Device Drivers Dummies

Rathod Jagadish

Context

Referance page

All these three chapters are covered from Linux Kernel Development which was author by Robert Love.

Chapter 1 introduces the concept of the process, one of the fundamental abstractions operating systems.

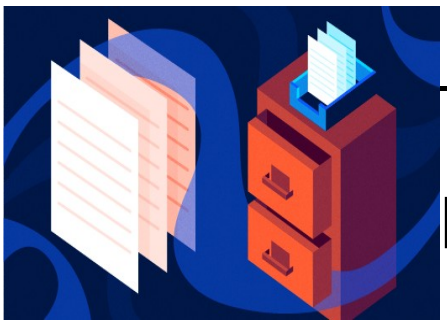
Chapter 2 introduces the concept of the system calls, The interfaces act as the messengers between applications and the kernel, with the applications issuing various requests and the kernel fulfilling them.

Chapter 3 introduces the concept of the memory management, The methods used to obtain memory inside the kernel.

Acknowledgement

I would like to express my deep gratitude to Prof. **Sankaran Vaidyanathan**, my research supervisors, for their patient guidance, enthusiastic encouragement and useful critiques of this research work.I would also like to thank Jhanavi, for her advice and assistance in keeping my progress on schedule. I would like thanks to prof. Sankaran sir for sorting my resources for making “**Device Drivers for Dummies**”.

Finally, I wish to thank **ctrl+C** and **ctrl+V** throught the complete the “Device Drivers for Dumines”.



Chapter 1

Process Management

This chapter introduces the concept of the process, one of the fundamental abstractions operating systems. It defines the process, as well as related concepts such as threads, and then discusses how the Linux kernel manages each process: how they are numerated within the kernel, how they are created, and how they ultimately die. Because running user applications is the reason we have operating systems, the process management is a crucial part of any operating system kernel, including Linux.

The Process

A process is a program (object code stored on some media) in the midst of execution. They also include a set of resources such as open files and pending signals, internal kernel data, processor state, a memory address space with one or more memory mappings, one or more threads of execution, and a data section containing global variables. Processes, in effect, are the living result of running program code.

Threads of execution, often shortened to threads, are the objects of activity within the process. Each thread includes a unique program counter, process stack, and set of processor registers. On modern operating systems, processes provide two virtualizations: a virtualized processor and virtual memory.

A program itself is not a process; a process is an active program and related resources.

A process begins its life when, not surprisingly, it is created. In Linux, this occurs by means of the *fork()* system call, which creates a new process by duplicating an existing one.The process that calls *fork()* is the parent, whereas the new process is the child.The parent resumes execution and the child starts execution at the same place: where the call to *fork()* returns.*The fork()* system call returns from the kernel twice: once in the parent process and again in the newborn child.When a process exits, it is placed into a special zombie state that represents terminated processes until the parent calls wait() or waitpid().

Process Descriptor and the Task Structure

The kernel stores the list of processes in a circular doubly linked list called the task list. Each element in the task list is a process descriptor of the type *struct task_struct* , which is defined in *<linux/sched.h> . The process descriptor contains the data that describes the executing program —open files, the process's address space,*

pending signals, the process's state, and much more figure 1.

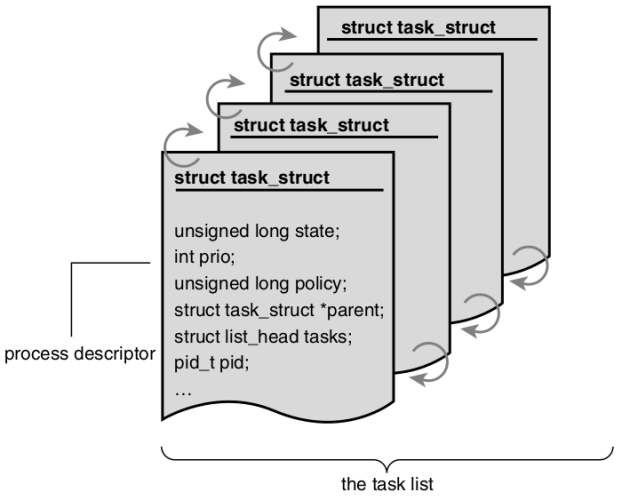


Figure 2.1. The process descriptor and task list

Allocating the Process Descriptor

The task_struct structure is allocated via the slab allocator to provide object reuse and cache coloring , to calculate the location of the process descriptor via the stack pointer without using an extra register to store the location. With the process descriptor now dynamically created via the slab allocator, a new structure, struct thread_info , was created that again lives at the bottom of the stack (for stacks that grow down) and at the top of the stack (for stacks that grow up).

The thread_info structure is defined on x86 in <asm/thread_info.h> as

```
struct thread_info {
struct task_struct       *task;
struct exec_domain       *exec_domain;
__u32                    flags;
__u32                    status;
__u32                    cpu;
int                      preempt_count;
mm_segment_t             addr_limit;
struct restart_block     restart_block;

void                     *sysenter_return;

int                      uaccess_err;
};
```

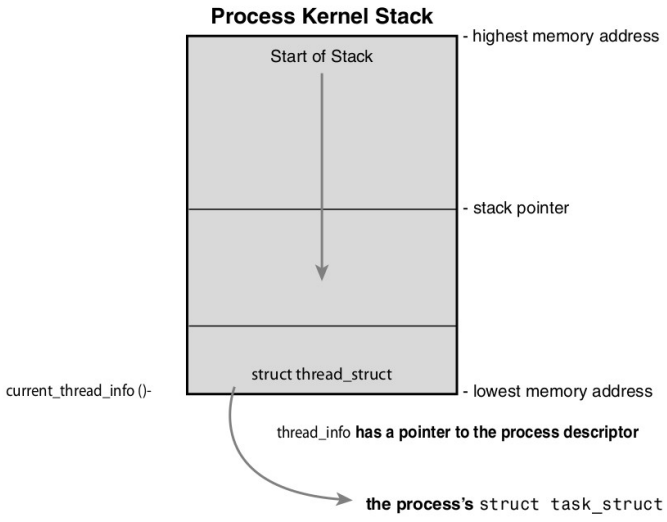


Figure 2.2 The process descriptor and kernel stack.

Each task's thread_info structure is allocated at the end of its stack.

Storing the Process Descriptor

The system identifies processes by a unique process identification value or PID.The PID is a numerical value represented by the opaque type *pid_t* , which is typically an int .

This maximum value is important because it is essentially the maximum number of processes that may exist concurrently on the system.If the system is willing to break compatibility with old applications, the administrator may increase the maximum value via */proc/sys/kernel/pid_max .*

Inside the kernel, tasks are typically referenced directly by a pointer to their task_struct structure. In fact, most kernel code that deals with processes works directly with struct task_struct .

On x86, current is calculated by masking out the 13 least-significant bits of the stack pointer to obtain the thread_info structure.This is done by the current_thread_info() function.The assembly is shown here:

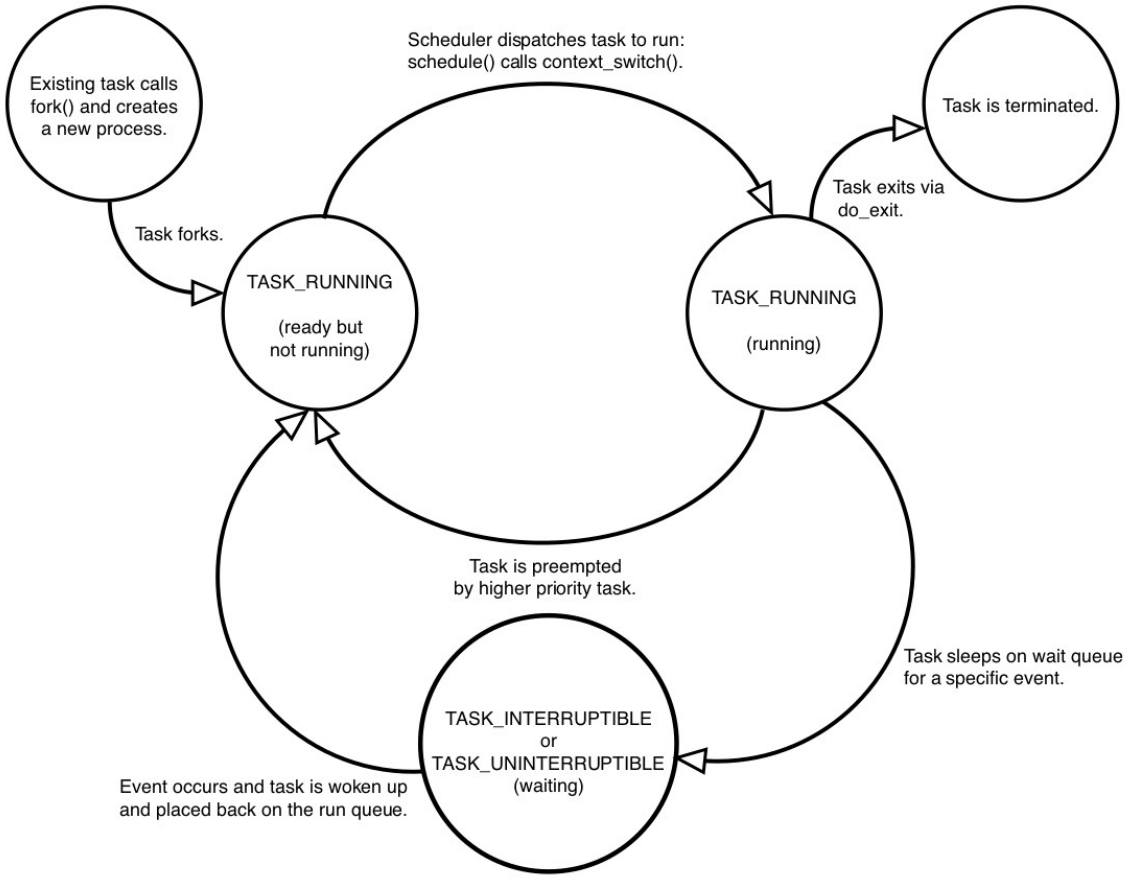
movl $-8192, %eax
andl %esp, %eax

Finally, current dereferences the task member of thread_info to return the task_struct :

current_thread_info()->task;

Process State

The state field of the process descriptor describes the current condition of the process (see Figure 2.3). Each process on the system is in exactly one of five different states.This value is represented by one of five flags:



TASK_RUNNING —The process is runnable; it is either currently running or on a run-queue waiting to run.

TASK_INTERRUPTIBLE —The process is sleeping (that is, it is blocked), waiting for some condition to exist.When this condition exists, the kernel sets the process's state to TASK_RUNNING.

TASK_UNINTERRUPTIBLE —This state is identical to TASK_INTERRUPTIBLE except that it does not wake up and become runnable if it receives a signal.

__TASK_TRACED —The process is being traced by another process, such as a debugger, via ptrace.

__TASK_STOPPED —Process execution has stopped; the task is not running nor is it eligible to run.This occurs if the task receives the SIGSTOP , SIGTSTP , SIGTTIN , or SIGTTOU signal or if it receives any signal while it is being debugged.

Process Context

One of the most important parts of a process is the executing program code.This code is read in from an executable file and executed within the program's address space.Normal program execution occurs in user-space.When a program executes a system call or triggers an exception, it enters kernel-space.

System calls and exception handlers are well-defined interfaces into the kernel.

The Process Family Tree

Every process on the system has exactly one parent. Likewise, every process has zero or more children. Processes that are all direct children of the same parent are called siblings. Consequently, given the current process, it is possible to obtain the process descriptor of its parent with the following code:

*struct task_struct *my_parent = current->parent;*

Similarly, it is possible to iterate over a process's children with

*struct task_struct *task;*
*struct list_head *list;*

The init task's process descriptor is statically allocated as init_task

*struct task_struct *task;*
for (task = current; task != &init_task; task = task->parent)

These two routines are provided by the macros *next_task(task)* and prev_task(task), resp. Finally macro for_each_process(task) is provided, which iterates over the entire task list.

*struct task_struct *task;*

Process Creation

Process creation in Unix is unique.Unix takes the unusual approach of separating these steps into two distinct functions: fork() and exec() .The first, fork() , creates a child process that is a copy of the current task. It differs from the parent only in its PID (which is unique), its PPID, The second function, exec() , loads a new executable into the address space and begins executing it.The combination of fork() followed by exec() is similar to the single function most operating systems provide.

Copy-on-Write

This approach is naive and inefficient in that it copies much data that might otherwise be shared. In Linux, fork() is implemented through the use of copy-on-write pages. Copy-on-write (or COW) is a technique to delay or altogether prevent copying of the data.

This technique delays the copying of each page in the address space until it is actually written to. In the case that the pages are never written—for example, if exec() is called immediately after fork() —they never need to be copied. The only overhead incurred by fork() is the duplication of the parent's page tables and the creation of a unique process descriptor for the child.

Forking

Linux implements fork() via the clone() system call.The *fork() , vfork()* , and *__clone()* library calls all invoke the *clone()* system call with the requisite flags.The clone() system call, in turn, calls *do_fork() .*

This function calls *copy_process()* and then starts the process running.This function calls copy_process() and then starts the process running.The interesting work is done by copy_process() :

1. It calls dup_task_struct() , which creates a new kernel stack, thread_info strucure, and task_struct for the new process.
2. It then checks that the new child will not exceed the resource limits on the number of processes for the current user.
3. The child needs to differentiate itself from its parent.Various members of the process descriptor are cleared or set to initial values.The bulk of the values in task_struct remain unchanged.
4. The child's state is set to TASK_UNINTERRUPTIBLE to ensure that it does not yet run.
5. It calls alloc_pid() to assign an available PID to the new task.
6. Depending on the flags passed to clone() , copy_process() either duplicates or shares open files, filesystem information, signal handlers, process address space, and namespace.
7. Finally, copy_process() cleans up and returns to the caller a pointer to the new child.

Vfork()

The vfork() system call has the same effect as fork() , except that the page table entries of the parent process are not copied.It is entirely possible to implement vfork() as a normal fork() —this is what
Linux did until version 2.2.
The vfork() system call is implemented via a special flag to the clone() system call:

1. In copy_process() , the task_struct member vfork_done is set to NULL .
2. In do_fork() , if the special flag was given, vfork_done is pointed at a specific address.
3. After the child is first run, the parent—instead of returning—waits for the child to signal it through the vfork_done pointer.
4. In the mm_release() function, which is used when a task exits a memory address space, vfork_done is checked to see whether it is NULL . If it is not, the parent is signaled.
5. Back in do_fork() , the parent wakes up and returns.

The Linux Implementation of Threads

Threads are a popular modern programming abstraction.They provide multiple threads of execution within the same program in a shared memory address space.They can also share open files and other resources.Threads enable concurrent programming and, on multiple processor systems, true parallelism.

The Linux kernel does not provide any special scheduling semantics or data structures to represent threads. Instead, a thread is merely a process that shares certain resources with other processes. The process descriptor describes the shared resources, such as an address space or open files.The threads then describe the resources they alone possess.

Creating Threads

Threads are created the same as normal tasks, with the exception that the clone() system

call is passed flags corresponding to the specific resources to be shared:
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);

In contrast, a normal *fork()* can be implemented as
clone(SIGCHLD, 0);

And *vfork()* is implemented as
clone(CLONE_VFORK | CLONE_VM | SIGCHLD, 0);

Kernel Threads

The kernel accomplishes this via kernel threads—standard processes that exist solely in kernel-space.The significant difference between kernel threads and normal processes is that kernel threads do not have an address space

The interface, declared in <linux/kthread.h> , for spawning a new kernel thread from an existing one is

*struct task_struct *kthread_create(int (*threadfn)(void *data),*
*void *data,*
const char namefmt[],
...)
The new task is created via the clone() system call by the kthread kernel process.A process can be created and made runnable with a single function, kthread_run() :

*struct task_struct *kthread_run(int (*threadfn)(void *data),*
*void *data,*
const char namefmt[],
...)
This routine, implemented as a macro, simply calls both *kthread_create()* and *wake_up_process()* :

```
#define kthread_run(threadfn, data, namefmt, ...)
({
struct task_struct *k;
k = kthread_create(threadfn, data, namefmt, ## __VA_ARGS__);
if (!IS_ERR(k))
wake_up_process(k);
k;
})
```

Process Termination

It is sad, but eventually processes must die.This occurs when the process receives a signal or exception it cannot handle or ignore. Regardless of how a process terminates, the bulk of the work is handled by do_exit() , defined in kernel/exit.c , which completes a number of chores:

1. It sets the PF_EXITING flag in the flags member of the task_struct .
2. It calls del_timer_sync() to remove any kernel timers. Upon return, it is guaranteed that no timer is queued and that no timer handler is running.
3. If BSD process accounting is enabled, do_exit() calls acct_update_integrals() to write out accounting information.

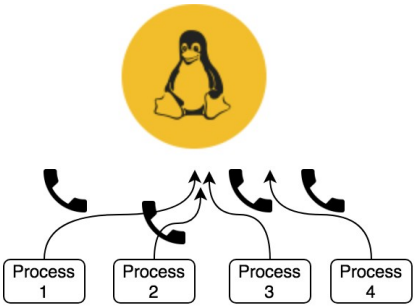
4. It calls exit_mm() to release the mm_struct held by this process. If no other process is using this address space—that it, if the address space is not shared—the kernel then destroys it.
5. It calls exit_sem() . If the process is queued waiting for an IPC semaphore, it is dequeued here.
6. It then calls exit_files() and exit_fs() to decrement the usage count of objects related to file descriptors and filesystem data, respectively. If either usage counts reach zero, the object is no longer in use by any process, and it is destroyed.
7. It sets the task's exit code, stored in the exit_code member of the task_struct , to the code provided by exit() or whatever kernel mechanism forced the termination.The exit code is stored here for optional retrieval by the parent.
8. It calls exit_notify() to send signals to the task's parent, reparents any of the task's children to another thread in their thread group or the init process, and sets the task's exit state, stored in exit_state in the task_struct structure, to EXIT_ZOMBIE .
9.calls schedule() to switch to a new process (see Chapter 4). Because the process is now not schedulable, this is the last code the task will ever execute. do_exit() never returns. do_exit()

Removing the Process Descriptor

After do_exit() completes, the process descriptor for the terminated process still exists, but the process is a zombie and is unable to run. The wait() family of functions are implemented via a single (and complicated) system call, wait4() .

When it is time to finally deallocate the process descriptor, release_task() is invoked. It does the following:
1. It calls __exit_signal() , which calls __unhash_process() , which in turns calls detach_pid() to remove the process from the pidhash and remove the process from the task list.

At this point, the process descriptor and all resources belonging solely to the process have been freed.



CHAPTER 2

System Calls

The interfaces act as the messengers between applications and the kernel, with the applications issuing various requests and the kernel fulfilling them (or returning an error).The existence of these interfaces, and the fact that applications are not free to directly do whatever they want, is key to providing a stable system.

Communicating with the Kernel

System calls provide a layer between the hardware and user-space processes.First, it provides an abstracted hardware interface for userspace.When reading or writing from a file. Second, system calls ensure system security and stability. Finally, a single common layer between user-space and the rest of the system allows for the virtualized system provided to processes.

APIs, POSIX, and the C Library

Applications are programmed against an Application Programming Interface (API) implemented in user-space, not directly to system calls.An API defines a set of programming interfaces used by applications.Those interfaces can be implemented as a system call, implemented through multiple system calls, or implemented without the use of system calls at all.See Figure 3.1 for an example of the relationship between a POSIX API, the C library, and system calls.

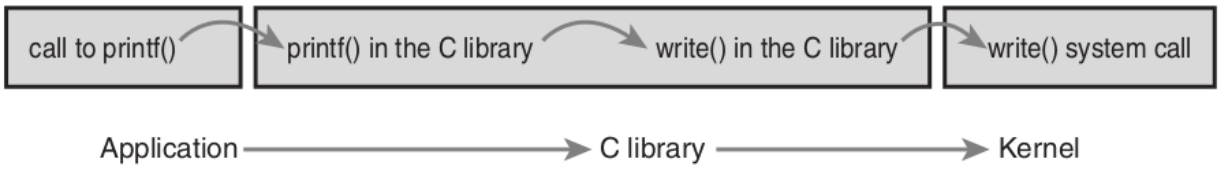


Figure 3.1. The relationship between applications, the C library, and the kernel with a call to **printf().**

One of the more common application programming interfaces in the Unix world is based on the POSIX standard. POSIX is an excellent example of the relationship between APIs and system calls.

The system call interface in Linux, as with most Unix systems, is provided in part by the C library.The C library is used by all C programs and, because of C's nature, is easily wrapped by other programming languages for use in their programs.The C library additionally provides the majority of the POSIX API.

