The OS/kernel will contain many abstraction layers to provide a degree of separation between the many systems in the kernel. This ensures we can make changes to an individual system without having to also make changes to every system that interacts with the changed system. Another advantage of abstraction layers is improved organization, since the interface is clearly defined in this document. When complex systems are built, its hard to keep track of the various dependencies across modules within this system and makes development difficult. We will attempt to split the kernel into distinct ‘systems’ each with a clean and well-defined interface. Also, finding the most optimal way to place code into files, and how to organize those files in a clear structure will be explored in this document.

Kernel System List (Platform Agnostic)

* Page Allocation
* Kernel Memory Map
* Kernel Heap
* Interrupt Management
* Interlink
* System Info
* Time Keeping
* System Timers

X86-64 Kernel System List

* ACPI Tables
* X86 Topology

File Structure

private/axon/

arch\_XXX/

internal\_drivers/

acpi\_timer.h

hpet\_timer.h

lapic\_timer.h

pit\_timer.h

tsc\_timer.h

xapic\_interrupts.h

x2apic\_interrupts.h

acpi\_tables.h

util.h

boot\_parameters.h

memory/

memory\_private.h

system/

interrupts\_private.h

interlink\_private.h

time\_private.h

timers\_private.h

sysinfo\_private.h

scheduler/

global\_scheduler.h

local\_scheduler.h

kernel/

boot\_parameters.h

public/axon/

library/

rbtree.h

spinlock.h

vector.h

atomic.h

system/

interlink.h

interrupts.h

sysinfo.h

time.h

timers.h

memory/

page\_map.h

page\_alloc.h

memmap.h

gfx/

basic\_terminal.h

font\_psf1.h

kernel/

kernel.h

panic.h

error.h

source/

arch\_XXX/

internal\_drivers/

acpi\_timer.c

hpet\_timer.c

lapic\_timer.c

pit\_timer.c

tsc\_timer.c

xapic\_interrupts.c

x2apic\_interrupts.c

system/

exceptions.c

interrupts.c

time.c

timers.c

topology.c

acpi\_tables.c

memory/

page\_map.c

util.c

entry.c

library/

rbtree.c

spinlock.c

vector.c

memory/

heap.c

page\_alloc.c

scheduler/

global\_scheduler.c

local\_smp\_scheduler.c

local\_amp\_scheduler.c

system/

interlink.c

interrupts.c

sysinfo.c

time.c

timers.c

kernel/

error.c

panic.c

Kernel Entry Point

* This is one of the more challenging systems to organize, because it is TOTALLY dependent on the architecture of the computer. Each architecture needs code to get the computer up into a viable 64-bit state, which would ideally occur in the bootloader, or in kernel assembly code.
* Then, we need to build two lists of parameters for the main kernel initialization.

1. axk\_boot\_parameters\_t – A list of parameters that is independent of the architecture, containing things like a physical memory map, etc.
2. axk\_(ARCH)\_boot\_parameters\_t – A list of parameters specific to the architecture, for example, on x86 we might have ACPI information in here

* The ‘main’ initialization function is a C function, located in ‘source/arch\_XXX/entry.c’
* Optionally, the entry point function can be in assembly code, which will end with a call to the initialization function in ‘source/arch\_xxx/entry.c’
* We need an architecture aware initialization function for a host of reasons, but later in the boot phase we will switch to architecturally generic code
* Because this portion of the codebase can vary so much depending on the architecture, there isn’t a lot of constraints on the interface. Just a few naming constraints, and a list of initialization functions that must be called in a specific order by the main initialization function.

TBD (Arch-Independent initialization code?)

…

1st: Initialize Systems

Bootloader

Optional:

Pre-Init Assembly Code

Main Initialization Function

arch\_XXX/entry.c

Memory System

SysInfo

Kernel Heap

Low Level Memory Systems

* These are the lowest level systems dedicated to managing system memory, and how its mapped virtually, there are higher level systems built atop of these. The low-level memory systems are split into two parts:
  + Page Allocator
  + Page Mapper
* The ‘Page Allocator’ keeps track of all physical memory pages in the computer. Each page is assigned a ‘status’, which indicates if its available, in-use or reserved. Also, we have flags to indicate if the page is an image, part of a heap, page table mappings, ram-disk image or a system information table. We also track the associated process ID for each physical page
* The ‘Page Mapper’ can manipulate address spaces, by mapping in physical pages, removing mappings, looking up mappings, and translating addresses. These operations are performed on structures called ‘axk\_memory\_map\_t’. The implementation of this system is dependent on architecture, so the code file is located at ‘source/arch\_XXX/memory/page\_map.c’

Page Allocator Functions:

bool axk\_page\_acquire( uint64\_t count, uint64\_t\* out\_list, uint32\_t process, uint32\_t flags )

bool axk\_page\_lock( uint64\_t count, uint64\_t\* in\_list, uint32\_t process, uint32\_t flags )

bool axk\_page\_release( uint64\_t count, uint64\_t\* in\_list, bool b\_allow\_kernel )

bool axk\_page\_status( uint64\_t page\_id, uint32\_t\* out\_process, uint32\_t\* out\_flags )

uint64\_t axk\_page\_freeproc( uint32\_t process )

Page Mapping Functions: