Unit 1: Processes

This slice set is based on Prof. Shih-Kun Huang's material and the book
"Understanding the Linux Kernel"

Processes, LWP, and threads

A process

- A process is an instance of a program in execution
- In traditional Unix systems
  - Each process consists of one thread
- In modern Unix systems
  - A process is composed of several *user threads* (or simply threads) via pthread
  - The M-1 model, just like a normal process
- Linux use LWPs to support multithreaded processes
  - The 1-1 model

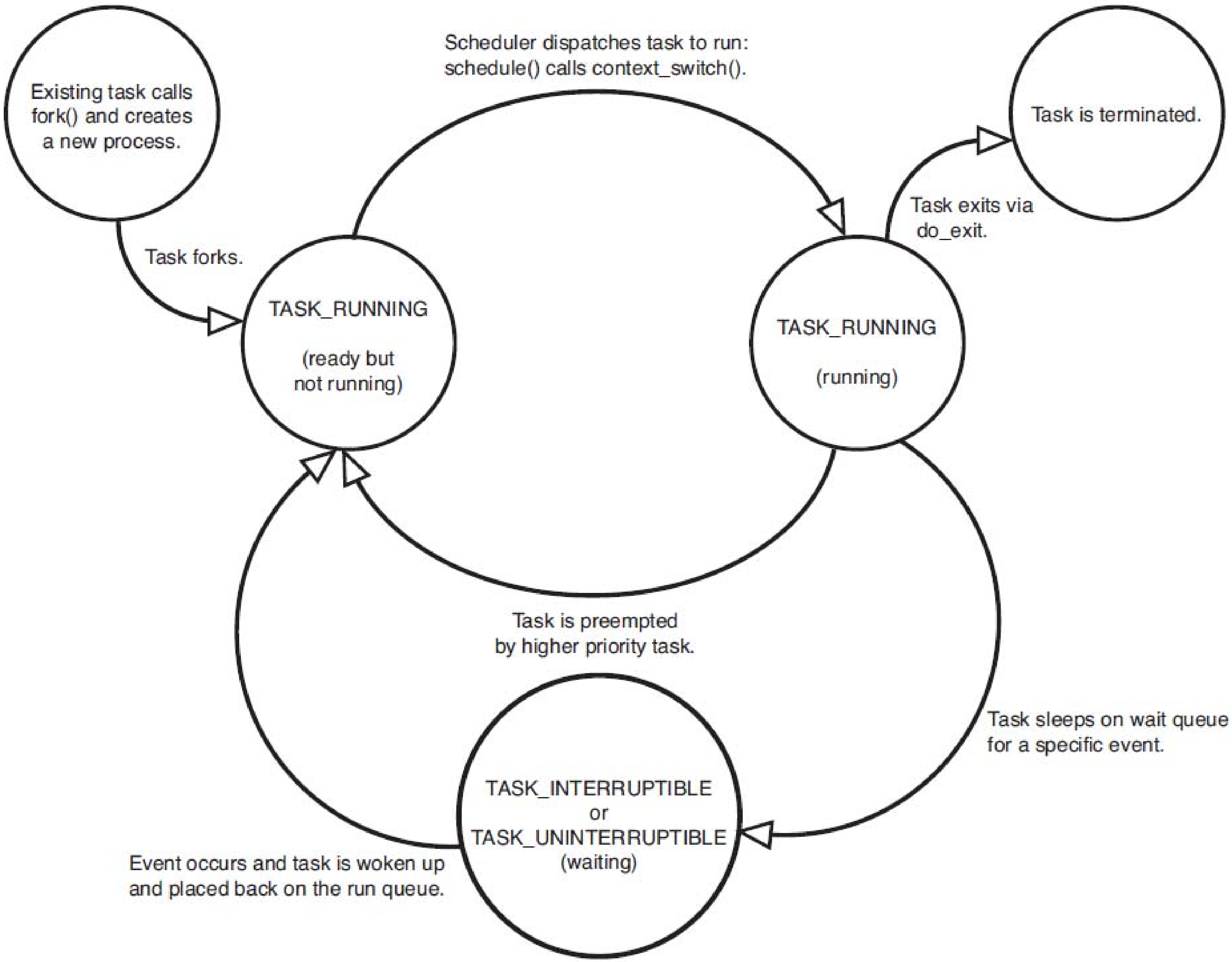


Figure 3.3 Flow chart of process states.

*Both running and ready are with the TASK_RUNNING state

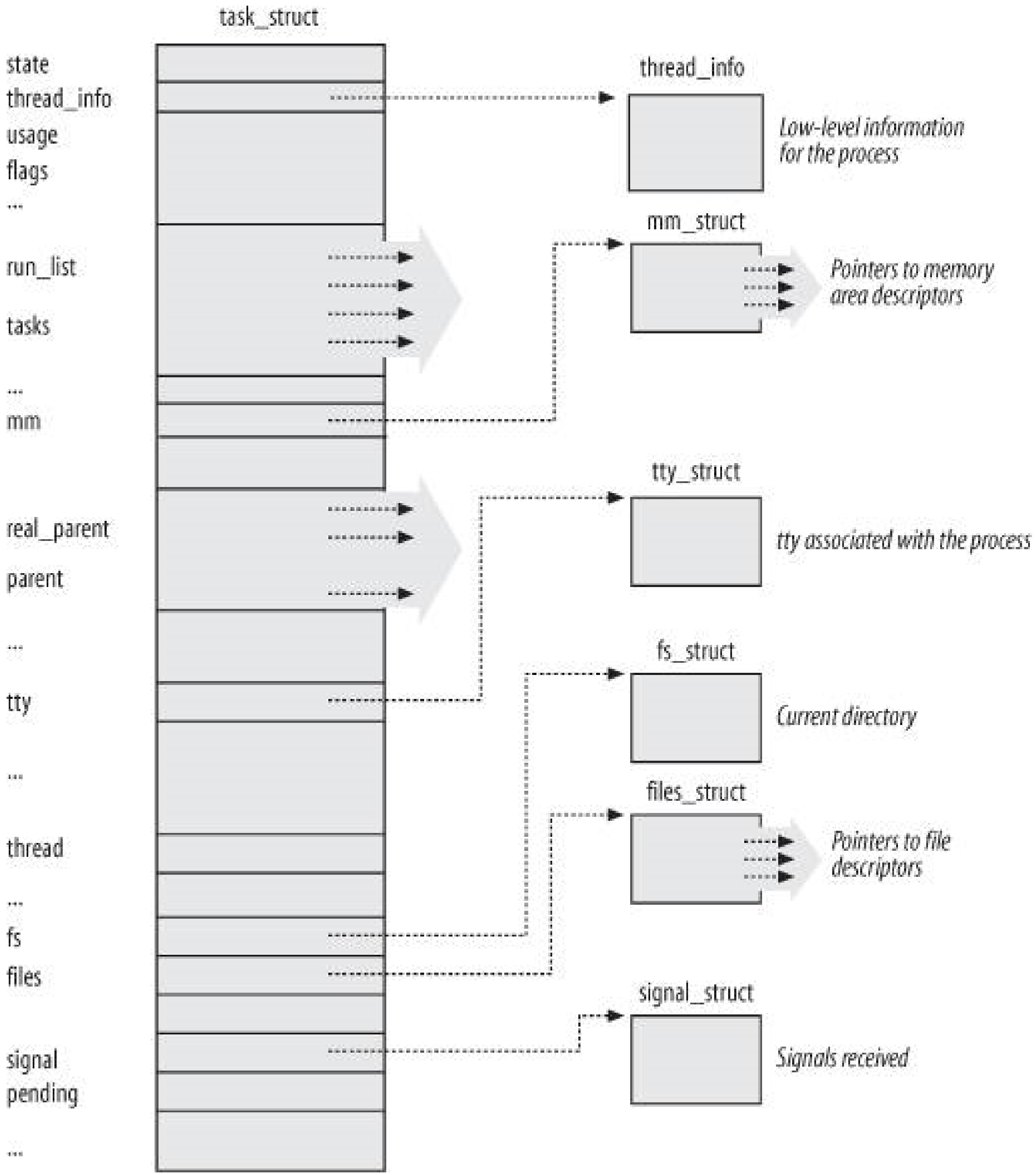
Process Descriptor

Process Descriptor

- **`task_struct`**
- To manage processes, the kernel must have a clear picture of what each process is doing
- Ex:
  - The process's priority
  - Address space
  - Running on CPU/IO

