Chasing the Penguin: State and Evolution of the Kernel

Wolfgang Mauerer
MPRG IOIP & linux-kernel.net
10. October 2008
Dynamics of Kernel Development

Kernel Documentation

Understanding the Kernel
   Documenting new Features
   Analysis Tools

Social Aspects
Outline

Dynamics of Kernel Development

Kernel Documentation

Understanding the Kernel
  Documenting new Features
  Analysis Tools

Social Aspects
Dynamics of Kernel Development

![Graph showing the dynamics of kernel development, with # lines affected on the y-axis and Kernel 2.6.x releases on the x-axis. Peaks indicate significant changes.]
Dynamics of Kernel Development

![Graph showing the dynamics of kernel development with lines indicating cumulative insertions and deletions over time. The x-axis represents Kernel 2.6.x versions from 12 to 26, and the y-axis represents the number of lines affected, ranging from 0 to 14 M. The graph includes four lines: red for cumulative, green for insertions, and blue for deletions.](logo-main)
Hare and Tortoise

- Code sufficient?
- How to document?
- Which parts?

Kernel 2.6.x
- Core Kernel
- Linear fit
- Complete Kernel
- Device Drivers

Kernel 2.5.x
Hare and Tortoise

- Code sufficient?
- How to document?
- Which parts?

Kernel 2.6.x
Core Kernel
Linear fit
Complete Kernel
Device Drivers

Kernel 2.5.x

www.linux-kernel.net
Outline

Dynamics of Kernel Development

Kernel Documentation

Understanding the Kernel
  Documenting new Features
  Analysis Tools

Social Aspects
## What’s available?

<table>
<thead>
<tr>
<th>In-Tree</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Comments and Kerneldoc</td>
<td>▶ LKML and others</td>
</tr>
<tr>
<td>▶ Documentation/</td>
<td>▶ Websites: lwn.net, kernelnewbies.org, ...</td>
</tr>
<tr>
<td>▶ Git commit messages</td>
<td>▶ Books and Articles</td>
</tr>
</tbody>
</table>
What’s available?

In-Tree

- Comments and Kerneldoc
- Documentation/
- Git commit messages

External

- LKML and others
- Websites: lwn.net, kernelnewbies.org, ...
- Books and Articles

/**
 * clocksource_khz2mult - calculates mult from khz and shift
 * @khz: Clocksource frequency in KHz
 * @shift_constant: Clocksource shift factor
 *
 * Helper functions that converts a khz counter frequency to a timsource
 * multiplier, given the clocksource shift value
 */
static inline u32 clocksource_khz2mult(u32 khz, u32 shift_constant)
...
### What’s available?

**In-Tree**
- Comments and Kerneldoc
- Documentation/
- Git commit messages

**External**
- LKML and others
- Websites: lwn.net, kernelnewbies.org, ...
- Books and Articles

---

**sx.txt** – specialix SX/SI multiport serial driver readme.
Copyright (C) 1997 Roger Wolff (R.E.Wolff@BitWizard.nl)

...  
Introduction  
==========

This file contains some random information, that I like to have online instead of in a manual that can get lost. Ever misplace your Linux kernel sources? And the manual of one of the boards in your computer? ...

---

www.linux-kernel.net W. Mauerer, Chasing the Penguin
What’s available?

In-Tree

- Comments and Kerneldoc
- Documentation/
- Git commit messages

External

- LKML and others
- Websites: lwn.net, kernelnewbies.org, ...
- Books and Articles

commit 2087a1ad822cd3a68b73338457047fcc54da726b
Author: Gregory Haskins <ghaskins@novell.com>
Date: Fri Jun 27 14:30:00 2008 -0600

sched: add avg-overlap support to RT tasks

We have the notion of tracking process-coupling (a.k.a. buddy-wake) via the p->se.last_wake / p->se.avg_overlap facilities, but it is only used for cfs to cfs interactions. There is no reason why an rt to cfs interaction cannot share in establishing a relationship in a similar manner.

