Device Proxy |
| 34 | PIC IRQ2 | Interrupt | Device Proxy |
| 35 | PIC IRQ3 | Interrupt | Device Proxy |
| 36 | PIC IRQ4 | Interrupt | Device Proxy |
| 37 | PIC IRQ5 | Interrupt | Device Proxy |
| 38 | PIC IRQ6 | Interrupt | Device Proxy |
| 39 | PIC IRQ7 | Interrupt | Device Proxy |
| 40 | PIC IRQ8 | Interrupt | Device Proxy |
| 41 | PIC IRQ9 | Interrupt | Device Proxy |
| 42 | PIC IRQ10 | Interrupt | Device Proxy |
| 43 | PIC IRQ11 | Interrupt | Device Proxy |
| 44 | PIC IRQ12 | Interrupt | Device Proxy |
| 45 | PIC IRQ13 | Interrupt | Device Proxy |
| 46 | PIC IRQ14 | Interrupt | Device Proxy |
| 47 | PIC IRQ15 | Interrupt | Device Proxy |
| | | | |
| 64 | Yield | Interrupt | I/O Manager |
| | | | |
| 80 | Receive Message | Trap | I/O Manager |
| 82 | Send + Receive Message | Trap | I/O Manager |
| 82 | Send Message | Trap | I/O Manager |
| 83 | Delete Message | Trap | I/O Manager |
| | | | |
| 90 | Contract Address Space | Trap | Memory Manager |
| 91 | Create Address Space | Trap | Memory Manager |
| 92 | Delete Address Space | Trap | Memory Manager |
| 93 | Destroy Address Space | Trap | Memory Manager |
| | | | |
| 100 | Create Thread | Trap | Thread Manager |
| 101 | Delete Thread | Trap | Thread Manager |
| | | | |
| 110 | Map Device | Trap | Device Proxy |
| 111 | Unmap Device | Trap | Device Proxy |

### 2.3.1.3 Interrupt Management

The HAL provides the necessary infrastructure for catching and dispatching all interrupts, traps, system calls, etc[1]. Whenever the CPU takes an interrupt, the HAL invokes the kernel component associated with that interrupt vector; see `::dispatch_interrupt()`.

#### *2.3.1.3.1 Shared Interrupts*

The HAL does not support shared interrupts. There is (at most) one kernel handler per interrupt vector. However, the kernel handlers themselves may allow for interrupt sharing on their interrupt vectors. In particular, the Device Proxy (section 2.3.6) allows multiple user-mode drivers to listen on the same interrupt vector simultaneously.

#### *2.3.1.3.2 Interrupt Context*

When the CPU takes an interrupt, the HAL invokes the kernel handler directly. The handler is thus executing in the context of the interrupted thread.

There are two flavors of interrupt handlers:

- Handlers attached to interrupt gates (essentially: handlers for external hardware devices and Intel-defined "fault" vectors; see section 2.3.1.2) execute in some arbitrary thread context. The handlers cannot make any assumptions about the address space in which they are executing. Interrupts are disabled on the local CPU during interrupt processing; and therefore the handlers are not subject to preemption. These handlers may send messages – either blocking[2] or non-blocking – but should not sleep indefinitely waiting for incoming messages.
- Handlers attached to trap-gates (essentially: all system-calls) always execute within the context of the thread that issued the request. Interrupts are still enabled, so that these handlers are subject to preemption. These handlers have full access to the I/O Manager – they may send and receive messages as necessary.

All handlers have full access to the Memory Manager. They may allocate or free kernel memory as necessary.

### 2.3.1.4 TSS Management

Although the kernel uses stack-based context switching (see section 2.3.1.5), the HAL still manages a TSS for two reasons:

- The TSS specifies the kernel context/stack to use when a thread executing at ring-3 takes an interrupt. This usage is required by the IA32 architecture.
- The TSS contains an I/O bitmap that allows ring-3 threads access to the I/O space. This usage is not strictly required, but allows device drivers to execute at ring-3.

The HAL supports one TSS per CPU. On a context switch, the HAL updates the TSS (for the local CPU only) to reflect the kernel context and I/O bitmap for the new thread. In addition, the HAL updates the TSS on the fly if/when a thread manipulates its I/O bitmap via either the MAP_DEVICE or UNMAP_DEVICE system calls.

---

[1] The Intel documentation distinguishes among device interrupts; software exceptions; system-calls; etc.; although we generically use the term "interrupt" to cover all of these cases.