Process Descriptor Overview

- State
  - Whether it is running on a CPU or blocked on an event
- Process Identity
  - How to Identify a Process ?
- Process Relationship
  - How the process trees are organized ?


task_struct
state
thread_info
usage
flags
...
run_list
tasks
...
mm
real_parent
parent
...
tty
...
thread
...
fs
files
signal
pending
...
thread_info
Low-level information for the process
mm_struct
Pointers to memory area descriptors
tty_struct
tty associated with the process
fs_struct
Current directory
files_struct
Pointers to file descriptors
signal_struct
Signals received

Linux Process Descriptor task_struct

pid

state

exit_state

stack

- thread_info
- Kernel mode stack
- current

thread

- thread_struct
- Hardware context
- Process switch

rt

- sched_rt_entity
- rq run_list

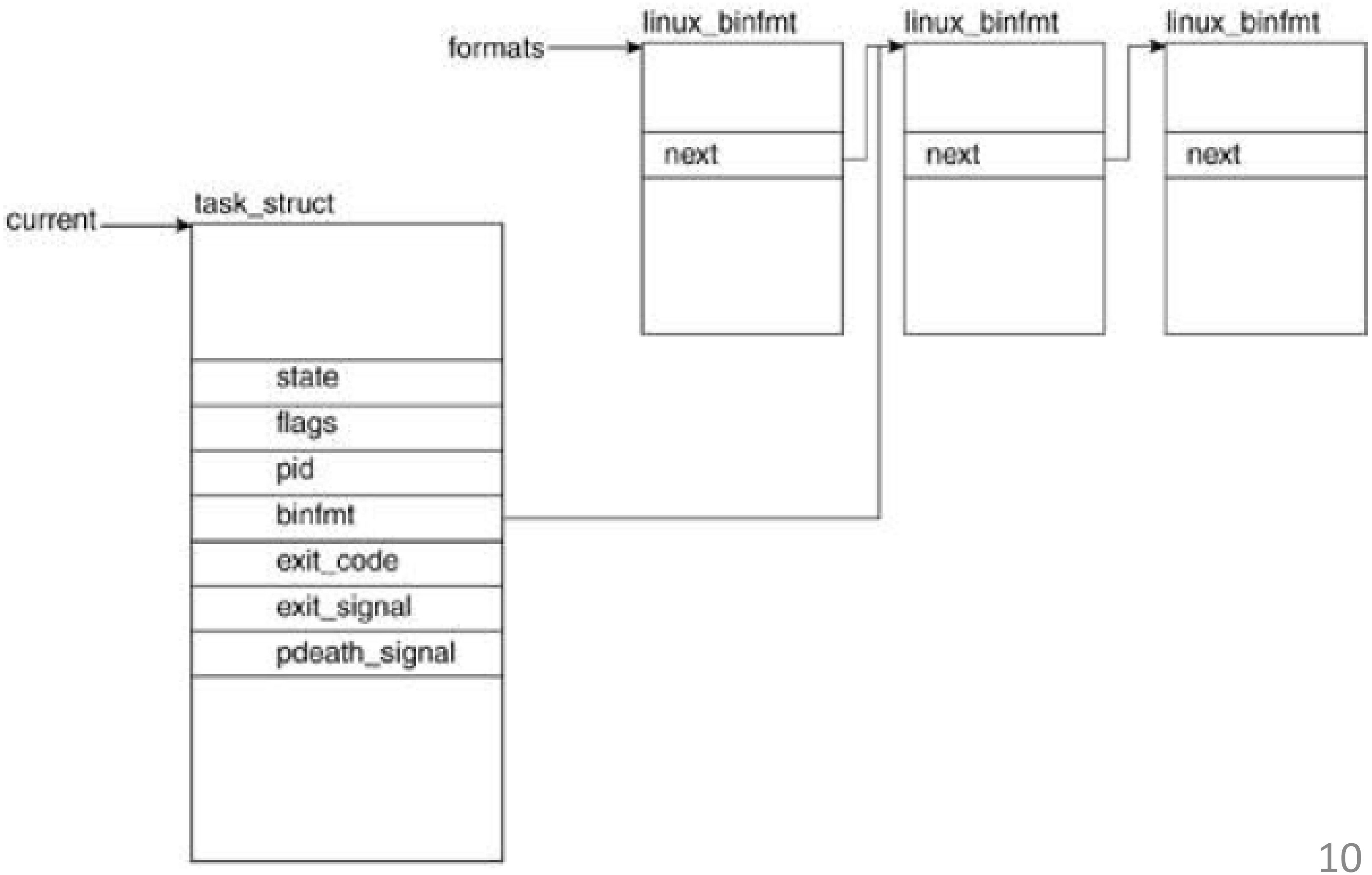
mm, active_mm

- mm_struct

parent, children, sibling

- list_head

Process Attribute Related Fields



Definition and Use (Linux 2.6.32)

- struct task_struct
  - include/linux/sched.h
  - http://lxr.free-electrons.com/source/include/linux/sched.h?v=2.6.32#L1217

Process State

- The state field of the process descriptor describes what is currently happening to the process
- It consists of an array of flags, each of which describes a possible process state
- Exactly *one* flag of state always is set (these states are mutual exclusive)

Process State

- TASK_RUNNING
  - The process is either executing on a CPU or waiting to be executed.
- TASK_INTERRUPTIBLE
  - The process is suspended (sleeping) until some condition becomes true.
  - Signals wake up the process
- TASK_UNINTERRUPTIBLE
  - Like TASK_INTERRUPTIBLE, except that delivering a signal to the sleeping process leaves its state unchanged.
  - Signals do not wake the process
  - Used when waiting for time-critical hardware interrupts

Process State

- __TASK_STOPPED
  - Process execution has been stopped (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU)
- __TASK_TRACED
  - Process execution has been stopped by a debugger.
- EXIT_ZOMBIE (related to waitpid() or wait4())
  - Process execution is terminated, but the parent process has not yet issued a wait() on the dead process.
  - Its resources have been released but its process descriptor remains in memory
- EXIT_DEAD
  - The final state: the process is being removed by the system

```
<linux/sched.h>
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define __TASK_ZOMBIE           4
#define __TASK_STOPPED          8
/* in task_exit_state */
#define EXIT_ZOMBIE
#define EXIT_DEAD
```

Changing state

- p->state = TASK_RUNNING;
  - Safe in uniprocessor systems
- set_task_state() related macros
  - For multiprocessor systems

Memory Barrier

- Memory fence or fence instruction
- Enforce an ordering constraint on memory operations before and after the barrier
  - Due to performance optimizations resulting out-of-order execution
  - Concurrent and device drivers will be with unpredictable behavior if out-of-order

Processor #1

```
while (f == 0);
// Memory fence required here
print x;
```

Processor #2

```
x = 42;
// Memory fence required here
f = 1;
```

After optimization, f=1 may be executed before x =42;