Syscalls

System calls (often called syscalls in Linux) are typically accessed via function calls defined in the C library. A return value of zero is usually (but again not always) a sign of success.The C library, when a system call returns an error, writes a special error code into the global errno variable. This variable can be translated into human-readable errors via library functions such as *perror()* . Finally, system calls have a defined behavior. For example, the system call *getpid()* is defined to return an integer that is the current process's PID.The implementation of this syscall in the kernel is simple:

```
SYSCALL_DEFINE0(getpid)
{
return task_tgid_vnr(current); // returns current->tgid
}
```

First, note the asmlinkage modifier on the function definition.This is a directive to tell the compiler to look only on the stack for this function's arguments.This is a required modifier for all system calls. Second, the function returns a long . For compatibility between 32- and 64-bit systems, system calls defined to return an int in user-space return a long in the kernel.Third, note that the getpid() system call is defined as sys_getpid() in the kernel.This is the naming convention taken with all system calls in Linux: System call bar() is implemented in the kernel as function sys_bar() .

System Call Numbers

In Linux, each system call is assigned a syscall number.This is a unique number that is used to reference a specific system call.The syscall number is important; when assigned, it cannot change, or compiled applications will break. Linux provides a "not implemented" system call, sys_ni_syscall() , ENOSYS is used to "plug the hole" in the rare event that a syscall is removed or otherwise made unavailable.

System Call Performance

This is partly because of Linux's fast context switch times; entering and exiting the kernel is a streamlined and simple affair. The other factor is the simplicity of the system call handler and the individual system calls themselves.

System Call Handler

The mechanism to signal the kernel is a software interrupt: Incur an exception, and the system will switch to kernel mode and execute the exception handler.The exception handler, in this case, is actually the system call handler. The system call handler is the aptly named function system_call() .

Denoting the Correct System Call

The system_call() function checks the validity of the given system call number by comparing it to *NR_syscalls.I*f it is larger than or equal to *NR_syscalls* , the function returns -*ENOSYS* . Otherwise, the specified system call is invoked:

```
call *sys_call_table(,%rax,8)
```

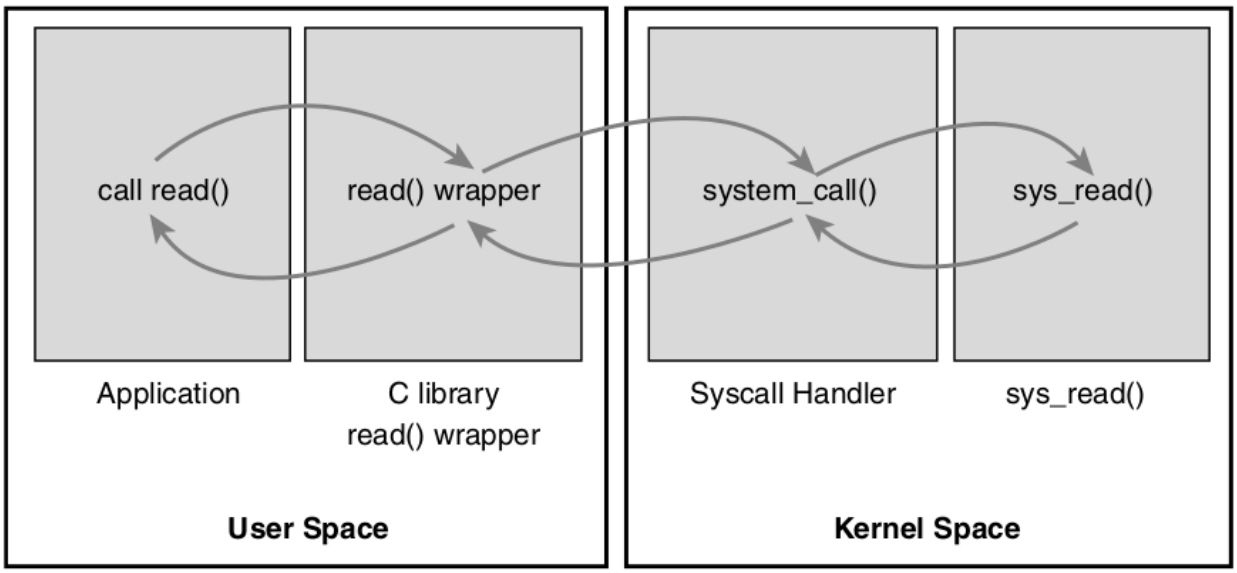


Figure 5.2 Invoking the system call handler and executing a system call.

Parameter Passing

The easiest way to do this is via the same means that the syscall number is passed: The parameters are stored in registers/the registers ebx , ecx , edx , esi , and edi contain, in order, the first five arguments. In the unlikely case of six or more arguments, a single register is used to hold a pointer to user-space where all the parameters are stored.

System Call Implementation

Thus, adding a new system call to Linux is relatively easy.The hard work lies in designing and implementing the system call; registering it with the kernel is simple.

Implementing System Calls

Multiplexing syscalls (a single system call that does wildly different things depending on a flag argument) is discouraged in Linux. Look at ioctl() as an example of what not to do.

What are the new system call's arguments, return value, and error codes? The system call should have a clean and simple interface with the smallest number of arguments possible.

Verifying the Parameters

For example, file I/O syscalls must check whether the file descriptor is valid. Process-related functions must check whether the provided PID is valid.Before following a pointer into user-space, the system must ensure that

- The pointer points to a region of memory in user-space. Processes must not be able to trick the kernel into reading data in kernel-space on their behalf.
- The pointer points to a region of memory in the process's address space.The process must not be able to trick the kernel into reading someone else's data.

- If reading, the memory is marked readable. If writing, the memory is marked writable. If executing, the memory is marked executable.The process must not be able to bypass memory access restrictions.
-

For writing into user-space, the method copy_to_user() is provided.The first is the destination memory address in the process's address space.The second is the source pointer in kernel-space. Finally, the third argument is the size in bytes of the data to copy.

For reading from user-space, the method copy_from_user() is analogous to copy_to_user() .The function reads from the second parameter into the first parameter the number of bytes specified in the third parameter.

```
SYSCALL_DEFINE3(silly_copy,
unsigned long *, src,
unsigned long *, dst,
unsigned long len)
{
 unsigned long buf;          /* copy src, which is in the user's address space, into buf */
if (copy_from_user(&buf, src, len))
return -EFAULT;             /* copy buf into dst, which is in the user's address space */
if (copy_to_user(dst, &buf, len))
return -EFAULT;             /* return amount of data copied */
return len;
}
```

Both *copy_to_user() and copy_from_user()* may block.

System Call Context

The current pointer points to the current task, which is the process that issued the syscall.In process context, the kernel is capable of sleeping (for example, if the system call blocks on a call or explicitly calls schedule()) and is fully preemptible.These two points are important. First, the capability to sleep means that system calls can make use of the majority of the kernel's functionality.

When the system call returns, control continues in system_call() , which ultimately switches to user-space and continues the execution of the user process.

Final Steps in Binding a System Call

After the system call is written, it is trivial to register it as an official system call:

1. Add an entry to the end of the system call table.This needs to be done for each architecture that supports the system call (which, for most calls, is all the architectures).The position of the syscall in the table, starting at zero, is its system call number. For example, the tenth entry in the list is assigned syscall number nine.