Because PREEMPT_RT runs many kernel threads as FIFO priority, we often times have heavy interaction between RT threads waking CFS applications. This patch offers a substantial boost (50-60%+) in performance under those circumstances.
## What’s available?

### In-Tree
- Comments and Kerdeldoc
- Documentation/
- Git commit messages

### External
- LKML and others
- Websites: lwn.net, kernelnewbies.org, ...
- Books and Articles
What's available?

In-Tree

- Comments and Kerneldoc
- Documentation/
- Git commit messages

External

- LKML and others
- Websites: lwn.net, kernelnewbies.org, ...
- Books and Articles
### What’s available?

**In-Tree**
- Comments and Kernelsdoc
- Documentation/
- Git commit messages

**External**
- LKML and others
- Websites: lwn.net, kernelnewbies.org, ...
- Books and Articles
What’s available?

<table>
<thead>
<tr>
<th>In-Tree</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comments and Kerneldoc</td>
<td>LKML and others</td>
</tr>
<tr>
<td>Documentation/</td>
<td>Websites: lwn.net, kernelnewbies.org, ...</td>
</tr>
<tr>
<td>Git commit messages</td>
<td>Books and Articles</td>
</tr>
</tbody>
</table>

Problems

- Available? Location?
- Uptodate? Complete?
## Summary

<table>
<thead>
<tr>
<th>What’s good</th>
<th>What’s bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Huge amount of documentation available</td>
<td>▶ Focus on people already intimate with the code</td>
</tr>
<tr>
<td>▶ Implicit documentation in git</td>
<td>▶ Implicit documentation in git</td>
</tr>
<tr>
<td>▶ Documentation infrastructure available</td>
<td>▶ No consistent style</td>
</tr>
<tr>
<td></td>
<td>▶ Very fragmented and scattered</td>
</tr>
</tbody>
</table>
Outline

Dynamics of Kernel Development

Kernel Documentation

Understanding the Kernel
  Documenting new Features
  Analysis Tools

Social Aspects
Documenting new features

Completely Fair Scheduler
- Turbulent emergence
- Completely replaces old scheduler
- Considerable in-tree development after merge

High Resolution Timers
- Long external development
- New foundation for existing framework
- Merged at very mature state

Opposite strategies...
... also with respect to documentation!
Documenting new features

<table>
<thead>
<tr>
<th>Completely Fair Scheduler</th>
<th>High Resolution Timers</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Turbulent emergence</td>
<td>▶ Long external development</td>
</tr>
<tr>
<td>▶ Completely <em>replaces</em> old scheduler</td>
<td>▶ New foundation for <em>existing</em> framework</td>
</tr>
<tr>
<td>▶ Considerable in-tree development after merge</td>
<td>▶ Merged at very mature state</td>
</tr>
</tbody>
</table>

Opposite strategies…

…also with respect to documentation!
Documenting new features

**Completely Fair Scheduler**
- Turbulent emergence
- Completely *replaces* old scheduler
- Considerable in-tree development after merge

**High Resolution Timers**
- Long external development
- New foundation for *existing* framework
- Merged at very mature state

Opposite strategies...

...also with respect to documentation!
## Documenting new features

<table>
<thead>
<tr>
<th>Completely Fair Scheduler</th>
<th>High Resolution Timers</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Turbulent emergence</td>
<td>- Long external development</td>
</tr>
<tr>
<td>- Completely <em>replaces</em> old scheduler</td>
<td>- New foundation for <em>existing</em> framework</td>
</tr>
<tr>
<td>- Considerable in-tree development after merge</td>
<td>- Merged at very mature state</td>
</tr>
</tbody>
</table>

### Opposite strategies...

...also with respect to documentation!
High Resolution Timers

- Devices with limited power (i.e., laptops, embedded systems, etc.) need to use as little energy as possible when there is nothing to do. If a periodic clock is running, there is, however, nearly always something to do — the tick must be provided. But if no users for the tick are present, it would basically not need to run. Nevertheless, the system needs to be brought from a low-power state into a state with higher power consumption just to implement the periodic tick.