[2] An interrupt handler may send a blocking message to a separate thread. The destination thread then executes within the same boundaries as the interrupt handler – not subject to preemption; should not sleep waiting for input messages.

#### 2.3.1.5 Switching Contexts

The I/O Manager is responsible for determining when a context switch is necessary; and which thread gains the CPU as the result of the context switch. The HAL is responsible for actually implementing the context switch. All context switches are assumed to occur within either the IRQ0 interrupt path or the YIELD soft-interrupt path (see section 2.3.3.6).

The HAL method `switch_thread()` implements stack-based context switching. The victim thread triggers the context switch by invoking this method. Conceptually, the context switch requires two basic steps:

1. The "suspending-the-victim-thread" stage. In this step, the HAL saves the execution state of the victim thread on its kernel stack.
2. The "resuming-a-new-thread" stage. Here, the HAL resumes the next thread by popping its last execution context from its stack, which enables that thread to return from `hal::switch_thread()` as if it had never been suspended.

Typically, a thread transparently executes both stages of the context-switch logic whenever it loses and regains the CPU, respectively – it first gives up the CPU and then later regains it. However, thread-creation and thread-exit both rely on the context switch mechanism in non-obvious ways:

- A newly-created thread (see 2.3.5.1) begins execution within the middle of `hal::switch_thread()`, as if it were an existing thread that had been suspended and is now waking up (i.e., it looks like a suspended thread to the HAL). But since the thread is just starting, it executes only the resumption stage of the normal context-switch logic without ever executing the suspension stage.
- A thread that exits gracefully (see section 2.3.5.3) eventually invokes `hal::switch_thread()` as if it were being suspended like any other thread. But since the thread is exiting, it never resumes from this point. Thus it behaves the opposite of the newly-created thread: it executes only the suspension stage without ever executing the resumption stage.

### 2.3.2 Built-in Drivers

Although most device drivers execute out in user-space, the kernel includes a number of built-in drivers for managing some of the key hardware components:

- PIC (8259 Programmable Interrupt Controller) driver. Responsible for initializing the local PIC.
- PIT (8254 Programmable Interval Timer) driver. Responsible for initializing the system clock.
- VGA driver. Writes kernel messages out to the display.
- Serial port/console driver. Writes kernel debug messages out to the serial port. Only included in the debug build.

### 2.3.3 I/O Manager

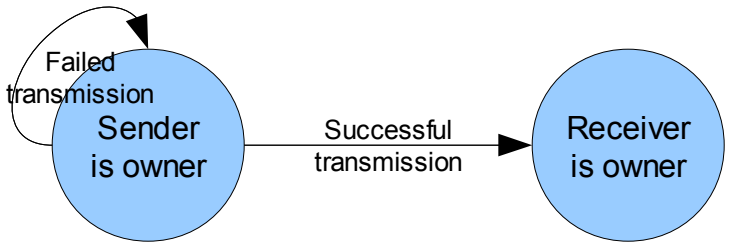
| Subsystem | I/O Manager |
|---|---|
| Purpose | Provides message-passing (IPC) facilities and scheduling. |
| Handles interrupts | IRQ0 (clock tick) |
| Handles system calls | Yield |
| | Receive Message |
| | Send + Receive Message |
| | Send Message |
| | Delete Message |
| Kernel Interfaces | Send message |
| | Receive message |
| Spinlocks | One lock, protects the pool of pending messages. @@also protects the topology of dependencies between blocked threads, although this is weak |

#### 2.3.3.1 Message Lifecycle

Each `message_c` object is conceptually owned by only a single thread[3]. However, the owner of a message changes as the message is transferred from sender to recipient. In general, only the current owner may safely access or modify the message contents. The message owner is specifically responsible for destroying the `message_c` object when it is no longer needed.

At any given point, a message is owned by exactly one of the following threads:

- The sending thread. This is the thread that initially created the message. The sending thread owns the message until it explicitly sends the thread to the recipient. If the transmission succeeds (i.e., if `io_manager_c::send_message()` returns successfully), then ownership of the message transfers to the recipient. If transmission fails, then the sending thread continues to own the message and is responsible for message cleanup.
- The recipient. This is the thread that actually receives the message. Once it retrieves the message from its mailbox, the recipient thread becomes the message owner. When this thread is finished with the message, it is responsible for message cleanup.