2. For each supported architecture, define the syscall number in <asm/unistd.h> .

3. Compile the syscall into the kernel image (as opposed to compiling as a module).

This can be as simple as putting the system call in a relevant file in kernel/ , such as sys.c , which is home to miscellaneous system calls.

Look at these steps in more detail with a fictional system call, foo() .

```
ENTRY(sys_call_table)
.long sys_restart_syscall       /* 0 */
.long sys_exit
.long sys_fork
.long sys_read
.long sys_write
.long sys_open                  /* 5 */
...
.long sys_eventfd2
.long sys_epoll_create1
.long sys_dup3                  /* 330 */
```

Accessing the System Call from User-Space

*T*he __NR_open define is in *<asm/unistd.h> ;* it is the system call number.The *_syscall3* macro expands into a C function with inline assembly; the assembly performs the steps discussed in the previous section to push the system call number and parameters into the correct registers and issue.

The number corresponds to the number of
parameters passed into the syscall because the macro needs to know how many parameters to expect and, consequently, push into registers. For example, consider the system call open() , defined as

*long open(const char *filename, int flags, int mode)*

The syscall macro to use this system call without explicit library support would be

```
#define __NR_open 5
_syscall3(long, open, const char *, filename, int, flags, int, mode)
```

Let’s write the macro to use our splendid new foo() system call and then write some test code to show off our efforts.

```
#define __NR_foo 283
__syscall0(long, foo)
int main ()
{
long stack_size;
stack_size = foo ();
printf (“The kernel stack size is %ld\n”, stack_size);
return 0;
}
```

Why Not to Implement a System Call

The pros of implementing a new interface as a syscall are as follows:

- System calls are simple to implement and easy to use.
- System call performance on Linux is fast.

The cons:

- You need a syscall number, which needs to be officially assigned to you.
- After the system call is in a stable series kernel, it is written in stone.The interface cannot change without breaking user-space applications.
- Each architecture needs to separately register the system call and support it.
- System calls are not easily used from scripts and cannot be accessed directly from the filesystem.
- Because you need an assigned syscall number, it is hard to maintain and use a system call outside of the master kernel tree.
- For simple exchanges of information, a system call is overkill.

The alternatives:

- Implement a device node and read() and write() to it. Use ioctl() to manipulate specific settings or retrieve specific information
- Certain interfaces, such as semaphores, can be represented as file descriptors and manipulated as such.
- Add the information as a file to the appropriate location in sysfs.

For many interfaces, system calls are the correct answer. Linux, however, has tried to avoid simply adding a system call to support each new abstraction that comes along.The result has been an incredibly clean system call layer with few regrets or deprecations (interfaces no longer used or supported).The slow rate of addition of new system calls is a sign that Linux is a relatively stable and feature-complete operating system.

Chapter 3

Memory Management

Memory allocation inside the kernel is not as easy as memory allocation outside the kernel. Simply put, the kernel lacks luxuries enjoyed by user-space. Unlike user-space, the kernel is not always afforded the capability to easily allocate memory.

Pages

The kernel treats physical pages as the basic unit of memory management.Although the processor's smallest addressable unit is a byte or a word, the memory management unit (MMU, the hardware that manages memory and performs virtual to physical address translations) typically deals in pages.

The kernel represents every physical page on the system with a struct page structure. This structure is defined in <linux/mm_types.h> . I've simplified the definition, removing two confusing unions that do not help color our discussion of the basics:

```
struct page {
unsigned long
flags;
atomic_t              _count;
atomic_t              _mapcount;
unsigned long         private;
struct address_space*mapping;
pgoff_t               index;
struct list_head      lru;
void                  *virtual;
};
```

Zones

The kernel uses the zones to group pages of similar properties.In particular, Linux has to deal with two shortcomings of hardware with respect to memory addressing:

- Some hardware devices can perform DMA (direct memory access) to only certain memory addresses.
- Some architectures can physically addressing larger amounts of memory than they can virtually address. Consequently, some memory is not permanently mapped into the kernel address space.

Because of these constraints, Linux has four primary memory zones:

- ZONE_DMA —This zone contains pages that can undergo DMA.
- ZONE_DMA32 —Like ZOME_DMA , this zone contains pages that can undergo DMA.Unlike ZONE_DMA , these pages are accessible only by 32-bit devices. On some architectures, this zone is a larger subset of memory.
- ZONE_NORMAL —This zone contains normal, regularly mapped, pages.

- ZONE_HIGHMEM —This zone contains “high memor

Zone	Description	Physical Memory
ZONE_DMA	DMA-able pages	< 16MB
ZONE_NORMAL	Normally addressable pages	16–896MB
ZONE_HIGHMEM	Dynamically mapped pages	> 896MB

Table 3.1 Zones on x86-32

Getting Pages

The kernel provides one low-level mechanism for requesting memory, along with several interfaces to access.The core function is

*struct page * alloc_pages(gfp_t gfp_mask, unsigned int order)*

This returns a pointer to the logical address where the given physical page currently resides.

unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)

This function works the same as alloc_pages() , except that it directly returns the logical address of the first requested page. If you need only one page, two functions are implemented as wrappers to save you a bit of typing:

*struct page * alloc_page(gfp_t gfp_mask)*
unsigned long __get_free_page(gfp_t gfp_mask)

Getting Zeroed Pages

All data must be zeroed or otherwise cleaned before it is returned to user-space to ensure system security is not compromised.Table 3.2 is a listing of all the low-level page allocation methods.

Flag	Description
alloc_page(gfp_mask)	Allocates a single page and returns a pointer to its
alloc_pages(gfp_mask, order)	Allocates 2^{order} pages and returns a pointer to the first page’s page structure
__get_free_page(gfp_mask)	Allocates a single page and returns a pointer to its logical address
__get_free_pages(gfp_mask,	Allocates 2^{order} pages and returns a pointer to the first page’s logical address
get_zeroed_page(gfp_mask)	Allocates a single page, zero its contents and returns a pointer to its logical address

Table 3.2 Low-Level Page Allocation Methods

Freeing Pages

A family of functions enables you to free allocated pages when you no longer need them:

*void __free_pages(struct page *page, unsigned int order)*
void free_pages(unsigned long addr, unsigned int order)
void free_page(unsigned long addr)

These low-level page functions are useful when you need page-sized chunks of physically contiguous pages, especially if you need exactly a single page or two. For more general byte-sized allocations, the kernel provides kmalloc() .

Kmalloc()

The kmalloc() function's operation is similar to that of user-space's familiar malloc() routine, with the exception of the additional flags parameter.The kmalloc() function is a simple interface for obtaining kernel memory in byte-sized chunks.
The function is declared in <linux/slab.h> :

*void * kmalloc(size_t size, gfp_t flags)*

If the kmalloc() call succeeds, p now points to a block of memory that is at least the requested size.The GFP_KERNEL flag specifies the behavior of the memory allocator while trying to obtain the memory to return to the caller of kmalloc() .

gfp_mask Flags

The flags are broken up into three categories: action modifiers, zone modifiers, and types.Action modifiers specify how the kernel is supposed to allocate the requested memory. .Zone modifiers specify from which of these zones to allocate. Type flags specify a combination of action and zone modifiers as needed by a certain type of memory allocation.