- Multimedia-oriented applications need very precise timekeeping, for instance, to avoid frame skips in videos, or jumps during audio playback. This necessitated increasing the available resolution.

Finding a good solution agreeable to all developers (and users!) who come into contact with time management — and there is quite a large number of them — took many years and a good many proposed patches. The current state is rather unusual because two rather distinct types of timers are supported by the kernel:

- **Classical timers** have been available since the initial versions of the kernel. Their implementation is located in `kernel/timer.c`. A resolution of typically 4 milliseconds is provided, but the value depends on the frequency with which the machine's timer interrupt is operated. These classical timers are called **low-resolution** or **timer wheel** timers.

- For many applications, especially media-oriented ones, a timer resolution of several milliseconds is not good enough. Indeed, recent hardware provides means of much more precise timing, which can achieve resolutions in the nanosecond range formally. During the development of kernel 2.6, an additional timer subsystem was added allowing the use of such timer sources. The timers provided by the new subsystem are conventionally referred to as **high-resolution timers**.

Some code for high-resolution timers is always compiled into the kernel, but the implementation will only perform better than low-resolution timers if the configuration option `HIGH_RES_TIMERS` is set. The framework introduced by high-resolution timers is reused by low-resolution timers (in fact, low-resolution timers are implemented on top of the high-resolution mechanism).

Classical timers are bound by a fixed raster, while high-resolution clock events can essentially happen at arbitrary times; see Figure 15-1. Unless the dynamic ticks feature is active, it can also happen that ticks occur when no event expires. High-resolution events, in contrast, only occur when some event is due.

![Figure 15-1: Comparison between low- and high-resolution timers.](image)

Why did the developers not choose the seemingly obvious path and improve the already existing timer subsystem, but instead added a completely new one? Indeed, some people tried to pursue this strategy, but the mature and robust structure of the old timer subsystem did not make it particularly easy to improve while still being efficient — and without creating new problems. Some more thoughts on this problem can be found in `Documentation/hrtimers.txt`.

[Diagram of time events]
Chapter 15: Time management

Independent of the resolution, the kernel nomenclature distinguishes two types of timers:

- **Time-outs** — Represent events that are bound to happen after some time, but can and usually will be canceled before. For example, consider that the network subsystem waits for an incoming packet that is bound to arrive within a certain period of time. To handle this situation, a timer is set that will expire after the time is over. Since packets usually arrive on time, chances are that the timer will be removed before it will actually go off. Besides resolution is not very critical for these types of timers. When the kernel allows an acknowledgment to a packet to be sent within 10 seconds, it does not really matter if the time-out occurs after 10 or 10.001 seconds.

- **Timers** — Are used to implement temporal sequences. For instance, a sound card driver could want to issue some data to a sound card in small, periodic time intervals. Timers of this sort will usually expire and require much better resolution than time-outs.

An overview of the building blocks employed to implement the timing subsystem is given in Figure 15-2. Owing to the nature of an overview, it is not too precise, but gives a quick glance at what is involved in timekeeping, and how the components interact with each other. Many details are left to the following discussion.

![Diagram of the components of the timing subsystem](image-url)

The raw hardware sits at the very bottom. Every typical system has several devices, usually implemented by clock chips, that provide timing functionality and can serve as clocks. Which hardware is available depends on the particular architecture. IA-32 and AMD64 systems, for instance, have a programmable interrupt timer (PIT, implemented by the 8253 chip) as a classical clock source that has only a very modest resolution and stability. CPU-local APICs (advanced programmable interrupt controllers), which were already mentioned in the context of IRQ handling, provide much better resolution and stability. They are suitable as high-resolution time sources, whereas the PIT is only good enough for low-resolution timers.