*Drawing 3: Message ownership*

#### 2.3.3.2 Blocking and Non-blocking Messages

The I/O Manager supports two flavors of messages:

- Blocking/synchronous messages. These are messages that require a response before the sender can continue executing. The sender blocks until it receives a response from the recipient.
- Non-blocking/asynchronous messages. These are messages that do not require a response; or at least: do not require an immediate response. The sender posts this message to the recipient and

[3] This applies only to the `message_c` object itself. In the case of the external payload associated with `large_message_c`, the payload data is still visible to both threads.

then continues executing without waiting for a response. The recipient may respond at some later point, but the sender is not blocked waiting for this reply.

Although the I/O Manager supports both models, applications are free to use either style (or both) as needed. Usage of blocking or non-blocking messages is essentially a policy decision left to the application.

The blocking/non-blocking distinction is a property on all `message_c` objects. In other words, any `message_c` instance (`small_message_c`, `medium_message_c` or `large_message_c`) may be marked as blocking if necessary.

#### 2.3.3.3 Message Transmission

A thread initiates a message transfer by invoking the I/O Manager. User-level threads do this via system calls (SEND_MESSAGE and SEND_AND_RECEIVE_MESSAGE); kernel threads can call the I/O Manager directly.

In either event, the sending thread eventually invokes `io_manager_c::send_message()` with the message to be transmitted. Execution continues within the context of this thread (i.e., the message-passing logic executes in the context of the sending thread). After validating the message, the I/O Manager queues it on the mailbox for the destination.

At this point, if the sending thread is issuing a blocking request, then the I/O Manager immediately suspends the thread and transfers control of the CPU directly to the recipient (see sections 2.3.3.2 and 2.3.3.6 below). The sending thread remains blocked here until the recipient wakes it by responding to the original request. The sending thread then resumes execution with the response in hand.

If instead the originating thread is issuing a non-blocking transfer, then the I/O Manager simply returns and the thread continues execution normally. The recipient thread eventually reads the message from its input queue and dispatches it as necessary. The originating thread does not know when the recipient receives the message or how it handles the message, unless the recipient issues an explicit response.

If the I/O Manager decides the message cannot or should not be delivered, it immediately returns an error to the calling thread. The calling thread is allowed to continue executing and is responsible for cleaning up the failed message.

#### 2.3.3.4 Message Receipt

A thread receives a message by invoking the I/O Manager. Like the transmission path, user-level threads accomplish this using system-calls (RECEIVE_MESSAGE and SEND_AND_RECEIVE_MESSAGE), while kernel threads can call the I/O Manager directly.

In either event, the thread eventually invokes `io_manager_c::receive_message()` and execution continues within the context of this thread (i.e., the message-reception logic executes in the context of the receiving thread).

If the receiving thread already has one or more messages pending in its mailbox, the I/O Manager de-queues the first (oldest) pending message and returns it the calling thread. The receiving thread then continues execution without blocking, now with the incoming message in hand.

If the receiving thread does not have any messages pending in its mailbox, then the I/O Manager suspends the thread until a new message arrives[4]. The I/O Manager wakes this thread when a new message arrives in its mailbox, and the thread resumes execution with the new message in hand.

[4]Alternately, the calling thread can issue a non-blocking request, in which case it just receives an error when its mailbox is empty but continues executing without waiting.

As a special case of message-reception, if a thread is sleeping while waiting for a response to a blocking message, the response message automatically jumps to the front of the mailbox queue when it arrives. This ensures that when the thread resumes, the first message it receives is always the response to its original request.

### 2.3.3.5 Delivery of Message Payloads

The different message types (see section 2.2.2) have different delivery requirements:

- "Small messages" have a single word of payload. The kernel just passes these words verbatim across the system call interface.
- "Medium messages" have payload blocks contained within the `medium_message_c` object itself[5]. The kernel delivers these payloads through simple memory copies: the payload is copied from the sender's address space to a temporary kernel buffer on transmission; and copied from the kernel buffer into the destination address space on reception.
- "Large messages" are associated with arbitrary-size payload blocks, possibly spanning multiple pages of the sender's address space. Instead of copying the payload data, the kernel uses shared page frames to transfer these payloads between address spaces. See 2.3.4.7.

### 2.3.3.6 Scheduling

The I/O Manager determines scheduling policy – when a thread should be preempted and which new thread should gain the CPU in its place. It relies on the HAL to actually implement the context switch (see section 2.3.1.5).