Set_task_state macro

```
#define set_task_state(tsk, state_value)                \
        set_mb((tsk)->state, (state_value))

#define set_mb(var, value) \
        do { var = value;  mb(); } while (0)
```

l

1. Why do {} while ()
2. mb()

mb(), rmb(), wmb()

```
#define mb()    asm volatile ("": : :"memory")
#define rmb()   mb()
#define wmb()   asm volatile ("": : :"memory")

#ifdef CONFIG_SMP
#define smp_mb()        mb()
#define smp_rmb()       rmb()
#define smp_wmb()        wmb()
#else
#define smp_mb()        barrier()
#define smp_rmb()       barrier()
#define smp_wmb()        barrier()
#endif
```

Compiler memory barrier

- Prevent a compiler from reordering instructions (won't prevent reordering by CPU)
- Forbit GCC to reorder read and write commands around it

```
asm volatile("" ::: "memory");
__asm__ __volatile__ ("" ::: "memory");
```

- Using spin lock to protect the process state variable also works here, but introducing much higher overhead than using memory barrier

Process Identification

Process ID

- Each execution context that can be independently scheduled must have its own process descriptor
- The PID of the previously created process increased by one
- When recycling PID numbers, the kernel recycles unused PIDs

Process ID

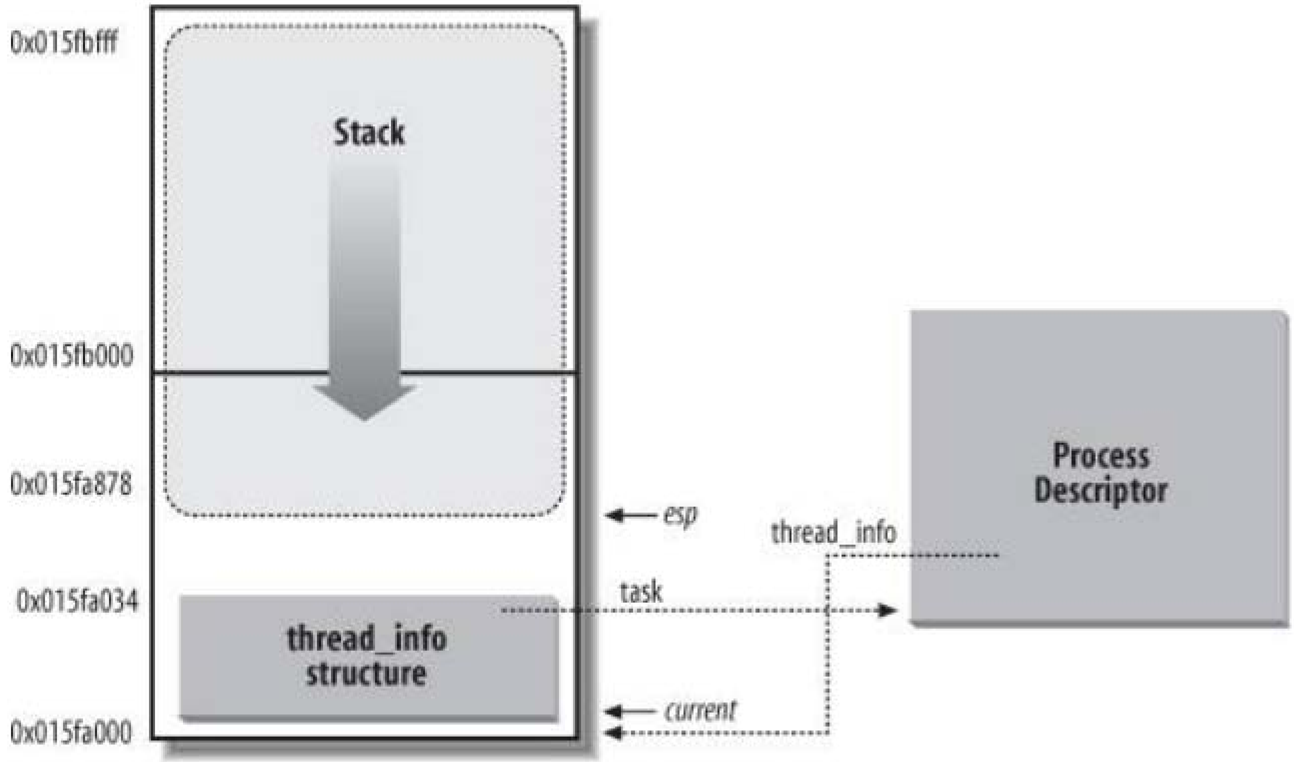
- Default max pid = 32767
- /proc/sys/kernel/pid_max
- For 64 bits, max pid= 4,194,303
- How to deal with pid recycling ?
  - pidmap_array (a page with 32768 bits) to denote used/unused pid
  - 64 bits needs more pages

Process descriptors handling

- The process descriptors are stored in dynamic memory rather than in the memory area permanently assigned to the kernel

- Linux packs two different data structures in a single per-process memory area for *efficient access*:
  - a small data structure linked to the process descriptor, the thread_info structure
  - Kernel Mode process stack
    - Why a kernel stack?
  - Occupying two physically contiguous pages

thread_info and the kernel stack



thread_info 52 bytes, kernel stack can be up to 8140 bytes

thread_info and the kernel stack

```
union thread_union {
        struct thread_info thread_info;
        unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

thread_info and the kernel stack

- The close association between the thread_info structure and the Kernel Mode stack just described offers a key benefit in terms of efficiency:
  - the kernel can easily obtain the address of the thread_info structure of the process currently running on a CPU from the value of the esp register
  - thread_info is 8k (213 bytes)

Identifying the current process

```
current_thread_info( ){
movl $0xffffe000,%ecx   /* or 0xffff000 for 4KB  stacks */
andl %esp,%ecx
movl %ecx,p
}
```

```
esp   0x015FA878
AND   0xFFFFE000
      0x015FA000
```

- Because the task field is at offset 0 in the thread_info structure, after executing these three instructions p contains the process descriptor pointer of the process running on the CPU.

- current macro is current_thread_info()->task
  - current->pid is the process id of the process currently running on the CPU
  - Earlier versions of linux, cureent is a global static variable

About “current”

Linux/include/asm-generic/current.h

```
#ifndef __ASM_GENERIC_CURRENT_H
#define __ASM_GENERIC_CURRENT_H

#include <linux/thread_info.h>

#define get_current() (current_thread_info()- >task)
#define current get_current()