Hardware naturally needs to be programmed by architecture-specific code, but the clock source abstraction provides a generic interface to all hardware clock chips. Essentially, read access to the current value of the running counter provided by a clock chip is granted.
Components of the high-resolution timer framework that are not universally applicable, but do really provide actual high-resolution capabilities are bracketed by the pre-processor symbol \texttt{CONFIG\_HIGH\_RES\_TIMERS}, and are only compiled in if high-resolution support is selected at compile time. The generic part of the framework is always added to the kernel. This means that even kernels that only support low resolution contain parts of the high-resolution framework, which can sometimes lead to confusion.

### 15.4.1 Data Structures

High-resolution timers can be based on two different types of clocks (which are referred to as clock bases). The monotonic clock starts at 0 when the system is booted (\texttt{CLOCK\_MONOTONIC}). The other clock (\texttt{CLOCK\_REALTIME}) represents the real time of the system. The latter clock may exhibit skips if, for instance, the system time is changed, but the monotonic clock runs, well, monotonously all the time.

For each CPU in the system, a data structure with both clock bases is available. Each clock base is equipped with a red-black tree that sorts all pending high-resolution timers. Figure 15-12 summarizes the situation graphically. Two clock bases (monotonic and real time) are available per CPU. All timers are sorted by expiration time on a red-black tree, and expired timers whose callback handlers still need to be executed are moved from the red-black tree to a linked list.

![Diagram of high-resolution timer data structures](image_url)

A clock base is given by the following data structure:

```c
struct hrtimer_clock_base {
    struct hrtimer_cpu_base *cpu_base;
    clockid_t index;
    struct rb_root active;
    struct rb_node *first;
    ktime_t resolution;
    ktime_t (*get_time)(void);
};
```
Chapter 15: Time management

The expiration time of the timer that is due next is stored in `expires_next`. Setting this to `KTIME_MAX` initially is another way of saying that no next timer is available. The main work is to iterate over all clock bases (monotonic and real-time).

```c
for (i = 0; i < HRTIMER_MAX_CLOCK_BASES; i++) {
    ktime_t basenow;
    struct rb_node *node;
    basenow = ktime_add(now, base->offset);
    basenow = ktime_add(basenow, base->offset);
    while ((node = base->first)) {
        struct hrtimer *timer;
        timer = rb_entry(node, struct hrtimer, node);
        if (basenow.tv64 < timer->expires.tv64) {
            ktime_t expires;
            expires = ktime_sub(timer->expires, base->offset);
            if (expires.tv64 < expires_next.tv64) {
                expires_next = expires;
                break;
            }
            if (timer->cb_mode == HRTIMER_CB_SOFTIRQ) {
                --remove_hrtimer(timer, base, HRTIMER_STATE_PENDING, 0);
                list_add_tail(&timer->cb_entry, &base->cpu_base->cb_pending);
                raise = 1;
                continue;
            }
        }
        --run_hrtimer(timer);
    }
    spin_unlock(&cpu_base->lock);
    base++;
}
```

Figure 15-13: Overview of expiration of high-resolution timers with high-resolution clocks.

- **High-resolution clock interrupt**
  - Select expired timers
  - Reprogram hardware for next event
  - Raise HRTIMER_SOFTIRQ
  - Move to expired list
  - Run hrtimer_softirq
  - Execute directly
  - Process pending timers
# High Resolution Timers

## Available
- Orthogonal patch structure
- Component submission
- Design Documentation

## Challenges
- Introduce conceptual parts
- Disentangle alternatives
- Prioritise important against unimportant code
Every time the scheduler is called, it picks the task with the highest waiting time and gives the CPU to it. If this happens often enough, no large unfairness will accumulate for tasks, and the unfairness will be evenly distributed among all tasks in the system.

Figure 2-12 illustrates how the scheduler keeps track of which process has been waiting for how long. Since runnable processes are queued, the structure is known as the run queue. All runnable tasks are time-ordered in a red-black tree, essentially with respect to their waiting time. The task that has been waiting for the CPU for the largest amount of time is the leftmost entry and will be considered next by the scheduler. Tasks that have been waiting less long are sorted on the tree from left to right.