The I/O Manager makes all of its scheduling decisions within the context of either the IRQ0 (clock tick) interrupt or the YIELD soft-interrupt[6]. See `io_manager::handle_interrupt()` and `io_manager::select_next_thread()`. When handling one of these interrupts, if the I/O Manager decides to allocate the CPU to another thread, it has three possible scheduling options:

- If the current thread yields because it's blocked, waiting for a response from some other thread, then the I/O Manager automatically passes the CPU to this second thread (see section 2.3.3.3).
- If the current thread yields the CPU without blocking on some other thread, the I/O Manager holds a lottery to determine which thread gains the CPU. It selects a message at random from the set of pending messages (i.e., the collection of messages that are pending in mailboxes, not yet retrieved by their owners) and allocates the CPU to the destination/recipient thread. In effect, each pending message is one lottery "ticket"; and the lottery "winner" is the thread to which that message was sent.
- If the current thread yields without blocking, but there are no messages currently pending, the I/O Manager dispatches the Idle Thread to consume CPU cycles until a message arrives. In this situation, no lottery is possible since there are no outstanding lottery tickets (i.e., messages) so the Idle Thread wins the lottery by default. See section 2.4

If the lottery winner is already blocked on another thread, then the I/O Manager passes the CPU onto this other thread instead, under the assumption that the second thread will eventually wake the original winner. This procedure potentially skips over multiple threads if one thread is blocking the progress of two or more

---

[5] Currently, the largest possible "medium payload" is 128 bytes. Payloads that exceed this limit require a `large_message_c`.

[6] As a side-effect of this behavior, disabling interrupts effectively prevents the I/O Manager from executing and therefore disables preemption unless a thread explicitly yields the CPU. So for example, handlers attached to interrupt gates are not subject to preemption; while handlers attached to trap gates may be preempted (see section 2.3.1.2).

threads. For example, if thread A is blocked waiting for a message from thread B; and thread B itself is blocked waiting for a message from thread C; then thread C gains the CPU if thread A wins the lottery, because thread C is ultimately preventing thread A from executing

## 2.3.4 Memory Manager

| Subsystem | Memory Manager |
|---|---|
| Purpose | Manages the various address spaces. Manages the kernel runtime heap. Manages physical memory. |
| Handles interrupts | Bounds check (x86 vector 5) |
| | Segment not present (x86 vector 11) |
| | Stack fault (x86 vector 12) |
| | General Protection Fault (x86 vector 13) |
| | Page fault (x86 vector 14) |
| Handles system calls | Contract Address Space |
| | Create Address Space |
| | Delete Address Space |
| | Expand Address Space |
| Kernel interfaces | Address space creation, tracking, deletion. |
| | Allocation and deletion of memory in the kernel heap. |
| | Allocation and deletion of physical memory |
| Spinlocks | One lock to protect the table of address spaces (within the Address Space Manager) |
| | One lock per Address Space to protect its Page Directory and Shared Frame Table. |
| | One lock per Memory Pool in the Kernel Heap. |

### 2.3.4.1 Division of Labor

The various memory management tasks are split between a kernel-mode component (the "kernel-mode Memory Manager" or "KMM") and a user-mode component ("user-mode Memory Manager", "UMM"). The UMM is largely responsible for dictating memory management policy; the KMM provides the underlying mechanisms.

More specifically:

- The KMM handles all memory requests for the kernel itself. The KMM also handles page table manipulation; and manages physical memory allocation.
- The UMM tracks the layout of the various address spaces, handles most page faults, makes working-set decisions, and implements paging.

### 2.3.4.2 Kernel Heap

The kernel uses a memory heap for handling dynamic memory allocation. Most kernel systems interact with the heap only through the `new()` and `delete()` operators.

The heap is subdivided into eight pools, which contain memory blocks of 8, 16, 32, 64, 128, 1024, 4096 and 8192 bytes, respectively. All blocks are "naturally aligned" (e.g., the 8-byte blocks are aligned on 8-byte boundaries, etc.). If a thread allocates a chunk of memory that is not an integral block size, the Memory Manager promotes the request to the next-largest block size.