Kfree()

The counterpart to kmalloc() is kfree() , which is declared in <linux/slab.h> :
*void kfree(const void *ptr)*

The kfree() method frees a block of memory previously allocated with kmalloc() .Doing so is a bug, resulting in bad behavior such as freeing memory belonging to another part of the kernel.The preprocessor macro BUF_SIZE is the size in bytes of this desired buffer, which is presumably larger than just a couple of bytes.

```
char            *buf;
buf = kmalloc(BUF_SIZE, GFP_ATOMIC);
if (!buf)                   /* error allocating memory ! */
```

Later, when you no longer need the memory, do not forget to free it:

```
kfree(buf);
```

vmalloc()

The vmalloc() function works in a similar fashion to kmalloc() , except it allocates memory that is only virtually contiguous and not necessarily physically contiguous. The kmalloc() function guarantees that the pages are physically contiguous (and virtually contiguous).The vmalloc() function ensures only that the pages are contiguous within the virtual address space.

The most kernel code uses kmalloc() and not vmalloc().The vmalloc() function, to make nonphysically contiguous pages contiguous in the virtual address space, must specifically set up the page table entries.Worse,pages obtained via vmalloc() must be mapped by their individual pages, when modules are dynamically inserted into the kernel, they are loaded into memory created via vmalloc() .

The vmalloc() function is declared in <linux/vmalloc.h> and defined in mm/vmalloc.c . Usage is identical to user-space's malloc() :

*void * vmalloc(unsigned long size)*

The function might sleep and thus cannot be called from interrupt context or other situations in which blocking is not permissible.

*void vfree(const void *addr)*

This function frees the block of memory beginning at addr that was previously allo-
cated via vmalloc().

```
char *buf;
buf = vmalloc(16 * PAGE_SIZE); /* get 16 pages */
if (!buf)
/* error! failed to allocate memory */
/*
* buf now points to at least a 16*PAGE_SIZE bytes
* of virtually contiguous block of memory
*/
```

After you finish with the memory, make sure to free it by using

vfree(buf);

Slab Layer

The concept of a slab allocator was first implemented in Sun Microsystem's SunOS 5.4 operating system. 5 The Linux data structure caching layer shares the same name and basic design.
The slab layer attempts to leverage several basic tenets:

- Frequently used data structures tend to be allocated and freed often, so cache them.
- Frequent allocation and deallocation can result in memory fragmentation (the inability to find large contiguous chunks of available memory).To prevent this, the cached free lists are arranged contiguously. Because freed data structures return to the free list, there is no resulting fragmentation.

- The free list provides improved performance during frequent allocation and deallocation because a freed object can be immediately returned to the next allocation.
- If the allocator is aware of concepts such as object size, page size, and total cache size, it can make more intelligent decisions.
- If part of the cache is made per-processor (separate and unique to each processor on the system), allocations and frees can be performed without an SMP lock.
- If the allocator is NUMA-aware, it can fulfill allocations from the same memory node as the requestor.
- Stored objects can be colored to prevent multiple objects from mapping to the same cache lines.

The slab layer in Linux was designed and implemented with these premises in mind.

Design of the Slab Layer

The slab layer divides different objects into groups called caches, each of which stores a different type of object.There is one cache per object type, whereas another cache is for inode objects (struct inode) . Interestingly, the kmalloc() interface is built on top of the slab layer, using a family of general purpose caches.

The caches are then divided into slabs (hence the name of this subsystem).The slabs are composed of one or more physically contiguous pages. Each cache may consist of multiple slabs.

Each slab contains some number of objects, which are the data structures being cached. Each slab is in one of three states: full, partial, or empty.A full slab has no free objects. (All objects in the slab are allocated.) An empty slab has no allocated objects. (All objects in the slab are free.) A partial slab has some allocated objects and some free objects.

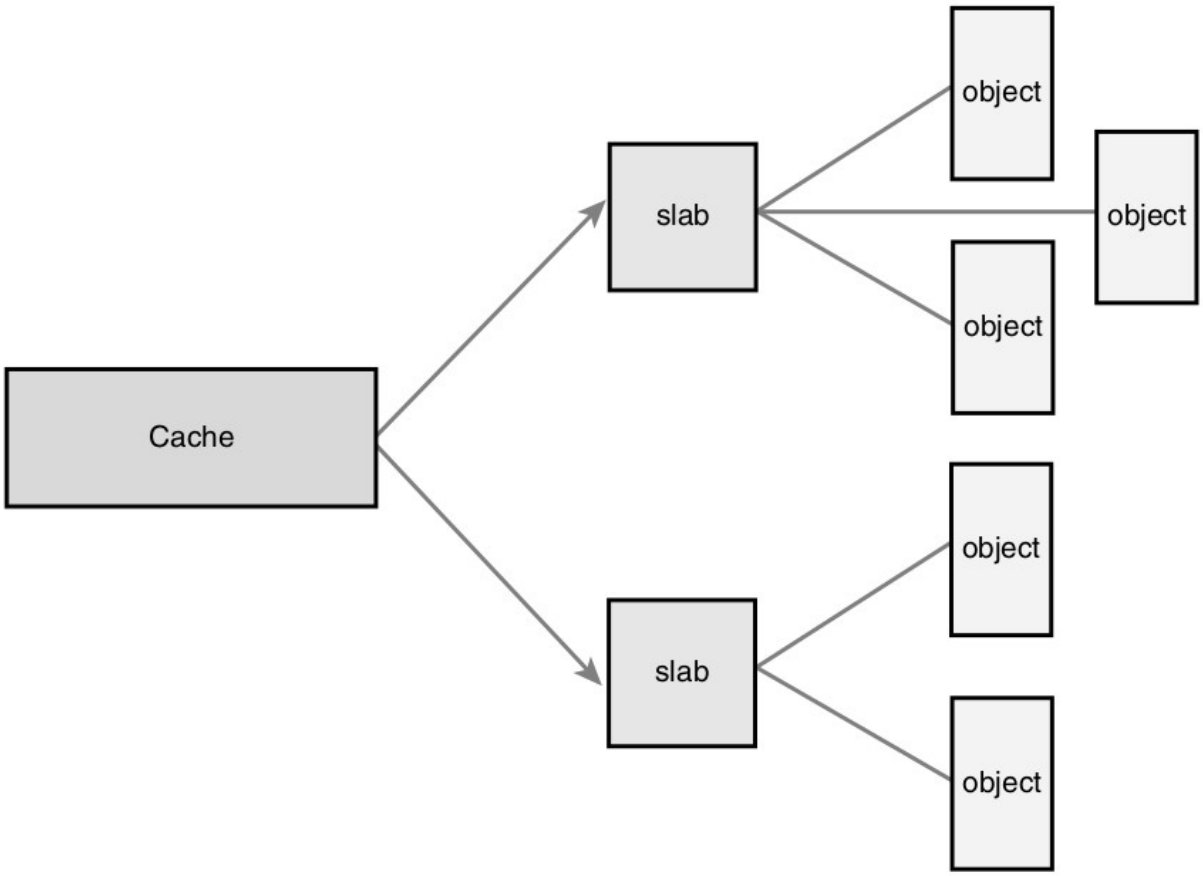


Figure 3.1 The relationship between caches, slabs, and objects.

