Linux Kernel Development
Week 2

工海四 b06501018 朱紹勳

	Linux Kernel Dev 3rd	Understand Linux Kernel 3rd	Extra
Bottom Half of interrupt context	Chapter 8	Chapter 4	
System Call	Chapter 5	Chapter 10	
Process Management	Chapter 3	Chapter 3	

Interrupt Context

Top Half & Bottom Half

- Like pipeline, to make handler deal with interrupt faster

- Top half :
  - Receive and response to interrupt
  - Reset hardware

- Bottom half :
  - Almost all the Interrupt handler part

The environment of bottom half

- First we have the “bottom half(BH)” mechanism
  - Static, simple , but , not allow concurrent, BOTTLENECK
- Task queue
  - Still not flexible enough
- Softirq
  - Static defined in compile time
- Tasklet
  - dynamic defined in runtime
    implement by Softirq
- Work queue

Table 8.1 **Bottom Half Status**

Bottom Half	Status
BH	Removed in 2.5
Task queues	Removed in 2.5
Softirq	Available since 2.3
Tasklet	Available since 2.3
Work queues	Available since 2.5

Softirq

- Rarely used, only in
- Defined in <linux/interrupt.h>
- Implement:
  - Handler
  - Executing
- Using
  - Assign index
  - Register handler

```
open_softirq(NET_TX_SOFTIRQ, net_tx_action);
open_softirq(NET_RX_SOFTIRQ, net_rx_action);
```

  - Raise softirq

```
raise_softirq(NET_TX_SOFTIRQ);
```

Table 8.2 **Softirq Types**

Tasklet	Priority
HI_SOFTIRQ	0
TIMER_SOFTIRQ	1
NET_TX_SOFTIRQ	2
NET_RX_SOFTIRQ	3
BLOCK_SOFTIRQ	4
TASKLET_SOFTIRQ	5
SCHED_SOFTIRQ	6
HRTIMER_SOFTIRQ	7
RCU_SOFTIRQ	8

Tasklets

```
struct tasklet_struct {
        struct tasklet_struct *next;  /* next tasklet in the list */
        unsigned long state;          /* state of the tasklet */
        atomic_t count;               /* reference counter */
        void (*func)(unsigned long);  /* tasklet handler function */
        unsigned long data;           /* argument to the tasklet function */
};
```

- used frequently, simpler interface & locking rules
- Usage
  - Declare

```
DECLARE_TASKLET(my_tasklet, my_tasklet_handler, dev);
tasklet_init(t, tasklet_handler, dev);
```

  - Write Tasklet Handler

```
void tasklet_handler(unsigned long data);
```

  - Schedule

```
tasklet_schedule(&my_tasklet);
tasklet_disable(&my_tasklet);
tasklet_enable(&my_tasklet);
```

  - ksoftirqd
    - A kernel thread to support a lot of tasklet & softirq : Scheduling
    - ksoftirqd use do_softirq() to deal with pending softirq call schedule() to enable more important process

```
for (;;) {
        if (!softirq_pending(cpu))
                schedule();

        set_current_state(TASK_RUNNING);

        while (softirq_pending(cpu)) {
                do_softirq();
                if (need_resched())
                        schedule();
        }

        set_current_state(TASK_INTERRUPTIBLE);
}
```

Work Queues

- Different mechanism, push the work to a kernel thread, context switch
- Implement : workqueue_struct, worker thread
- Usage
  - create task:
    - `DECLARE_WORK(name, void (*func)(void *), void *data);`
    - `INIT_WORK(struct work_struct *work, void (*func)(void *), void *data)`
  - work handling function : `void`
    - `work_handler(void *data) // exec by kernel thread, run at process context`
  - scheduling
    - `schedule_work(&work);`
    - `schedule_delayed_work(&work, delay);`
  - flushing
    - `void flush_scheduled_work(void);`
    - `int cancel_delayed_work(struct work_struct *work);`
  - create new workque
    - `struct workqueue_struct *create_workqueue(const char *name);`
    - `int queue_work(struct workqueue_struct *wq, struct work_struct *work)`
    - `int queue_delayed_work(struct workqueue_struct *wq, struct work_struct *work, unsigned long delay)`

Which Bottom Half Should I Use?

- Tasklet is a simpler form of Softirq
- Softirq leak support of serializations
- Workque use kernel thread.

Table 8.3 **Bottom Half Comparison**

Bottom Half	Context	Inherent Serialization
Softirq	Interrupt	None
Tasklet	Interrupt	Against the same tasklet
Work queues	Process	None (scheduled as process context)

  - 1. If the program fully support thread, CAN use Softirq.
  - 2. Prefer Tasklet to Softirq in general case.
  - 3. If the task need to run in context switch(sleep), choose Workque.

- Overhead: Workque > others (because of context switch)
- Easy to use: Workque > tasklet > Softirq

Locking & Disabling in Bottom Halves

- Further discussion in synchronization chapter
- Serialization between tasklets:
  - The same tasklet will not run concurrently, even on two different processors
  - Don’t have to worry about synchronization in tasklet
- Workque need lock mechanism (because of context switch & SMP)
- To control bottom-half process (Softirq & tasklet), call the follow func.

Table 8.4 **Bottom Half Control Methods**

Method	Description
`void local_bh_disable()`	Disables softirq and tasklet processing on the local processor
`void local_bh_enable()`	Enables softirq and tasklet processing on the local processor

defined in <asm/softirq.h>

- Workque works just like process context

System Call

System Call

- A layer between user process and hardware
  - Make programming easier (API) - POSIX
  - improve system security : Access Right, Encapsulation
  - Other ways to enter Kernel : exception, external interrupt

- API vs. Sys Call → A function definition / A request to kernel by **trap**

  trap **:** the software interrupt defined #interrupt 128 in X86

  Trigger by **int $0x80** instruction

- Return of Sys Call:
  - usually (-) means error
  - or defined in errno.h ex : *include/asm-i386/errno.h*

System Call
- Handler & Service Routine

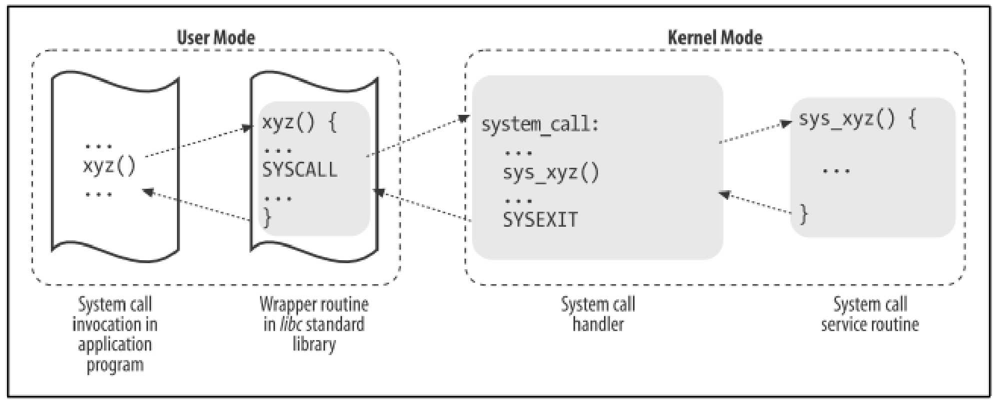


Figure 10-1. Invoking a system call

- Routine
  1. Save register to Kernel Mode Stack
  2. → C func() → System call handler(assembly) → System Call Service Routine
  3. Exit from handler, Stack → register, Switch to User Mode

- System Call Identification
  - We will pass system call number to help us, EAX register → Kernel
  - find #system call in dispatch table(store in an array), there are 289 in 2.6 ver
  - NR_syscalls macro 檢查 system call 是否合法
    - if so, call *sys_call_table(,%rax,8);

- Parameter of func() : ebx, ecx, edx, esi, edi register in X86_32
- Name of Service Routine : sys_xyz() ← xyz()

System Call
- Ways to invoke Sys Call

- There are two ways to invoke and exit Sys Call:

1.	invoke :	用 int $0x80	assembly	(old)
	exit :	用 iret	assembly	(old)
2.	invoke :	用 sysenter	assembly	
	exit :	用 sysexit	assembly	

Appendix ?