If you are not familiar with red-black trees, suffice it to know here that this data structure allows for efficient management of the entries it contains, and that the time required for lookup, insertion, and deletion operations will only moderately rise with the number of processes present in the tree. Red-black trees are available as a standard data structure of the kernel, and Appendix C provides more information about them. Besides, a discussion of such trees can be found in every textbook on data structures.

Besides the red-black tree, a run queue is also equipped with a virtual clock. Time passes slower on this clock than in real time, and the exact speed depends on the number of processes that are currently waiting to be picked by the scheduler. Suppose that four processes are on the queue: Then the virtual clock will run at one-quarter of the speed of a real clock. This is the basis to determine how much CPU time a waiting process would have gotten if computation power could be shared in a completely fair manner. Sitting on the run queue for 20 seconds in real time amounts to 5 seconds in virtual time. Four tasks executing for 5 seconds each would keep the CPU occupied for 20 seconds in real time.

To be precise: Time complexity is $O(\log n)$, where $n$ is the number of elements in the tree. This is worse than for the old scheduler, which was famous for being an $O(1)$ scheduler, that is, its run time was independent of the number of processes it had to deal with. However, the slow-down caused by the linear-logarithmic dependency of the new scheduler is negligible unless a huge number of processes is simultaneously runnable. In practice, such a situation does not occur.

Notice that the kernel really used the concept of a virtual clock for the scheduling mechanism in kernel 2.6.23, but currently computes the virtual time a little differently. Since the method is easier to understand with virtual clocks, I will stick to this now and discuss how the virtual clock is emulated when I discuss the scheduler implementation.
Completely Fair Scheduler

Virtual clock replacement

```c
static inline s64 entity_key(struct cfs_rq *cfs_rq, 
    struct sched_entity *se) {
    return se->vruntime - cfs_rq->min_vruntime;
}
```
<table>
<thead>
<tr>
<th>Conclusions</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Code history can ease documentation</td>
</tr>
<tr>
<td>▶ Identify stable components</td>
</tr>
<tr>
<td>▶ Reduce complexity</td>
</tr>
</tbody>
</table>
Analysis Tools
Analysis Tools
```c
static int
do_next(struct task_struct *prev, struct task_struct *next)
{
    if (unlikely(signal_pending_state(prev, prev)))
        prev->state = TASK_RUNNING;
    else
        deactivate_task(prev, 1);

    switch_count = &next->switch_count;
    return 0;
}
```

(K)GDB and DDD

www.linux-kernel.net  W. Mauerer, Chasing the Penguin
Andrew Morton on KGDB

I used kgdb continuously for 4-5 years until it broke. I don’t think I ever used it much for “debugging” as such. I used it more for general observation of what’s going on in the kernel. And for confirmation of what’s going on (ie: testing that the actual state matches the expected state).
19

Git and GUIs

www.linux-kernel.net

W. Mauerer, Chasing the Penguin
How it all fits together

Data Structures

LXR

GDB KGDB

Algorithms

GIT

UML

W. Mauerer, Chasing the Penguin
How it all fits together
Outline

Dynamics of Kernel Development

Kernel Documentation

Understanding the Kernel
  Documenting new Features
  Analysis Tools

Social Aspects
From: Con Kolivas

Ooh you have a vm patch that helps swap on the desktop! I can help you here with my experience from swap prefetch.

1. Get it reviewed and have noone show any evidence it harms
2. Find hundreds of users who can testify it helps
3. Find a way of quantifying it.
4. ...
5. Merge into mainline.

There, that should get you as far as 4. I haven’t figured out what 4 is yet. I believe it may be goto 1;
From: Con Kolivas

Ooh you have a vm patch that helps swap on the desktop! I can help you here with my experience from swap prefetch.

1. Get it reviewed and have noone show any evidence it harms
2. Find hundreds of users who can testify it helps
3. Find a way of quantifying it.
4. ...
5. Merge into mainline.