This structure contains three lists— slabs_full , slabs_partial , and slabs_empty —stored inside a kmem_list3 structure, which is defined in mm/slab.c .These lists contain all the slabs associated with the cache.A slab descriptor, struct slab , represents each slab:

```
struct slab {
struct list_head    list;       /* full, partial, or empty list */
unsigned long       colouroff;  /* offset for the slab coloring */
void                s_mem;      /* first object in the slab */
unsigned in         inuse;      /* allocated objects in the slab */
kmem_bufctl_t       free;       /* first free object, if any */
};
```

The slab allocator creates new slabs by interfacing with the low-level kernel page allocator via __get_free_pages() :

```
static void *kmem_getpages(struct kmem_cache *cachep, gfp_t flags, int nodeid)
{
        struct page *page;
        void *addr;
        int i;
        flags |= cachep->gfpflags;
        if (likely(nodeid == -1)) {
                addr = (void*)__get_free_pages(flags, cachep->gfporder);
                if (!addr)
                        return NULL;
                page = virt_to_page(addr);
```

```
        } else {
                page = alloc_pages_node(nodeid, flags, cachep->gfporder);
                if (!page)
                        return NULL;
                addr = page_address(page);
        }
        i = (1 << cachep->gfporder);
        if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
                atomic_add(i, &slab_reclaim_pages);
        add_page_state(nr_slab, i);
        while (i—) {
                SetPageSlab(page);
                page++;
        }
        return addr;
}
```

This provides better performance on NUMA systems, in which accessing memory outside your node results in a performance penalty. For educational purposes, we can ignore the NUMA-aware code and write a simple kmem_getpages() :

```
static inline void * kmem_getpages(struct kmem_cache *cachep, gfp_t flags)
{
void *addr;
flags |= cachep->gfpflags;
addr = (void*) __get_free_pages(flags, cachep->gfporder);
return addr;
}
```

Memory is then freed by kmem_freepages() , which calls free_pages() on the given cache's pages. The sophisticated management of caches and the slabs within is entirely handled by the internals of the slab layer.After you create a cache, the slab layer works just like a specialized allocator for the specific type of object

Slab Allocator Interface

A new cache is created via

```
struct kmem_cache * kmem_cache_create(const char *name,
                                      size_t size,
                                      size_t align,
                                      unsigned long flags,
                                      void (*ctor)(void *));
```

The first parameter is a string storing the name of the cache.The second parameter is the size of each element in the cache.The third parameter is the offset of the first object within a slab.This is done to ensure a particular alignment within the page.

For frequently used caches in performance-critical code, setting this option is a good idea; otherwise, think twice.

- SLAB_POISON —This flag causes the slab layer to fill the slab with a known value (a5a5a5a5).This is called poisoning and is useful for catching access to uninitialized memory.
- SLAB_RED_ZONE —This flag causes the slab layer to insert "red zones" around the

allocated memory to help detect buffer overruns.

- SLAB_PANIC —This flag causes the slab layer to panic if the allocation fails.This flag is useful when the allocation must not fail, as in, say, allocating the VMA structure cache (see Chapter 15,“The Process Address Space”) during bootup.
- SLAB_CACHE_DMA —This flag instructs the slab layer to allocate each slab in DMA-able memory.This is needed if the allocated object is used for DMA and must reside in ZONE_DMA . Otherwise, you do not need this and you should not set it.

On success, kmem_cache_create() returns a pointer to the created cache. Otherwise, it returns NULL .This function must not be called from interrupt context because it can sleep.To destroy a cache, call

*int kmem_cache_destroy(struct kmem_cache *cachep)*

The caller of this function must ensure two conditions are true prior to invoking this function:

- All slabs in the cache are empty. Indeed, if an object in one of the slabs were still allocated and in use, how could the cache be destroyed?
- No one accesses the cache during (and obviously after) a call to *kmem_cache_destroy()* .The caller must ensure this synchronization.

Allocating from the Cache

After a cache is created, an object is obtained from the cache via

*void * kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)*
This function returns a pointer to an object from the given cache cachep . If no free objects are in any slabs in the cache, and the slab layer must obtain new pages via kmem_getpages() , the value of flags is passed to __get_free_pages() .

Example of Using the Slab Allocator

During kernel initialization, in fork_init() , defined in kernel/fork.c , the cache is created:

```
task_struct_cachep = kmem_cache_create(“task_struct”,
                                       sizeof(struct task_struct),
                                       ARCH_MIN_TASKALIGN,
                                       SLAB_PANIC | SLAB_NOTRACK,
                                       NULL);
```

Each time a process calls fork(), a new process descriptor must be created (recall Chapter3, “Process Management”).This is done in dup_task_struct() , which is called from do_fork() :

```
struct task_struct *tsk;
tsk = kmem_cache_alloc(task_struct_cachep, GFP_KERNEL);
if (!tsk)
return NULL;
```

After a task dies, if it has no children waiting on it, its process descriptor is freed and returned to the task_struct_cachep slab cache.This is done in free_task_struct() (in which tsk is the exiting task):

```
kmem_cache_free(task_struct_cachep, tsk);
```

Because process descriptors are part of the core kernel and always needed, the cache is never destroyed. If it were, however, you would destroy the cache via task_struct_cachep

int err;
err = kmem_cache_destroy(task_struct_cachep);
if (err)
/ error destroying cache */*

Statically Allocating on the Stack

User-space is afforded the luxury of a large, dynamically growing stack, whereas the kernel has no such luxury—the kernel's stack is small and fixed.When each process is given a small, fixed stack, memory consumption is minimized, and the kernel need not burden itself with stack management code

The size of the per-process kernel stacks depends on both the architecture and a compile-time option. Historically, the kernel stack has been two pages per process.This is usually 8KB for 32-bit architectures and 16KB for 64-bit architectures because they usually have 4KB and 8KB pages, respectively.

Single-Page Kernel Stacks

When enabled, each process is given only a single page—4KB on 32-bit architectures and 8KB on 64-bit architectures.This was done for two reasons. First, it results in a page with less memory consumption per process. Second and most important is that as uptime increases, it becomes increasingly hard to find two physically contiguous unallocated pages.

To rectify this problem, the kernel developers implemented a new feature: interrupt stacks. Interrupt stacks provide a single per-processor stack used for interrupt handlers

Playing Fair on the Stack

There is no hard and fast rule, but you should keep the sum of all local (that is, automatic) variables in a particular function to a maximum of a couple hundred bytes. Performing a large static allocation on the stack, such as of a large array or structure, is dangerous. Because the kernel does not make any effort to manage the stack, when the stack overflows, the excess data simply spills into whatever exists at the tail end of the stack.The first thing to eat it is the thread_info structure.