Passing & Verifying Parameters

- (1). Exceed 32bit (because of register) (2). 5 parameters
- More than 5 : A register will point to a memory area in the process address space that contains the parameter values.

- An useful & efficient examination process (return –EFAULT when error):
  - Verify the linear address is lower than PAGE_OFFSET

    (i.e., that it doesn’t fall within the range of interval addresses reserved to the kernel).
  - Checking until the last possible moment
  - `access_ok();` two parameters: addr, size, addr+size < (1) 2^32-1 (2)thread_info addr_limit.seg store
  - Check pointer legal access, Check if in user space…
    - `<linux/capability.h>`

Binding A System Call

- Register to an official system call:
  1. Add an entry to the end of the system call table (for every achitecture)
  2. Define system call number in <asm/unistd.h>
  3. Compile syscall to kernel image(not module)

     put system call into relevant file under kernel/

Summary : Create System Call

```
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_rt_tgsigqueueinfo     /* 335 */
       .long sys_perf_event_open
       .long sys_recvmmsg
```

- Assume a system call “foo”

```
/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall      0
#define __NR_exit                 1
#define __NR_fork                 2
#define __NR_read                 3
#define __NR_write                4
#define __NR_open                 5
```

1. Add `.long sys_foo` to the end of the entry.s file (the order indicate the syscall number)
2. Add syscall number (ex: `#define __NR_foo 338`) to <asm/unist.h>

```
#include <asm/page.h>

/*
 * sys_foo - everyone's favorite system call.
 *
 * Returns the size of the per-process kernel stack.
 */
asmlinkage long sys_foo(void)
{
        return THREAD_SIZE;
}
```

3. Put code under kernel/sys.c file

Access system call from user space

- Usually, the system call is provide from C lib
- Linux provide a macro, so that we can access without glibc

- General form of system call:

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

- Linux Macro form:

```
#define __NR_open 5 // The system call number of open is 5
_syscall3(long, open, const char *, filename, int, flags, int, mode)
// The parameters:
        1. The system call return type
        2. The system call name
        3. The type and name of system call parameters
```

Why Not to Implement a System Call

- Pro of creating new system call:
  - easy and efficient under linux
- Con of system call:
  - need extra syscall number
  - no modification
  - overkill

- Alternative Solutions:
  - Implement a device node and read() and write() to it. Use ioctl() to manipulate specific settings or retrieve specific information.
  - Use file descriptors to represent interfaces, such as semaphores, and manipulate.
  - Add the information as a file to the appropriate location in ***sysfs***.