There, that should get you as far as 4. I haven’t figured out what 4 is yet. I believe it may be goto 1;
Linus on smelly pets

Ok, so now that I’ve insulted you and your pets (they’re ugly!), show me wrong, and then call me a d*ckhead. (“Linus – you’re a d*ckhead, and you didn’t understand the problem, so you’re a stupid d*ckhead. And my pet may be ugly, but yours smells bad!”).
#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
#define ARRAY_SIZE(arr) (sizeof(arr) / sizeof((arr)[0])
  + sizeof(typeof(int[1 - 2*!!__builtin_types_compatible_p(typeof(arr),
  typeof(&arr[0])))]) * 0)

Reply from Linus Torvalds

Rusty, that’s a work of art.
However, I would suggest that you never show it to anybody ever again. I’m sure that in fifty years, it will be worth much more. So please keep it tightly under wraps, to keep people from gouging their eyes out^W^W^W^W^W^W^W make a killing in the art market.
OK, many people complained that it needed a comment. Good point!

Add comment to ARRAY_SIZE macro.

diff -r 933e410f204f include/linux/kernel.h
--- a/include/linux/kernel.h Sat Mar 10 09:55:31 2007 +1100
+++ b/include/linux/kernel.h Sat Mar 10 09:55:53 2007 +1100

 extern const char linux_proc_banner[];

#define ALIGN(x,a) __ALIGN_MASK(x,(typeof(x))(a)-1)
#define __ALIGN_MASK(x,mask) (((x)+(mask))&~(mask))

+/* GCC is awesome. */
+define ARRAY_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]) \ 
+ sizeof(typeof(int[1 - 2*!!__builtin_types_compatible_p(typeof(arr), \ 
+ typeof(&arr[0]))]))*0)
Thanks for your attention!
Everybody needs memory

- Core OS service
- Stable interface (introduced ≈ v0.98)
- Documentation situation representative
malloc(9)  FreeBSD Kernel Developer's Manual  malloc(9)

NAME
    malloc, MALLOC, free, FREE, realloc, reallocf, MALLOC_DEFINE,
    MALLOC_DECLARE -- kernel memory management routines

SYNOPSIS
    #include <sys/types.h>
    #include <sys/malloc.h>

    void *
    malloc(unsigned long size, struct malloc_type *type, int flags);
    MALLOC(size, cast, unsigned long size, struct malloc_type *type,
            int flags);

    void *
    free(void *addr, struct malloc_type *type);
    FREE(void *addr, struct malloc_type *type);

    void *
    realloc(void *addr, unsigned long size, struct malloc_type *type,
            int flags);
    reallocf(void *addr, unsigned long size, struct malloc_type *type,
             int flags);
    MALLOC_DECLARE(type);
    #include <sys/param.h>
    #include <sys/malloc.h>
    #include <sys/kernel.h>

    MALLOC_DEFINE(type, shmmibuf, longdoc);

DESCRIPTION
    The malloc() function allocates uninitialized memory in kernel address
    space for an object whose size is specified by size.

    The free() function releases memory at address addr that was previously
    allocated by malloc() for re-use.  The memory is not zeroed.  If addr is
    NULL, then free() does nothing.

    The realloc() function changes the size of the previously allocated mem-
    ory referenced by addr to size bytes.  The contents of the memory are
    unchanged up to the lesser of the new and old sizes.  Note that the
    returned value may differ from addr.  If the requested memory cannot be
    allocated, NULL is returned and the memory referenced by addr is valid
    and unchanged.  If addr is NULL, the realloc() function behaves identi-
    cally to malloc() for the specified size.

    The reallocf() function is identical to realloc() except that it will
    free the passed pointer when the requested memory cannot be allocated.

    The MALLOC() macro variant is functionally equivalent to
    (space) = (cast)malloc((u_long)size, type, flags)
Object Creation and Destruction

Because exceptions are excluded from the kernel's restricted form of C++, you cannot implement "normal" C++ constructors and destructors without jeopardy. Constructors and destructors are tied to return no value (such as an error code). Normally, if you encounter a problem, they raise an exception. But because exceptions aren't supported in the kernel's C++ runtime, there is no way for you to know when an allocation or deallocation error occurred.