#endif /* __ASM_GENERIC_CURRENT_H */

```

- “current” is a macro

Process descriptor list

- All process descriptors are linked in a circular list
- Link lists are very commonly used in the kernel
- Common approach: every data structure needs its list management routines
- Linux's approach: *embedded a list head* to data structures and use generic list management routines

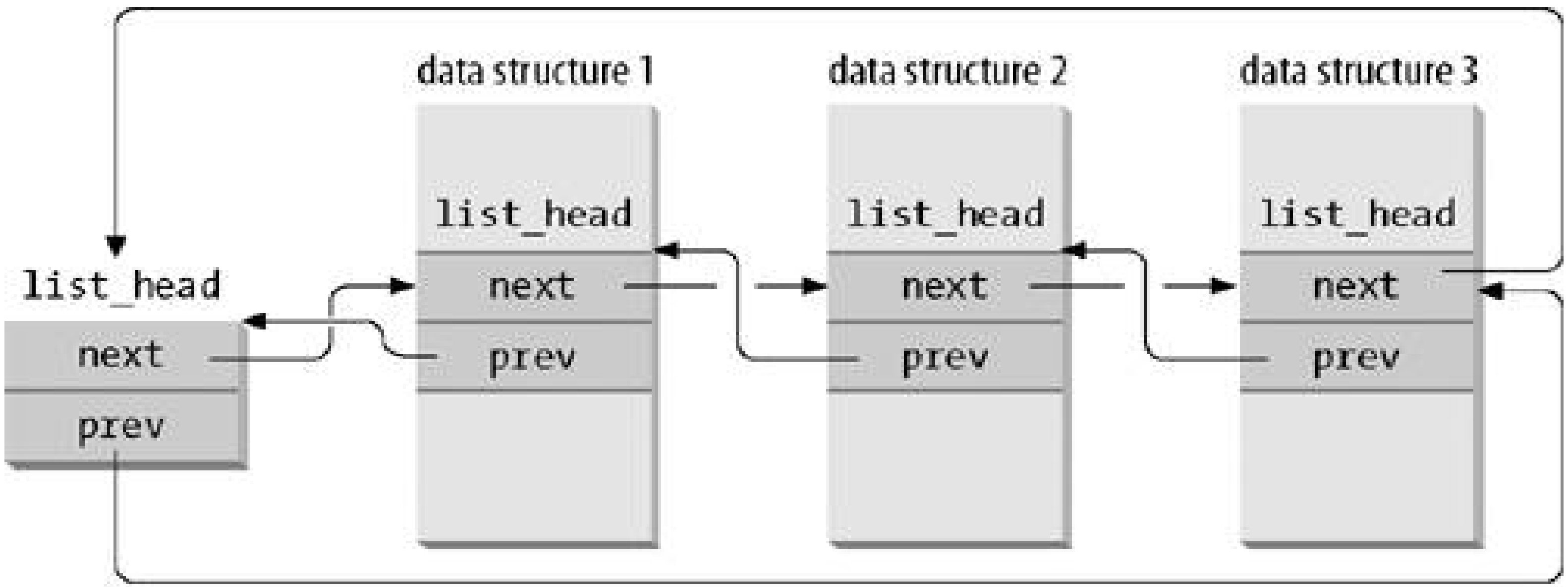
Doubly linked lists

- A new list is created using the LIST_HEAD(list_name) macro.
- It declares a new variable named list_name of type list_head
- Other list handling function and macros name:

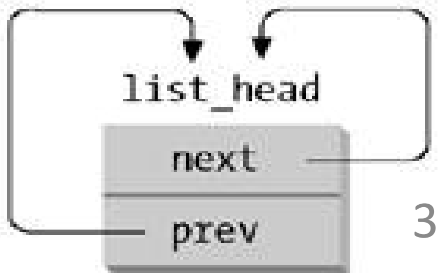
Table 3-1. List handling functions and macros

Name	Description
`list_add(n,p)`	Inserts an element pointed to by `n` right after the specified element pointed to by `p`. (To insert `n` at the beginning of the list, set `p` to the address of the list head.)
`list_add_tail(n,p)`	Inserts an element pointed to by `n` right before the specified element pointed to by `p`. (To insert `n` at the end of the list, set `p` to the address of the list head.)
`list_del(p)`	Deletes an element pointed to by `p`. (There is no need to specify the head of the list.)
`list_empty(p)`	Checks if the list specified by the address `p` of its head is empty.
`list_entry(p,t,m)`	Returns the address of the data structure of type `t` in which the `list_head` field that has the name `m` and the address `p` is included.
`list_for_each(p,h)`	Scans the elements of the list specified by the address `h` of the head; in each iteration, a pointer to the `list_head` structure of the list element is returned in `p`.
`list_for_each_entry(p,h,m)`	Similar to `list_for_each`, but returns the address of the data structure embedding the `list_head` structure rather than the address of the `list_head` structure itself.

Doubly linked lists



(a) a doubly linked listed with three elements



(b) an empty doubly linked list

The lists of TASK_RUNNING processes

- When looking for a new process to run on a CPU, the kernel has to consider *only the runnable processes* (that is, the processes in the TASK_RUNNING state)

- The trick used to achieve the scheduler speedup consists of *splitting the runqueue in many lists* of runnable processes, one list per process priority

The lists of TASK_RUNNING processes

- Each task_struct descriptor includes a run_list field of type list_head.

- If the process priority is equal to k (a value ranging between 0 and 139), the run_list field links the process descriptor into the list of runnable processes having priority k

The lists of TASK_RUNNING processes

- The main data structures of a runqueue (rq)are the lists of process descriptors belonging to the runqueue; all these lists are implemented by a single prio_array_t data structure

Type	Field	Description
int	nr_active	The number of process descriptors linked into the lists
unsigned long [5]	bitmap	A priority bitmap: each flag is set if and only if the corresponding priority list is not empty
struct list_head[140]	queue	The 140 heads of the priority lists

nr_active, nr_running, and priority bitmap

```
/*
 * Either called from update_cpu_load() or from a cpu going idle
 */
static void calc_load_account_active(struct rq *this_rq)
{
        long nr_active, delta;

        nr_active = this_rq->nr_running;
        nr_active += (long) this_rq->nr_uninterruptible;

        if (nr_active != this_rq->calc_load_active) {
                delta = nr_active - this_rq->calc_load_active;
                this_rq->calc_load_active = nr_active;
                atomic_long_add(delta, &calc_load_tasks);
        }
}
```

```
struct rt_prio_array {
        DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1); /* include 1 bit for delimiter */
        struct list_head queue[MAX_RT_PRIO];
};
```

Process Relationship

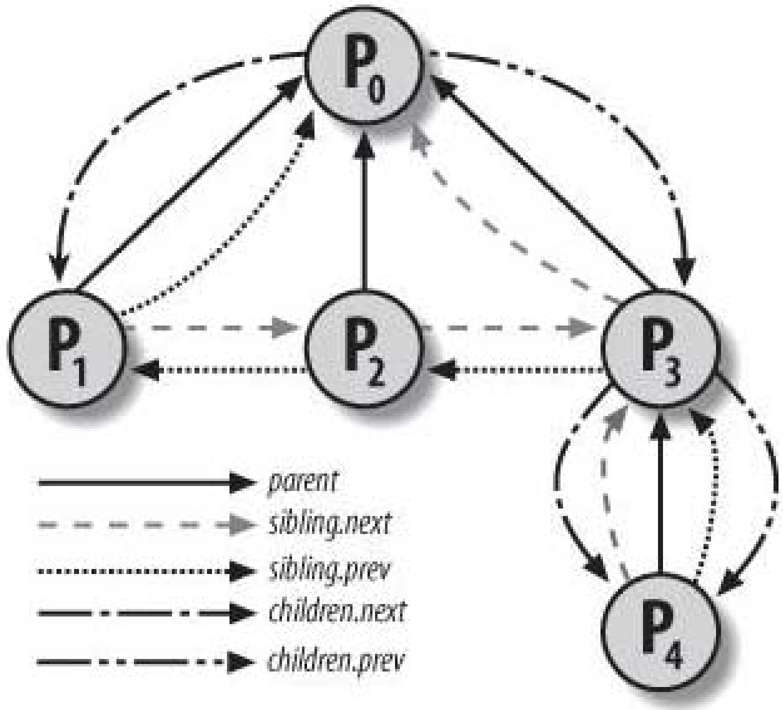
- Processes 0 and 1 are created by the kernel
  - 0: swapper
  - 1: init
    - The ancestor of all processes
    - Orphan processes will be adopted by init
    - Kernel thread become init’s children after calling daemonize()
      - Why?

Process Relationship

Fields of a process descriptor used to express parenthood relationships :

Field name	Description
real_parent	Points to the process descriptor of the process that created P or to the descriptor of process 1 (init) if the parent process no longer exists.
parent	Points to the current parent of P
children	The head of the list containing all children created by P
sibling	The pointers to the next and previous elements in the list of the sibling processes, those that have the same parent as P

Process Relationship

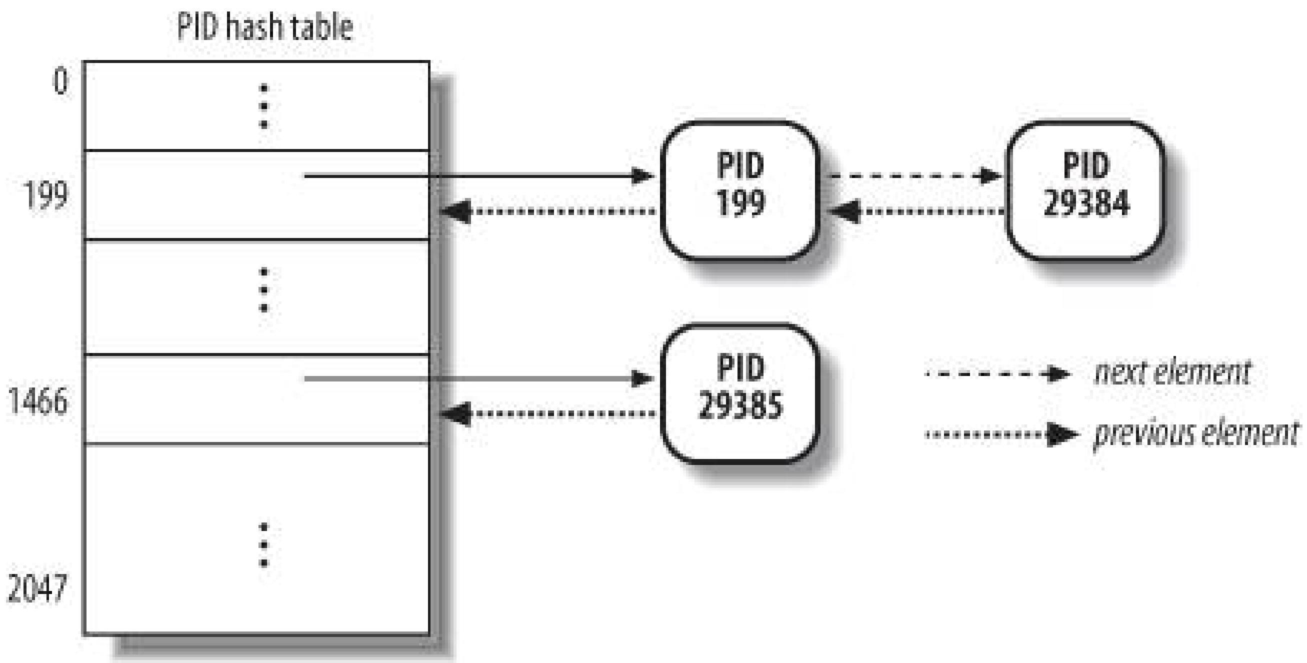


Process Relationship

- A process can have many roles
  - Itself (PID)
  - Thread group leader (TGID)
    - For multithreaded processes
  - Group leader leader (PGID)
  - Session leader (SID)
    - For login sessions

- Need to efficiently identify the process(es) corresponding to the given ID

A simple PID hash table and chained ists



Process Hash Table

- The kernel derives the process descriptor pointer from the PID by hash table
- The PID is transformed into a table index using the pid_hashfn macro, which expands to:
  -