Process Management

1 Intro of Process

2 Process Descriptor & Task Structure

3 Process Switch

4 Process Creation

5 Thread

6 Process Termination

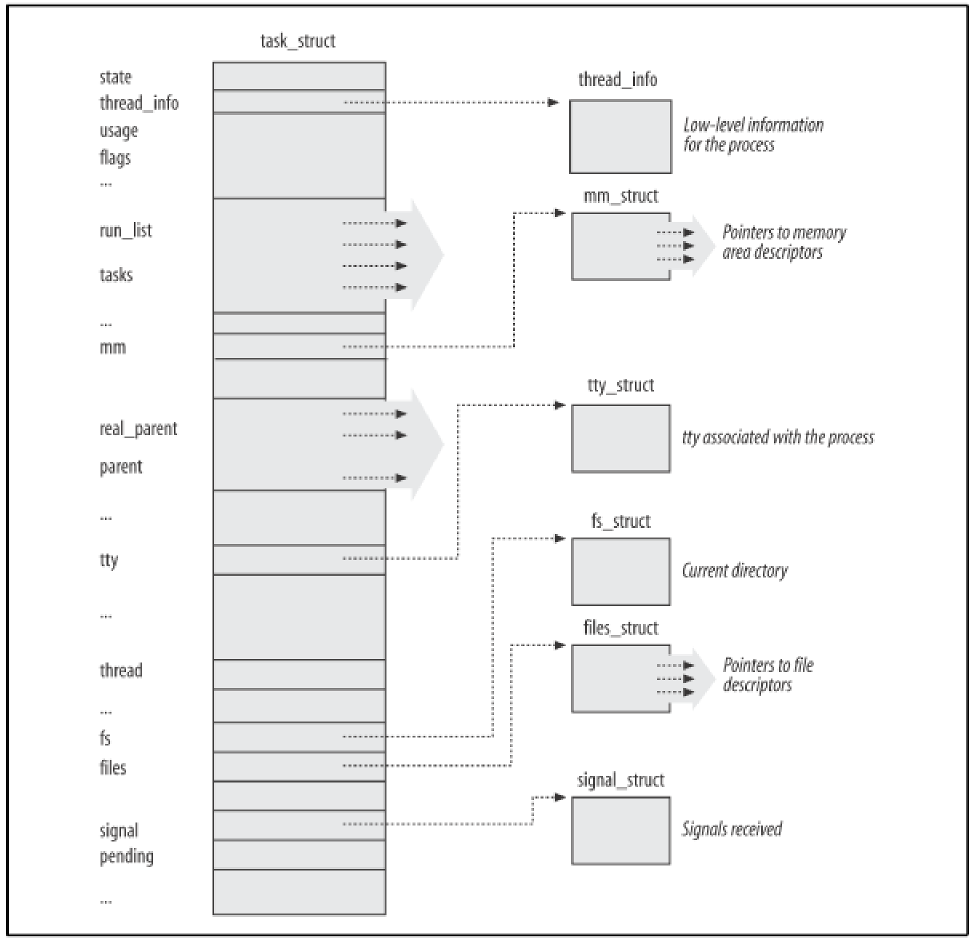
Intro of Process

- is a running program
- need other resource, ex : kernel data, state, MM, data section…

```
fork()      // clone()
exec()      //
wait()      //
waitpid()   //
```

Process Descriptor & Task Structure

- In type of struct task_struct
- Item of doubly linked list - called “task list”
- Defined in <linux/sched.h>
- task_struct is huge, 17kb in x86 arch



Allocation, Storage of Process Descriptor

- struct thread_info is defined on x86 in <asm/thread_info.h>
- struct task_struct stored at the end of the kernel stack of each process
  - to use less register, stack pointer is enough

- Use unique PID(pid_t : int) to identify processes
- max as 32,768 (or modify in /proc/sys/kernel/pid_max)
- In x86, offset of thread info is 8192 (13 LSB)

```
current_thread_info()->task;
getpid(); //  tgid vs pid
```

```
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;
};
```