@not currently used, but kernel components can create their own pools if necessary

### 2.3.4.3 Virtual Address Space Layout

All address spaces share the same basic layout:

| Virtual addresses | Usage |
|---|---|
| 0x00000000 – 0x003FFFFF | Kernel code/image at 2MB, followed by first portion of ramdisk. Kernel only. Non-paged. Identity-mapped with a single superpage, so it (physically) spans the BIOS data area, Multiboot structure, the ISA hole and the VGA buffer. Managed by KMM. |
| 0x00400000 – 0x007FFFFF | Reserved for ramdisk. Kernel only. Non-paged. Identity-mapped with a single super-page. Managed by KMM. |
| 0x00800000 – 0x00BFFFFF | Context for boot thread. GDT, IDT and TSS. Kernel heap. Kernel only. Non-paged. Identity-mapped with a single super-page. Managed by KMM. |
| 0x00C00000 – 0x1FFFFFFF | Currently unused. Reserved for kernel. |
| 0x20000000 – 0x3FFFFFFF | Message payload area. The payload of each incoming message is mapped into some virtually-contiguous portion of this range. Visible to user-threads. Paged. Managed by KMM. |
| 0x40000000 – 0xFFFFFFFF | User space. Visible to user threads (obviously). Paged. Managed by UMM. |
| 0xFFC00000 – 0xFFFFFFFF | Environment block for address space. Contains process parameters, run-time settings for current address space. Managed by UMM. |

### 2.3.4.4 Address Space Manager

Within the Memory Manager subsystem, a dedicated Address Space Manager handles the details of creating, tracking + deleting address spaces. This Address Space Manager is private to the Memory Manager; it is not exposed to other kernel systems and exposes no public kernel interfaces.

### 2.3.4.5 Physical Address Space Layout

The layout of the physical address space is simpler:

| Physical addresses | Usage |
|---|---|
| 0x00000000 – 0x00BFFFFF | The kernel image and data is identity-mapped here, so usage is dictated by the virtual address space layout (see section 2.3.4.3). |
| 0x00C00000 – 0xFFFFFFFF | No specific purpose. May be subdivided and allocated to any address space as needed. |

### 2.3.4.6 Page Frame Manager

Within the Memory Manager subsystem, a dedicated Page Frame Manager handles the allocation and recycling of physical page frames. Like the Address Space Manager, the Page Frame Manager is private to the Memory Manager and is not exposed to other kernel systems.

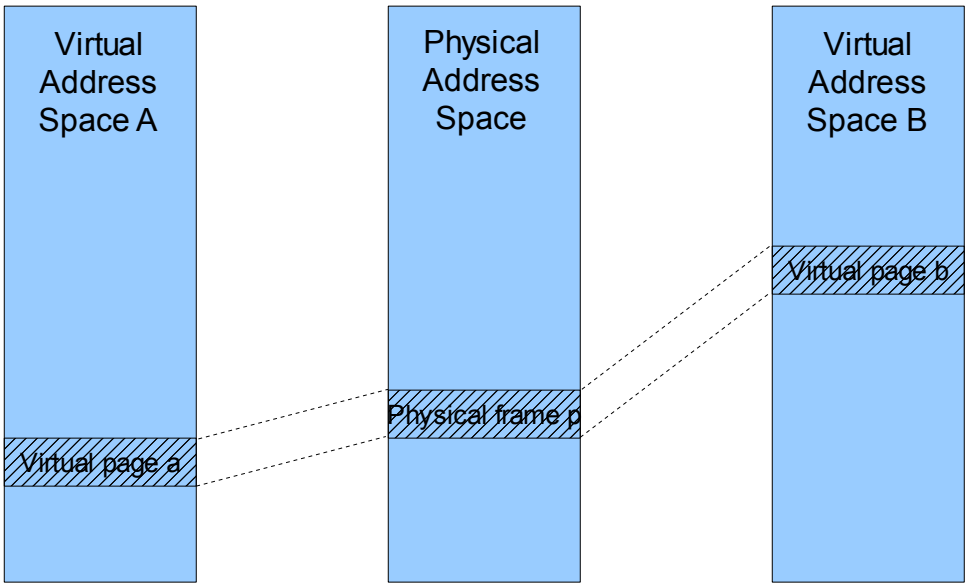
The Page Frame Manager carves the available physical memory into 4MB regions. A dedicated "buddy allocator" manages each region, subdividing it into blocks of one or more contiguous page frames. Each buddy allocator can support blocks up to 256KB of physical memory (i.e., up to 64 physically-contiguous frames).

### 2.3.4.7 Shared Frames

The kernel uses shared frames (`shared_frame_c`) to map physical page frames[7] into two or more address spaces simultaneously. Each address space has its own view of the shared frame; and the same frame may typically be mapped at a different virtual address in each address space.

Shared frames are primarily useful for transferring large blocks of payload data between address spaces; instead of wasting cycles to copy the data from one address space to another, the kernel "delivers" the payload by creating a view of it within the recipient's address space. Shared frames are typically marked as copy-on-write, in the event that either the sender or the recipient modifies the payload data.