```
#define pid_hashfn(x) hash_long((unsigned long) x, pidhash_shift)

unsigned long hash_long(unsigned long val, unsigned int bits)
 {
 unsigned long hash = val * 0x9e370001UL;
 return hash >> (32 - bits);
 }
```

Pointers to 4 hash tables

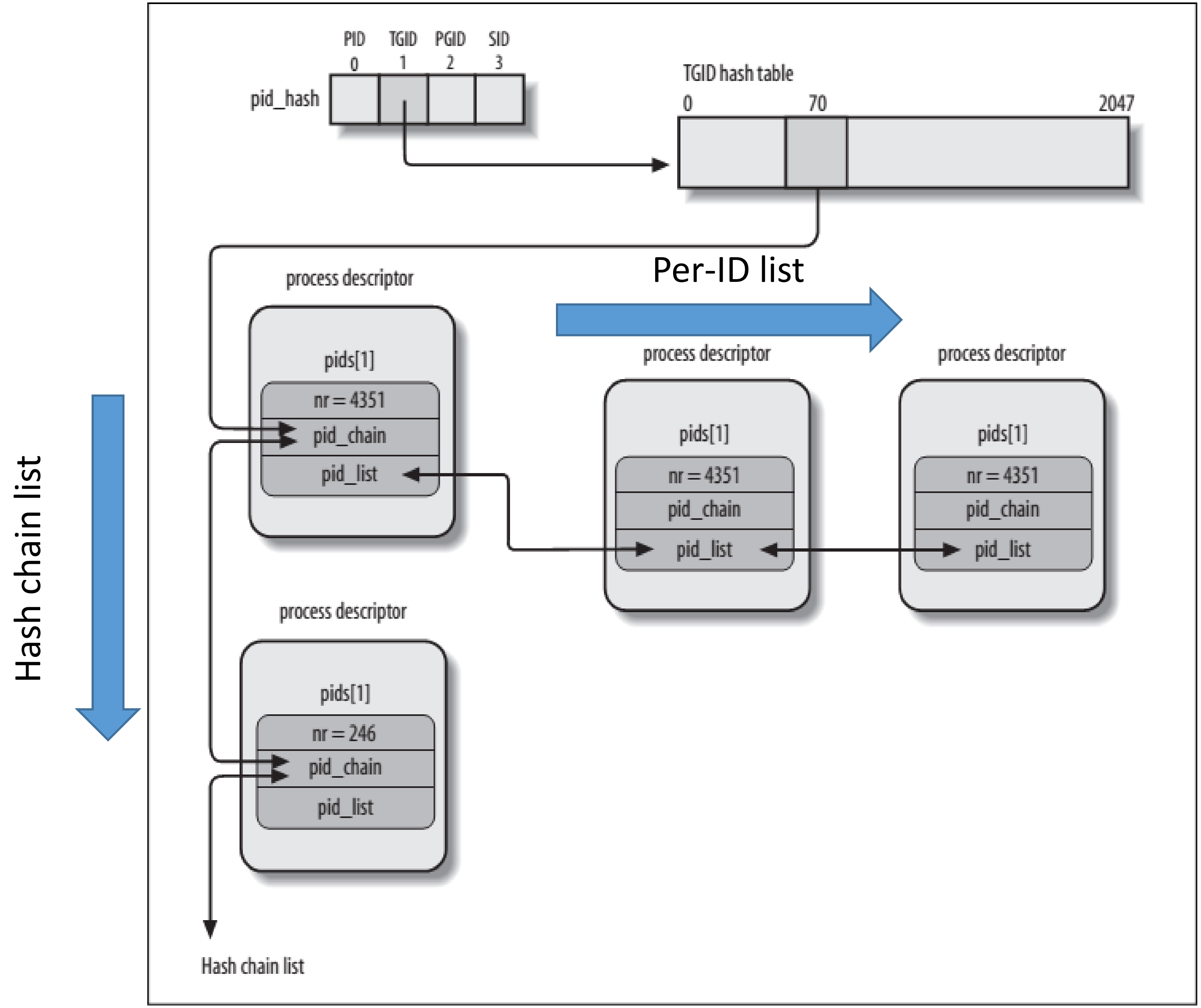


Figure 3-6. The PID hash tables

Runqueue Structure

Runqueue

- The runqueue lists group all processes in a TASK_RUNNING state
- Processes in a TASK_STOPPED, EXIT_ZOMBIE, or EXIT_DEAD state are not linked in specific lists ,they are accessed only via PID or via linked lists of the child processes for a particular parent

How Processes Are Organized

- Processes in a TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE state are subdivided into many classes, each of which corresponds to a specific event
  - A wait queue represents an event

- Wait queues have several uses in the kernel
  - Particularly for interrupt handling
  - Process synchronization
  - Timing

Wait queues

- Wait queues are implemented as doubly linked lists whose elements include pointers to process descriptors
- Each element in the wait queue list represents a sleeping process, which is waiting for some event to occur.

structure of type wait_queue_head_t:

```
struct __wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
```

The elements of a wait queue list

- The descriptor address is stored in the *task* field
- The *task_list* field contains the pointers that link this element to the list of processes waiting for the same event
- The *flag* field denotes the type of the event:
  - exclusive processes (flags = 1) // waking up process a time
  - nonexclusive processes (flags = 0) // waking up all processes
- The *func* field is used to specify how the processes should be woken up

Elements of a wait queue list are of type wait_queue_t:

```
struct __wait_queue {
    unsigned int flags;
    struct task_struct * task;
    wait_queue_func_t func;
    struct list_head task_list;
};
typedef struct __wait_queue wait_queue_t;
```

Handling wait queues

- A new wait queue head may be defined by using the DECLARE_WAIT_QUEUE_HEAD(name) macr o, which statically declares a new wait queue head.

- The init_waitqueue_head( ) function may be used to initialize a wait queue head variable that was allocated dynamically.

The init_waitqueue_entry(q,p) function initializes a wait_queue_t structure q as follows:

```
q->flags = 0;
q->task = p;
q->func = default_wake_function;
```

Handling wait queues

- A process wishing to wait for a specific condition can invoke any of the functions shown in the following list.
  - sleep_on( )
  - sleep_on_timeout( )
  - prepare_to_wait( ), prepare_to_wait_exclusive( ), and finish_wait( )
  - prepare_to_wait( ) and prepare_to_wait_exclusive( )

sleep_on() on the current process

- The sleep_on() function operates on the current process:

```
void sleep_on(wait_queue_head_t *wq)
{
    wait_queue_t wait;
    init_waitqueue_entry(&wait, current);
    current->state = TASK_UNINTERRUPTIBLE;
    add_wait_queue(wq,&wait); /* wq points to the wait queue head */
    schedule();
    remove_wait_queue(wq, &wait);
}
```

sleep_on_common

```c
sleep_on_common(wait_queue_head_t *q, int state, long timeout)
{
        unsigned long flags;
        wait_queue_t wait;

        init_waitqueue_entry(&wait, current);

        __set_current_state(state);

        spin_lock_irqsave(&q->lock, flags);
        __add_wait_queue(q, &wait);
        spin_unlock(&q->lock);
        timeout = schedule_timeout(timeout);
        spin_lock_irq(&q->lock);
        __remove_wait_queue(q, &wait);
        spin_unlock_irqrestore(&q->lock, flags);

        return timeout;
}
```

Other sleep functions

- interruptible_sleep_on(): sleep_on() with TASK_INTERRUPBILE state (can be woken up by receiving signal)
- sleep_on_timeout() and interrupbible_sleep_on_timeout(): invoke schedule_timeout() instead of schedule()
- prepare_to_wait() and prepare_to_wait_exclusive()

prepare_to_wait()

- Sometimes the wait condition had been *untrue* before a process starts sleeping
  - Insert to the wait queue
  - Check the sleep condition
    - If true, re-schedule. Otherwise, finish wait

- The prepare_to_wait(), prepare_to_wait_exclusive(), and finish_wait() functions, introduced in Linux 2.6, offer yet another way to put the current process to sleep in a wait queue. Typically, they are used as follows:

```
DEFINE_WAIT(wait);
prepare_to_wait_exclusive(&wq, &wait, TASK_INTERRUPTIBLE);
                              /* wq is the head of the wait queue */
...
if (!condition)
    schedule();
finish_wait(&wq, &wait);
```

wake_up()

```
void wake_up(wait_queue_head_t *q)
{
struct list_head *tmp;
wait_queue_t *curr;
list_for_each(tmp, &q->task_list) {
        curr = list_entry(tmp, wait_queue_t, task_list);
        if (curr->func(curr,
                TASK_INTERRUPTIBLE|TASK_UNINTERRUPTIBLE,
                 0, NULL) && curr->flags)
                break;
        }
}
```

Process Resource Limits

- Each process has an associated set of resource limits , which specify the amount of system resources it can use, such as CPU, disk space, and so on.
- The resource limits for the current process are stored in the current->signal->rlim field which is an array of elements of type struct rlimit.

```
struct rlimit {
    unsigned long rlim_cur;
    unsigned long rlim_max;
};
```

Process Resource Limits

- The rlim_cur field is the current resource limit for the resource.
- For example, current->signal->rlim[RLIMIT_CPU]. It represents the current limit on the CPU time of the running process.
- The rlim_max field is the maximum allowed value for the resource limit.

Table 3-7. Resource limits

Field name	Description
RLIMIT_AS	The maximum size of process address space, in bytes. The kernel checks this value when the process uses malloc( ) or a related function to enlarge its address space (see the section "The Process's Address Space" in Chapter 9).
RLIMIT_CORE	The maximum core dump file size, in bytes. The kernel checks this value when a process is aborted, before creating a core file in the current directory of the process (see the section "Actions Performed upon Delivering a Signal" in Chapter 11). If the limit is 0, the kernel won't create the file.
RLIMIT_CPU	The maximum CPU time for the process, in seconds. If the process exceeds the limit, the kernel sends it a SIGXCPU signal, and then, if the process doesn't terminate, a SIGKILL signal (see Chapter 11).
RLIMIT_DATA	The maximum heap size, in bytes. The kernel checks this value before expanding the heap of the process (see the section "Managing the Heap" in Chapter 9).
RLIMIT_FSIZE	The maximum file size allowed, in bytes. If the process tries to enlarge a file to a size greater than this value, the kernel sends it a SIGXFSZ signal.
RLIMIT_LOCKS	Maximum number of file locks (currently, not enforced).
RLIMIT_MEMLOCK	The maximum size of nonswappable memory, in bytes. The kernel checks this value when the process tries to lock a page frame in memory using the mlock( ) or mlockall( ) system calls (see the section "Allocating a Linear Address Interval" in Chapter 9).
RLIMIT_MSGQUEUE	Maximum number of bytes in POSIX message queues (see the section "POSIX Message Queues" in Chapter 19).
RLIMIT_NOFILE	The maximum number of open file descriptors. The kernel checks this value when opening a new file or duplicating a file descriptor (see Chapter 12).
RLIMIT_NPROC	The maximum number of processes that the user can own (see the section "The clone( ), fork( ), and vfork( ) System Calls" later in this chapter).
RLIMIT_RSS	The maximum number of page frames owned by the process (currently, not enforced).
RLIMIT_SIGPENDING	The maximum number of pending signals for the process (see Chapter 11).
RLIMIT_STACK	The maximum stack size, in bytes. The kernel checks this value before expanding the User Mode stack of the process (see the section "Page Fault Exception Handler" in Chapter 9).

Process Switch

Process Switch (Context Switch)

- Suspend the execution of the process running on the CPU
- Resume the execution of some other process previously suspended

- Older Linux versions hardware-based context
- Since 2.6, Linux uses software context switch
  - Can do more security checks
  - Comparable performance

Context Switch

- Hardware Context
  - current->thread: CPU registers
  - current->stack (kernel mode stack in thread_info) : other hardware context
- Context Switch
  - Only on schedule()
  - Switch Page Global Directory to a new address space
  - Switch the Kernel Mode stack and Switch the hardware context
    - By the switch_to Marco

TSS (Task State Segment)

- In the original Intel design, each process has a TSS to store hardware contexts
- Linux uses one single TSS for each CPU, assessable via GDT
- The single TSS stores the following per-process information
  - Kernel space stack address
  - IO permission bitmap
    - User-mode accessible IO ports

The switch process

- context_switch(rq, prev, next)
- switch_to(prev, next, prev)
  - Save flags
  - Save ebp
  - Save esp to prev->thread.esp
  - Restore esp from next->thread.esp
  - Save l: to prev->thread.eip
  - Push next->thread.eip
  - Jmp __swtch_to
  - l: Restore ebp
  - Restore flags
  - Save prev to last

Several local variables

- prev: process descriptor of the process to be switched out
- next: process to be switched in
- Process switch
    - Saving the hardware context of prev
    - Replacing it with the hardware context of next

Task State Segment

- Task State Segment (TSS) to store hardware contexts in x86
- When an 80x86 CPU switches from User Mode to Kernel Mode, it fetches the address of the Kernel Mode stack from the TSS
- When a User Mode process attempts to access an I/O port by means of an in or out instruction, the CPU may need to access an I/O Permission Bitmap stored in the TSS