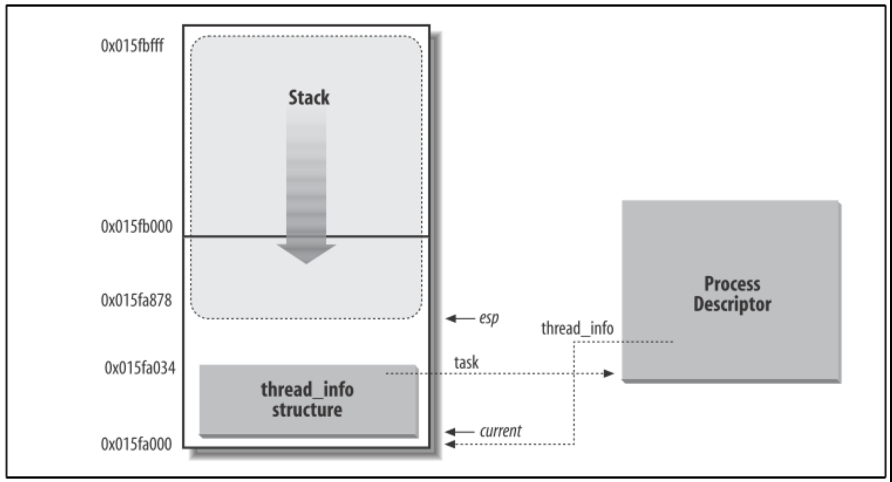


Figure 3-2. Storing the thread_info structure and the process kernel stack in two page frames

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

Process State

TASK_RUNNING:

Runnable, running, waiting for run

TASK_INTERRUPTIBLE:

The process is sleeping (blocked)

TASK_UNINTERRUPTIBLE:

it does not wake up and become runnable if it receives a signal

TASK_STOPPED:

This occurs if the task receives the
SIGSTOP , SIGTSTP , SIGTTIN , or SIGTTOU signal

TASK_TRACED:

In debugger, `ptrace()` by another process

--

EXIT_ZOMBIE:

process terminate before parent call waitpid()

EXIT_DEAD:

The final state before delete, after parent issued waitpid()

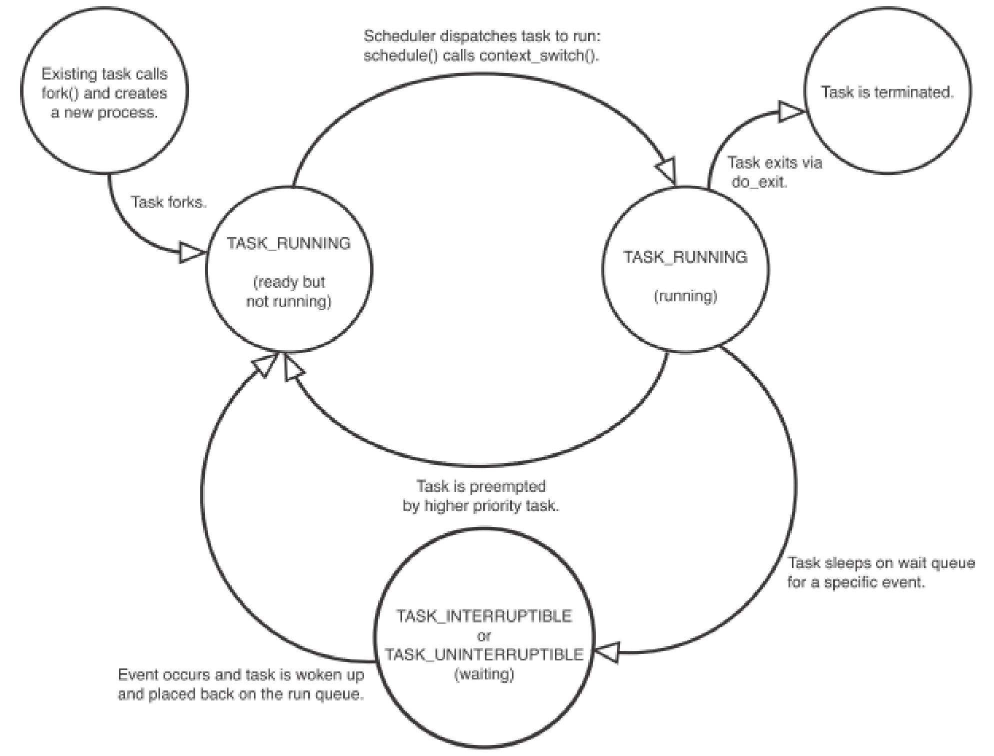


Figure 3.3 Flow chart of process states.

```
    p->state = task_running;
kernel:
    set_task_state, set_current_state
```

Process Relationship

- PID 0 is called *swapper, use for scheduling*
- All process are the child of *init* (PID = 1)