This situation prompted a design feature of the library's C++ runtime system that uses OSMetaClass macros to specify the structure of a class—that is, the meta-class data structures and functional interfaces—for the runtime typing system. The macros also define the primary constructor and a destructor for a class. These macro-created constructors are guaranteed to fail because they do not themselves perform any allocations. Instead, the runtime system calls the actual allocation of objects until the initialization (usually in the Init member function).

Because the Init function is tied to return a void, it makes impossible to retain an error upon any failure.

In this section:
- Using the OSMetaClass Constructor Macros
- Allocating Objects Dynamically
- Global Initializers

Using the OSMetaClass Constructor Macros

When you create a C++ class based on OSMetaObject, your code must call a matching pair of macros based upon the OSMetaClass class. The calls must be among the first statements in both the definition and implementation of the class. These macros are critical to your class because they enter meta-class information about the class into the library runtime typing facility and define the class constructor and destructor for your class.

For concrete (that is, non-abstract) classes, the first macro, OSDeclareDefaultStructors, declares the C++ constructors, by convention you insert this macro as the first element of the class declaration in the header file. For example:

```c
class com_MyCompany_Driver_MyDriver : public IDataService {
    OSMetaClass() com_MyCompany_Driver_MyDriver;
    /* ... */
};
```

Your class implementation must include the companion "define" macro, USDefineMetaClassAndStructors. This macro defines the constructor and destructor. Implements the OSMetaClass allocate member function (alloc) for the class, and supplies the meta-class information for the runtime typing system. USDefineMetaClassAndStructors takes as input the name of your class and the name of your base class. It then returns a macro that allows you to declare objects to be handled and instantiated while the compiler developer.apple.com/documentation/DeviceDrivers/Conceptual/WritingDeviceDriver/CPluPlusRuntime/chapter_2_section_3.html#//apple_ref/doc/uid/TP30000695-BAJCCBGJ
Windows Driver Kit: Kernel-Mode Driver Architecture

ExAllocatePoolWithTag

This function allocates pool memory of the specified type and returns a pointer to the allocated block.

Parameters

PoolTag

Specifies the type of pool memory to allocate. Each allocation code path should use a unique pool tag to help debuggers and verifiers identify the code path. You can modify the PoolTag value by using a bitwise OR with the POOL_GROWTH or POOL_ALLOCATION_FAILURE flag. This flag causes an exception to be raised if the request cannot be satisfied.

Similarly, you can modify PoolTag by using a bitwise OR with the POOL_GROWTH or POOL_ALLOCATION_FAILURE flag as a hint to the kernel to allocate the memory from pages that are likely to be paged out quickly. To reduce the amount of resident pool memory as much as possible, you should not reference these allocations frequently. The POOL_GROWTH or POOL_ALLOCATION_FAILURE flag is only advisory and is available for Microsoft Windows XP and later operating systems. For a description of the available pool memory types, see POOL_TYPE.

NumberOfBytes

Specifies the number of bytes to allocate.

Tag

Specifies the pool tag for the allocated memory. Specify the pool tag as a character literal of up to four characters delimited by single quotation marks (for example, "Tag"). The string is usually specified in reverse order (for example, "Tag"). The ASCII value of each character in the tag must be between 0 and 127. Every allocation code path should use a unique pool tag to ensure that debuggers and verifiers identify a distinct allocated block.

Return Value

ExAllocatePoolWithTag returns NULL if there is insufficient memory in the free pool to satisfy the request. Use of POOL_GROWTH or POOL_ALLOCATION_FAILURE is not recommended because it is costly. Any successful allocation that requests NumberOfBytes > PAGE_SIZE wastes all unused bytes on the last-allocated page. However, on Windows Vista and later, the unused bytes are no longer wasted.

Comments

This routine is used for the general pool allocation of memory.