A shared frame remains "live" until all address spaces have freed/unmapped it, overwrite it or otherwise destroy their references to it. Once all of these references are removed, the frame is then freed and eligible for reuse.




*Drawing 4: A single physical frame shared between two address spaces*

#### *2.3.4.7.1 Copy-on-write*

When a single frame is shared among two or more address spaces, the kernel marks the frame as read-only in each address space. This allows the kernel to intercept writes to this frame, which in turn allows the kernel to support copy-on-write for these shared frames.

[7] Only 4KB pages may be shared between address spaces. The kernel does not support shared 4MB super-pages.

### 2.3.5 Thread Manager

| | |
|---|---|
| Subsystem | Thread Manager |
| Purpose | Provides thread creation, tracking + deletion services. |
| Handles interrupts | None |
| Handles system calls | Create Thread |
| | Delete Thread |
| Kernel Interfaces | Thread creation, tracking and deletion. |
| Spinlocks | One to protect the table of current threads. |

#### 2.3.5.1 Creating a New Thread

The Thread Manager's main duty is the creation of new threads. It exposes a method, `thread_manager::create_thread()` that creates, initializes and returns a handle to a new thread (a new `thread_c` object).

Each thread executes within the confines of a single address space[8]. This address space container must be specified at thread-creation time; the `thread_manager_c::create_thread()` method then inserts this new thread into the specified address space.

Each new thread begins execution in `hal::switch_thread()` as if it were waking from suspension (see section 2.3.1.5). Unlike existing threads that simply resume where they left off, a new thread has no previous context and immediately jumps to `hal::run_thread()` instead. The `hal::run_thread()` logic then acts as a wrapper around the actual thread-specific entry point, launching the thread + cleaning up as necessary.

Both user-mode and kernel-mode threads are created via this same logic. For dedicated kernel threads, the thread-specific entry point is some internal kernel routine. For user-mode threads, the entry point is `::user_thread_entry()`.

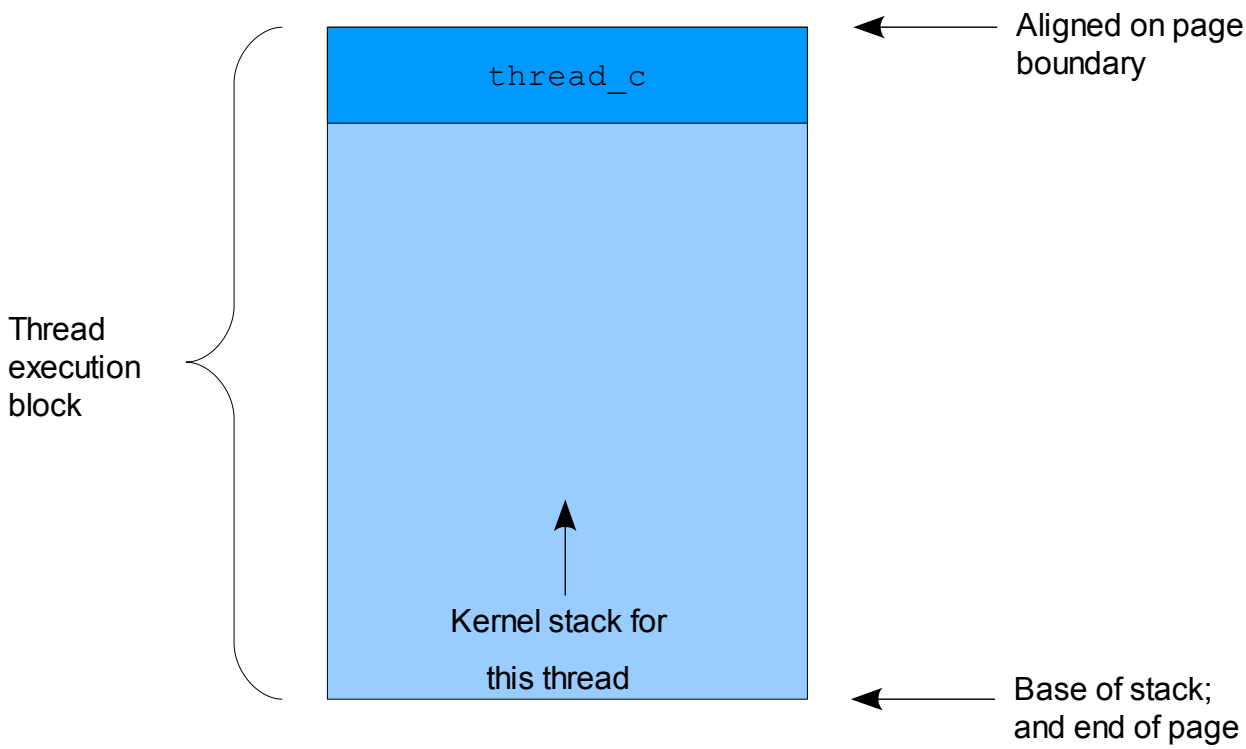
@fork() support?