```c
struct tss_struct {
	/*
	 * The hardware state:
	 */
	struct x86_hw_tss	x86_tss;

	/*
	 * The extra 1 is there because the CPU will access an
	 * additional byte beyond the end of the IO permission
	 * bitmap. The extra byte must be all 1 bits, and must
	 * be within the limit.
	 */
	unsigned long		io_bitmap[IO_BITMAP_LONGS + 1];

	/*
	 * .. and then another 0x100 bytes for the emergency kernel stack:
	 */
	unsigned long		stack[64];

} ____cacheline_aligned;
```

The thread field

- The hardware context is saved in the thread_struct then the process switched out
- In current->thread

```c
struct thread_struct {
	/* Cached TLS descriptors: */
	struct desc_struct	tls_array[GDT_ENTRY_TLS_ENTRIES];
	unsigned long		sp0;
	unsigned long		sp;
#ifdef CONFIG_X86_32
	unsigned long		sysenter_cs;
#else
	unsigned long		usersp; /* Copy from PDA */
	unsigned short		es;
	unsigned short		ds;
	unsigned short		fsindex;
	unsigned short		gsindex;
#endif
#ifdef CONFIG_X86_32
	unsigned long		ip;
#endif
#ifdef CONFIG_X86_64
	unsigned long		fs;
#endif
	unsigned long		gs;
	/* Hardware debugging registers: */
```

Performing the process switch

- Every process switch consists of two steps:
  - Switching the Page Global Directory to install a new address space.
  - Switching the Kernel Mode stack and the hardware context.

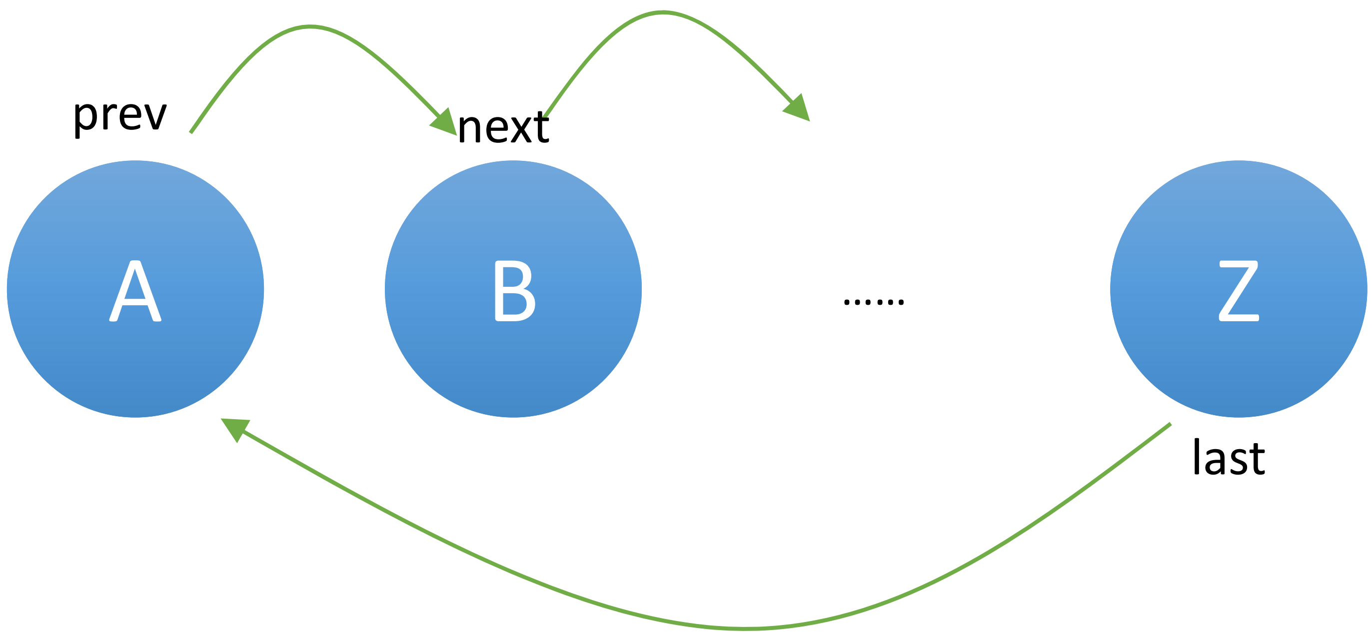
The switch_to macro

- The second step of the process switch is performed by the switch_to macro
  - The macro has three parameters, called prev, next, and last

- prev
  - The running process that will be suspended
- next
  - The ready process that will use the CPU
- Last
  - When prev resumes, the process that transferred the CPU to prev

The switch_to macro

- last: will be used to account the CPU time usage



```
/*
 * Saving eflags is important. It switches not only IOPL between tasks,
 * it also protects other tasks from NT leaking through sysenter etc.
 */
#define switch_to(prev, next, last)                                    \
do {                                                                   \
        /*                                                             \
         * Context-switching clobbers all registers, so we clobber     \
         * them explicitly, via unused output variables.               \
         * (EAX and EBP is not listed because EBP is saved/restored    \
         * explicitly for wchan access and EAX is the return value of  \
         * __switch_to())                                              \
         */                                                            \
        unsigned long ebx, ecx, edx, esi, edi;                         \
                                                                       \
        asm volatile("pushfl\n\t"               /* save    flags */    \
                     "pushl %%ebp\n\t"          /* save    EBP   */    \
                     "movl %%esp,%[prev_sp]\n\t"        /* save    ESP   */ \
                     "movl %[next_sp],%%esp\n\t"        /* restore ESP   */ \
                     "movl $1f,%[prev_ip]\n\t"  /* save    EIP   */    \
                     "pushl %[next_ip]\n\t"     /* restore EIP   */    \
                     __switch_canary                                   \
                     "jmp __switch_to\n"        /* regparm call  */    \
                     "1:\t"                                            \
                     "popl %%ebp\n\t"           /* restore EBP   */    \
                     "popfl\n"                  /* restore flags */    \
                                                                       \
                     /* output parameters */                           \
                     : [prev_sp] "=m" (prev->thread.sp),               \
                       [prev_ip] "=m" (prev->thread.ip),               \
                       "=a" (last),                                    \
                                                                       \
                       /* clobbered output registers: */               \
                       "=b" (ebx), "=c" (ecx), "=d" (edx),             \
                       "=S" (esi), "=D" (edi)                          \
                                                                       \
                       __switch_canary_oparam                          \
                                                                       \
                       /* input parameters: */                         \
                     : [next_sp]  "m" (next->thread.sp),               \
                       [next_ip]  "m" (next->thread.ip),               \
                                                                       \
                       /* regparm parameters for __switch_to(): */     \
                       [prev]     "a" (prev),                          \
                       [next]     "d" (next)                           \
                                                                       \
                       __switch_canary_iparam                          \
                                                                       \
                     : /* reloaded segment registers */                \
                        "memory");                                     \
} while (0)
```

The switch_to macro

- Saves prev and next into the eax and edx registers

```
movl prev, %eax
movl next, %edx
```

The switch_to macro

- Saves eflags and ebp registers in the prev Kernel Mode stack

```
pushfl
pushl %ebp
```

The switch_to macro (save esp to prev->thread.esp)

- Saves esp in prev->thread.esp
  - the field points to the top of the prev Kernel Mode stack:

```
movl %esp, 484(%eax)
```

- The 484(%eax) operand identifies the memory cell whose address is the contents of eax plus 484
  - 484(%eax) is prev->thread.esp

The switch_to macro

- Loads next->thread.esp in esp.
  - the kernel operates on the Kernel Mode stack of next.
- the address of a process descriptor is closely related to that of the Kernel Mode stack
  - changing the kernel stack means changing the current process:

```
movl 484(%edx), %esp
```

The switch_to macro

- Saves the address labeled 1 in prev->thread.eip
- When the process being replaced resumes its execution, the process executes the instruction labeled as 1:

```
movl $1f, 480(%eax)
```

  - 480(%eax) is prev->thread.eip

The switch_to macro

- On the Kernel Mode stack of next, the macro pushes the next->thread.eip value, which, in most cases, is the address labeled as 1:

```
pushl 480(%edx)
```

- 480(%edx) is next->thread.eip

The switch_to macro

- Jumps to the _ _switch_to( ) C function (see next):

```
jmp _ _switch_to
```

- The CPU will be transferred from prev to next

The switch_to macro

- Now the CPU executes the instruction at “1”, i.e., process prev has been resumed
  - esp points to the kernel mode stack of prev
- restore the contents of the eflags and ebp registers.

```
1:
popl %ebp
Popfl
```

The switch_to macro

- Copies the content of the eax register (loaded in step 1 above) into the memory location identified by the third parameter last of the switch_to macro:

  - **`movl %eax, last`**

__switch_()

- The _ _switch_to( ) function does the bulk of the process switch started by the switch_to( ) macro.

Lazy Restoration of FPU registers

- FPU/MMX/SSE/SSE2 instructions are rarely used in the kernel code, but there are a large number of related registers
  - To speed up process switch, FPU registers are saved when a process suspends, but they are *not* restored when a process resumes
  - Explicitly warp FPU-related code with kernel_fpu_begin() and kernel_fpu_end() so that the kernel will restore FPU registers

Process Creation/Deletion

Creating Processes

- The semantic of the fork() system call causes performance issue
  - The child must be an exact copy of the parent
  - However, the child usually calls exec() right after fork()

- Three type of creating process:
  - Copy On Write (ordinary fork())
    - parent and child read the same physical pages.
    - either one tries to write on a physical page, copies its contents into a new physical page that is assigned to the writing process.
  - Lightweight processes
  - The vfork( ) system call

Creating Processes

- The semantic of the fork() system call causes performance issue
  - The child must be an exact copy of the parent
  - However, the child usually calls exec() right after fork()

- Some methods for fork() performance improvement
  - Copy On Write
  - Lightweight processes
  - The vfork( ) system call

Creating Processes

- Copy On Write (ordinary fork())
  - parent and child read the same physical pages.
  - either one tries to write on a physical page, copies its contents into a new physical page that is assigned to the writing process.

Creating Processes

- Lightweight processes
  - allow both the parent and the child to share many per-process kernel data structures, such as the paging tables, the open file tables, and the signal dispositions.

Creating Processes

- The vfork( ) system call
    - creates a process that shares the memory address space of its parent.
    - To prevent the parent from overwriting data needed by the child, the parent's execution is blocked until the child exits or executes a new program (by exec).

The clone,fork,vfork System Calls

- Lightweight processes are created in Linux by using a function named clone( )
- The fork( ) system call is implemented by Linux as a clone( ) system
  - With SIGCHLD signal and all the clone flags cleared
  - child_stack parameter is the current parent stack pointer.

The clone,fork,vfork System Calls

- The vfork( ) system call, is implemented by Linux as a clone( ) system call
  - flags parameter specifies both a SIGCHLD signal and the flags CLONE_VM and CLONE_VFORK,
  - child_stack parameter is equal to the current parent stack pointer.
- The children process is inserted before the parent in the scheduler queue
  - Why?

do_fork() and copy_process() functions

- The do_fork( ) function,handles the clone( ), fork( ), and vfork( ) system calls
  - Then calls copy_processes()

- The copy_process( ) function sets up the process descriptor and any other kernel data structure required for a child's execution.
  - Its parameters are the same as do_fork( ), plus the PID of the child

Kernel Threads

- Because some of the system processes run only in Kernel Mode, modern operating systems delegate their functions to kernel threads

- kernel threads differ from regular processes in the following ways:
  - Kernel threads run only in Kernel Mode, regular processes run *alternatively* in Kernel Mode and in User Mode.
  - Kernel threads use only linear addresses greater than PAGE_OFFSET (i.e., the 3^{rd} GB)
  - Regular processes use all four gigabytes of linear addresses, in either User Mode or Kernel Mode.

Creating a kernel thread

- The kernel_thread( ) receives as parameters the address of the kernel function to be executed (fn), the argument to be passed to that function (arg), and a set of clone flags (flags).
- The function invokes do_fork( ) as follows: do_fork(flags|CLONE_VM|CLONE_UNTRACED, 0, pregs, 0, NULL, NULL);

Common Linux Kernel Threads

- keventd
  - Executes the functions in the keventd_wq
- kapmd
  - Handles the events related to power management
- kswapd
  - Reclaims memory by swapping out pages
- pdflush
  - Submits dirty pages to the scheduler queue
- kblockd
  - One thread for each block device. Submits block requests from device queue to device driver
- kirqd
  - Runs tasklets

Process 0

- The ancestor of all processes, called process 0, or the *swapper**, is a kernel thread created from scratch during the initialization phase of Linux. It performs the following:
  - Create process 1
  - Executes HLT instruction for power saving when there is no process to run

For historical reasons. However, it is nothing to do with swap in Linux. Swap is handled by kswapd.

Process 1

- Process 1 is also called the *init* process. It performs the following:
  - Completes the initialization of the kernel.
  - Invokes the execve( ) system call to load the executable program init
  - init kernel thread becomes a regular process having its own per-process kernel data structure.
- Init is the parent of
  - Orphan processes
  - Daemonized kernel threads

Destroying Processes

Destroying Processes

- When **processes die**, the kernel must be notified so that it can release the resources owned by the process, including
  - Pages and page table
  - Semaphores
  - File system instances
  - Opened files
  - Namespace
  - IO permission bitmap (in TSS)

Process Termination

- In Linux 2.6 there are two system calls that terminate a User Mode application:
  - The exit_group( ) , which terminates a full thread group, that is, a whole multithreaded application.
    - The main kernel function that implements this system call is called do_group_exit( ).
  - The _exit( ) , which terminates a single process.
    - The main kernel function that implements this system call is called do_exit( ).

The do_group_exit function

- The function executes the following operations:
  1. Checks whether the SIGNAL_GROUP_EXIT flag of the exiting process is not zero, if true jump step 4
  2. Otherwise, it sets the SIGNAL_GROUP_EXIT flag of the process and stores the termination code in the current->signal->group_exit_codefield.
  3. Invokes the zap_other_threads( ) function, it sends a SIGKILL signal to other processes in the thread group of current
  4. Invokes the do exit() function passing to it the process termination code.

The do_exit() function

- The do_exit( ) function receives as a parameter the process termination code and essentially executes the following actions:
  1. Sets the PF_EXITING flag in the flag field of the process descriptor
  2. Detaches from the process descriptor the data structures related to paging, semaphores, filesystem, open file descriptors, namespaces, and I/O Permission Bitmap
  3. Removes, the process descriptor from a dynamic timer queue via the del_timer_sync( )
  4. Sets the exit_code field of the process descriptor to the process termination code.
  5. Invokes the exit_notify( )

Process Removal

- Unix kernels are not allowed to discard data included in a process descriptor field right after the process terminates.
- They are allowed to do so only after the parent process has issued a wait( )-like system call that refers to the terminated process.
  - That’s why the EXIT_ZOMBIE state has been introduced.

End of Unit 1