If NumberOfBytes is PAGE_SIZE or greater, a page-aligned buffer is allocated. Memory allocations of PAGE_SIZE or less do not cross page boundaries. Memory allocations of less than PAGE_SIZE are not necessarily page-aligned but are aligned on an 8-byte boundary.
Memory Allocation

It is suggested that you use the version of malloc that is distributed with Perl. It keeps
pools of various sizes of unallocated memory in order to satisfy allocation requests
more quickly. However, on some platforms, it may cause spurious malloc or free
errors.

```
New(x, pointer, number, type);
Newc(x, pointer, number, type, cast);
Newz(x, pointer, number, type);
```

These three macros are used to initially allocate memory.

The first argument `x` was a "magic cookie" that was used to keep track of who called
the macro, to help when debugging memory problems. However, the current code
makes no use of this feature (most Perl developers now use run-time memory check-
ners), so this argument can be any number.

The second argument `pointer` should be the name of a variable that will point to the
newly allocated memory.

The third and fourth arguments `number` and `type` specify how many of the specified
type of data structure should be allocated. The argument `type` is passed to `sizeof`.

The final argument to `Newc`, `cast`, should be used if the `pointer` argument is dif-
ferent from the `type` argument.

Unlike the `New` and `Newc` macros, the `Newz` macro calls `memzero` to zero out all the
newly allocated memory.

```
Renew(pointer, number, type);
Renewc(pointer, number, type, cast);
Safefree(pointer);
```

These three macros are used to change a memory buffer size or to free a piece of mem-
ory no longer needed. The arguments to `Renew` and `Renewc` match those of `New` and
`Newz` with the exception of not needing the "magic cookie" argument.

```
Move(source, dest, number, type);
Copy(source, dest, number, type);
Zero(dest, number, type);
```

These three macros are used to move, copy, or zero out previously allocated memory.
The `source` and `dest` arguments point to the source and destination starting points.
Perl will move, copy, or zero out `number` instances of the size of the `type` data struc-
ture (using the `sizeof` function).

PerlIO

The most recent development releases of Perl have been experimenting with removing
Perl's dependency on the "normal" standard I/O suite and allowing other stdio imple-
mentations to be used. This involves creating a new abstraction layer that then calls
whichever implementation of stdio Perl was compiled with. All XSUBs should now
use the functions in the PerlIO abstraction layer and not make any assumptions about
what kind of stdio is being used.
Chapter 5. Memory Management in Linux

Table of Contents
The Slab Cache
User Space Memory Access
More Memory Management Functions

The Slab Cache

kalloc — allocate memory for an array. The memory is set to zero.
kmalloc_node — allocate memory from a specific node
ksmalloc — allocate memory. The memory is set to zero.
kzmalloc_node — allocate zeroed memory from a particular memory node.
kmem_cache_create — Create a cache.
kmem_cache_shrink — Shrink a cache.
kmem_cache_destroy — delete a cache
kmem_cache_alloc — Allocate an object
kmem_cache_free — Deallocate an object
kfree — free previously allocated memory
**Name**

`kmalloc_node` — allocate memory from a specific node

**Synopsis**

```c
void *kmalloc_node(size_t size,
                   gp_t flags,
                   int node);
```

**Arguments**

- `size`:
  - how many bytes of memory are required.
- `flags`:
  - the type of memory to allocate (see `kalloc`).
- `node`:
  - node to allocate from.

**Description**

`kmalloc` for non-local nodes, used to allocate from a specific node if available. Equivalent to `kalloc` in the non-NUMA single-node case.
Name

kcalloc — allocate memory for an array. The memory is set to zero.

Synopsis

```c
void *kcalloc (size_t n,
    size_t size,
    gfp_t flags);
```

Arguments

- `n`
  - number of elements.
- `size`
  - element size.
- `flags`
  - the type of memory to allocate.

Description

The `flags` argument may be one of:
- `KUSER` - Allocate memory on behalf of user. May sleep.
- `USER` - Allocate normal kernel ram. May sleep.
- `DIRECT` - Allocation will not sleep. May use emergency pools. For example, use this inside interrupt handlers.
- `HIGHUSER` - Allocate pages from high memory.
- `GFP_FREEZE` - Do not do any I/O or allocate memory while trying to get memory.
That’s what you also get...