#### 2.3.5.2 Thread Execution Block

Although most kernel code sees only the `thread_c` object that describes the thread state, the Thread Manager actually allocates a larger structure (the "Thread Execution Block") with this `thread_c` context embedded within it. This block is a single page of memory containing both the `thread_c` context and its dedicated kernel stack, as illustrated below.

Only the Thread Manager, the thread itself and the HAL are aware of the underlying block. All other kernel logic sees and interacts only with the `thread_c` object.

[8] Kernel threads all share a single, common address space. Kernel threads do not specifically require a dedicated address space since they can effectively execute in any address space, but this simplifies the logic for switching contexts (the reference-counting, in particular).




*Drawing 5: Thread execution block and thread_c context*

#### 2.3.5.3 Deleting an Existing Thread

Thread deletion occurs in one of two ways:

- The thread exits gracefully, either by returning from its entry point or by specifically invoking `thread_exit()`. In either event, the thread sends a THREAD_EXIT message to the Cleanup Thread (see section 2.4) , yields the CPU and never executes again (see section 2.3.1.5).
- Some other thread explicitly kills or destroys the victim thread. In this case, the state of the victim thread is unknown – it may be sleeping or executing normally, but it may also have crashed, runaway, etc.

Regardless of how the deletion is triggered, the Thread Manager (in the context of the Cleanup thread) marks the victim thread as deleted and removes it from its internal tables. At this point, the victim thread will not win any future scheduling lotteries; and subsequent queries/lookups for this thread will fail. The actual `thread_c` context and the underlying execution block, however, persist until all remaining references are released[9].

Thread deletion typically occurs within the context of the Cleanup thread, which actually invokes `thread_manager::delete_thread()`, but may occur in the context of any thread holding a reference to the victim.

[9] The only remaining references should be those that existed prior to the deletion, since the Thread Manager will not return new references to the victim after its deletion.

### 2.3.6 Device Proxy

| Subsystem | Device Proxy |
|---|---|
| Purpose | Provides intermediate layer for supporting device drivers in user-space. In effect, the Device Proxy exports raw device resources to user-mode drivers. |
| Handles interrupts | All PIC vectors except IRQ0. |
| Handles system calls | Map Device |
| | Unmap Device |
| Kernel Interfaces | None |
| Spinlocks | One, to protect the lists of interrupt handlers. @@this is broken on SMP. |

#### 2.3.6.1 Hardware Resources

The Device Proxy is primarily responsible for making raw hardware/device resources available to drivers in user space. It specifically exports three types of resources:

- I/O ports. A driver may request access to one or more I/O ports. I/O ports are allocated per-address space (meaning that all threads in the same address space have access to the same set of ports).
- Physical memory. A driver may map or request access to a specific range of physical memory. Like I/O ports, the Device Proxy manages physical memory per-address space.
- Interrupts. A driver may register interest in one or more interrupt vectors. Unlike I/O ports and physical memory blocks, interrupt registration is per-thread: only the thread that registered for interrupts is notified of the interrupt.

#### 2.3.6.2 Dispatching Device Interrupts

When an external device interrupts the CPU, the Device Manager sends a blocking message to each thread that has indicated an interest in the corresponding interrupt line. This simultaneously suspends the current (interrupted) thread and launches the handler thread in its place. By definition, each of these handlers executes in interrupt context (see 2.3.1.3.2).

If a handler discovers that its device is interrupting, it is expected to acknowledge or otherwise disable the interrupt and send a response message back to the original (interrupted) thread. The handler may send additional messages as well, for example, to allow another thread to finish the device cleanup outside of interrupt context, etc. The libdx library provides a simple framework for implementing this behavior (see 3.2).