High Memory Mappings

By definition, pages in high memory might not be permanently mapped into the kernel's address space.Thus, pages obtained via alloc_pages() with the __GFP_HIGHMEM flag might not have a logical address.

On the x86 architecture, all physical memory beyond the 896MB mark is high memory and is not permanently or automatically mapped into the kernel's address space, despite x86 processors being capable of physically addressing up to 4GB (64GB with PAE 6) of physical RAM.After they are allocated, these pages must be mapped into the kernel's logical address space.

Permanent Mappings

To map a given page structure into the kernel's address space, use this function, declared in <linux/highmem.h> :

void *kmap(struct page *page)

This function works on either high or low memory. If the page structure belongs to a page in low memory, the page's virtual address is simply returned. If the page resides in high memory, a permanent mapping is created and the address is returned.The function may sleep, so kmap() works only in process context. Because the number of permanent mappings are limited (if not, we would not be in this mess and could just permanently map all memory), high memory should be unmapped when no longer needed.This is done via the following function, which unmaps the given page :
void kunmap(struct page *page)

Temporary Mappings

For times when a mapping must be created but the current context cannot sleep, the kernel provides temporary mappings (which are also called atomic mappings).These are a set of reserved mappings that can hold a temporary mapping.The kernel can atomically map a high memory page into one of these reserved mappings. Consequently, a temporary mapping can be used in places that cannot sleep, such as interrupt handlers, because obtaining the mapping never blocks.
Setting up a temporary mapping is done via

*void *kmap_atomic(struct page *page, enum km_type type)*
This function does not block and thus can be used in interrupt context and other places that cannot reschedule. It also disables kernel preemption, which is needed because the mappings are unique to each processor. (And a reschedule might change which task is running on which processor.)
The mapping is undone via

*void kunmap_atomic(void *kvaddr, enum km_type type)*

Per-CPU Allocations

Modern SMP-capable operating systems use per-CPU data—data that is unique to a given processor —extensively.Typically, per-CPU data is stored in an array. Each item in the array corresponds to a possible processor on the system.You declare the data as

unsigned long my_percpu[NR_CPUS];

Then you access it as

```
int cpu;
cpu = get_cpu();        /* get current processor and disable kernel preemption */
my_percpu[cpu]++;  /* ... or whatever */
printk("my_percpu on cpu=%d is %lu\n", cpu, my_percpu[cpu]);
put_cpu();              /* enable kernel preemption */
```

Kernel preemption is the only concern with per-CPU data. Kernel preemption poses two problems, listed here:

- If your code is preempted and reschedules on another processor, the cpu variable is no longer valid because it points to the wrong processor. (In general, code cannot

sleep after obtaining the current processor.)

- If another task preempts your code, it can concurrently access my_percpu on the same processor, which is a race condition

The New percpu Interface

This interface generalizes the previous example. Creation and manipulation of per-CPU data is simplified with this new approach.

The previously discussed method of creating and accessing per-CPU data is still valid and accepted.This new interface, however, grew out of the needs for a simpler and more powerful method for manipulating per-CPU data on large symmetrical multiprocessing computers.

The header <linux/percpu.h> declares all the routines.You can find the actual definitions there, in mm/slab.c , and in <asm/percpu.h> .

Per-CPU Data at Compile-Time

This creates an instance of a variable of type type , named name , for each processor on the system. If you need a declaration of the variable elsewhere, to avoid compile warnings, the following macro is your friend:

DECLARE_PER_CPU(type, name);
You can manipulate the variables with the get_cpu_var() and put_cpu_var() routines.A call to get_cpu_var() returns an lvalue for the given variable on the current processor. It also disables preemption, which put_cpu_var() correspondingly enables.

```
get_cpu_var(name)++;          /* increment name on this processor */
put_cpu_var(name);            /* done; enable kernel preemption */
```

You can obtain the value of another processor’s per-CPU data, too:

```
per_cpu(name, cpu)++;         /* increment name on the given processor */
```

Per-CPU Data at Runtime

The kernel implements a dynamic allocator, similar to kmalloc() , for creating per-CPU data.This routine creates an instance of the requested memory for each processor on the systems.The prototypes are in <linux/percpu.h> :

```
void *alloc_percpu(type); /* a macro */
void *__alloc_percpu(size_t size, size_t align);
void free_percpu(const void *);
```

A call to alloc_percpu() or __alloc_percpu() returns a pointer, which is used to indirectly reference the dynamically created per-CPU data.The kernel provides two macros to make this easy:

```
get_cpu_var(ptr);             /* return a void pointer to this processor’s copy of ptr */
put_cpu_var(ptr);             /* done; enable kernel preemption */
```

Let’s look at a full example of using these functions. Of course, this example is a bit silly because you would normally allocate the memory once (perhaps in some initialization function), use it in various places, and free it once (perhaps in some shutdown function). Nevertheless, this example should make usage quite clear:

```
void *percpu_ptr;
unsigned long *foo;
percpu_ptr = alloc_percpu(unsigned long);
if (!ptr)
        /* error allocating memory .. */
foo = get_cpu_var(percpu_ptr);
        /* manipulate foo .. */
put_cpu_var(percpu_ptr);
```

Reasons for Using Per-CPU Data

There are several benefits to using per-CPU data.The first is the reduction in locking requirements. Depending on the semantics by which processors access the per-CPU data, you might not need any locking at all. Keep in mind that the “only this processor accesses this data” rule is only a programming convention.You need to ensure that the local processor accesses only its unique data. Nothing stops you from cheating. Second, per-CPU data greatly reduces cache invalidation.This occurs as processors try to keep their caches in sync. If one processor manipulates data held in another processor’s cache, that processor must flush or otherwise update its cache. Constant cache invalidation is called thrashing the cache and wreaks havoc on system performance.The use of per-CPU data keeps cache effects to a minimum because processors ideally access only their own data.The percpu interface cache-aligns all data to ensure that accessing one processor’s data does not bring in another processor’s data on the same cache line.

Picking an Allocation Method

If you want to allocate from high memory, use alloc_pages() .The alloc_pages() function returns a struct page and not a pointer to a logical address. Because high memory might not be mapped, the only way to access it might be via the corresponding struct page structure.To obtain an actual pointer, use kmap() to map the high memory into the kernel’s logical address space. I

f you do not need physically contiguous pages—only virtually contiguous—use vmalloc() , although bear in mind the slight performance hit taken with vmalloc() over kmalloc() .The vmalloc() function allocates kernel memory that is virtually contiguous but not, per se, physically contiguous. It performs this feat much as user-space allocations do, by mapping chunks of physical memory into a contiguous logical address space.

If you are creating and destroying many large data structures, consider setting up a slab cache.The slab layer maintains a per-processor object cache (a free list), which might greatly enhance object allocation and deallocation performance. Rather than frequently allocate and free memory, the slab layer stores a cache of already allocated objects for you. When you need a new chunk of memory to hold your data structure, the slab layer often does not need to allocate more memory and instead simply can return an object from the cache.