## *2.4 Special Kernel Threads*

There are a handful of dedicated threads that execute solely within the kernel to provide special services.

| Name | Id | Purpose |
|---|---|---|
| Boot thread | 0 | The initial thread of execution, inherited from the boot loader when it jumps to the kernel. All kernel initialization occurs within the context of the boot thread. The boot thread does not respond to incoming messages. |
| Null thread | 1 | The null thread acts as message-sink. It consumes any messages sent to it, but never replies. The null thread provides a convenient source for messages sent from the kernel to user threads. |
| Cleanup thread | 2 | The cleanup thread assists the Thread Manager by providing a dedicated context for thread deletion/cleanup. Threads may exit gracefully by sending a (blocking) THREAD_EXIT message to the cleanup thread; or threads may send DELETE_THREAD messages |

| | | |
|---|---|---|
| | | to the cleanup thread to forcibly delete other threads. In either event, the cleanup thread provides the context for this deletion. |
| Idle thread | Any | The idle thread provides a dedicated context for burning CPU cycles when no other threads are ready to execute. The idle thread just loops forever + so it is always ready to execute. The idle thread does not accept incoming messages and therefore can never win a scheduling lottery; it executes only when specifically dispatched by the I/O Manager. See also section 2.3.3.6 |

# 3 Kernel-User Boundary

## 3.1 System Calls

All system calls are implemented in the same manner:

1. The user application populates a `syscall_data_s` structure with the necessary system call parameters.
2. The user application traps to the kernel.
3. The kernel returns the resulting status, and possibly additional arguments, via the original `syscall_data_s` structure.

## 3.2 libdx

The `libdx` library is the main dx system library. It provides C-language wrappers around all of the dx system calls. Device drivers may need to invoke these interfaces directly; but most applications will typically only need the system C library (which in turn invokes the underlying `libdx` routines where necessary).

In addition to the system calls, the `libdx` library also includes routines for implementing device drivers in user space. In particular, it provides a framework for handling interrupts in user mode drivers (the `register_interrupt_handler()` and `interrupt_handler_loop()` routines).

## 3.3 Start-up Object

The `user_start.o` object file provides the default user-mode entry point for most user executables. It prepares the various environmental settings and then invoked the application's `main()` entry point. If the user application returns from its `main()` entry point, the `user_start.o` object automatically calls `exit()`.

# 4 Application-level Servers

Keyboard driver
Console driver
Pager
FS
Networking
Alarm Clock

# 5 User Processes + Servers

# 6 Boot Process

## 6.1 Bootstrap sequence

At a high-level, the boot process looks like this:

- The BIOS loads the boot loader (GRUB) from disk and jumps to it.
- GRUB loads the kernel image from disk; reads the ELF header on the kernel image to determine the kernel's expected load address (i.e., physical address 0x00200000, see section 2.3.4.3); and places the kernel at this location.
- GRUB loads the ramdisk from disk; and places it somewhere beyond the end of the kernel image. Currently the kernel code image + ramdisk image are assumed to fit within the 6MB of RAM starting at 0x00200000.
- GRUB jumps to the dx kernel. At this point, the dx kernel now has control of the system. This thread of execution becomes the kernel "boot thread".
- The kernel initializes the various subsystems and internal threads. First the HAL, then the remaining subsystems. At this point, the kernel is now ready: it can send messages, schedule threads, allocate and free pages, etc.
- The kernel then spawns a separate thread to handle the user-mode initialization.
- This new thread locates the ramdisk; and jumps to the first entry in the ramdisk directly. At this point, the thread is now executing the contents of the ramdisk at ring-3.
- The loader then unpacks the rest of the executables in the ramdisk and launches them as separate processes.
-

## 6.2 RAMDISK

The kernel requires a ramdisk in order to boot. The ramdisk is a .tar file containing user-mode initialization code, user-mode device drivers and other user-mode components required to boot the rest of the system.

The first entry in the ramdisk is special: it contains the first instructions executed at ring-3. This entry is the "user loader" executable, The initial user thread jumps directly into this code and expects it to unpack, load or otherwise initialize all of the other entries in the ramdisk.

All of the remaining entries are normal executables. The user loader can unpack and launch each of these executables via normal system calls.

The ramdisk layout essentially looks like this:

| TAR header 0 (512 bytes) |
| --- |
| Loader executable (variable size) |
| TAR header 1 (512 bytes) |
| Executable 1 (variable size) |
| TAR header 2 (512 bytes) |
| Executable 2 (variable size) |
| … etc., … |
| TAR header N (512 bytes) |
| Executable N (variable size) |
| TAR terminator (1024 bytes) |