Field name	Description
group_leader	Process descriptor pointer of the group leader of P
signal->pgrp	PID of the group leader of P
tgid	PID of the thread group leader of P
signal->session	PID of the login session leader of P
ptrace_children	The head of a list containing all children of P being traced by a debugger
ptrace_list	The pointers to the next and previous elements in the real parent's list of traced processes (used when P is being traced)

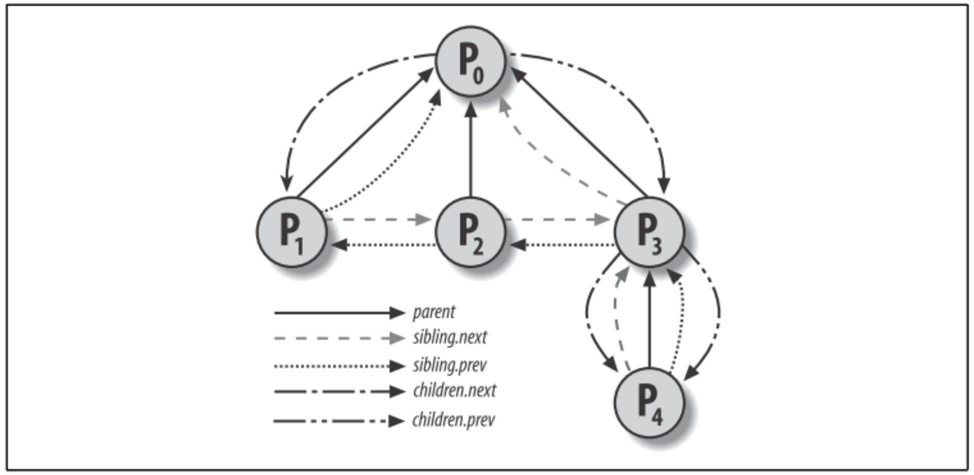


Figure 3-4. Parenthood relationships among five processes

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. (Therefore, when a user starts a background process and exits the shell, the background process becomes the child of *init*.)
parent	Points to the current parent of P (this is the process that must be signaled when the child process terminates); its value usually coincides with that of `real_parent`. It may occasionally differ, such as when another process issues a `ptrace( )` system call requesting that it be allowed to monitor P (see the section "Execution Tracing" in Chapter 20).
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.

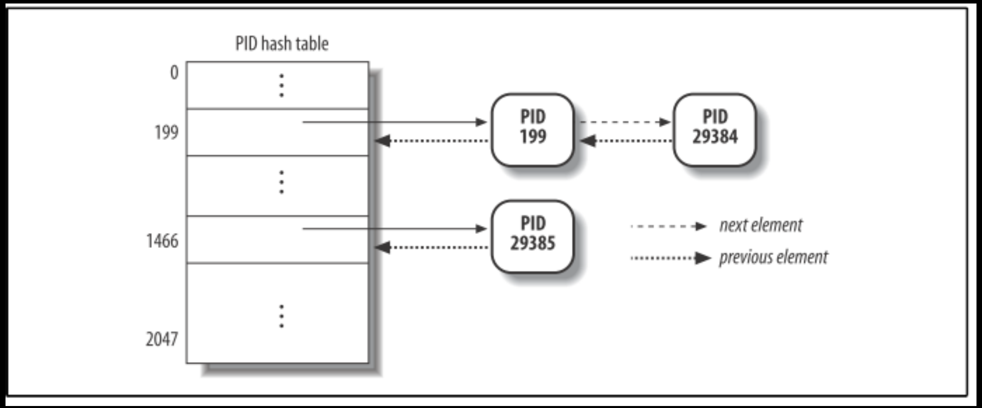
Hash table type	Field name	Description
PIDTYPE_PID	pid	PID of the process
PIDTYPE_TGID	tgid	PID of thread group leader process
PIDTYPE_PGID	pgrp	PID of the group leader process
PIDTYPE_SID	session	PID of the session leader process

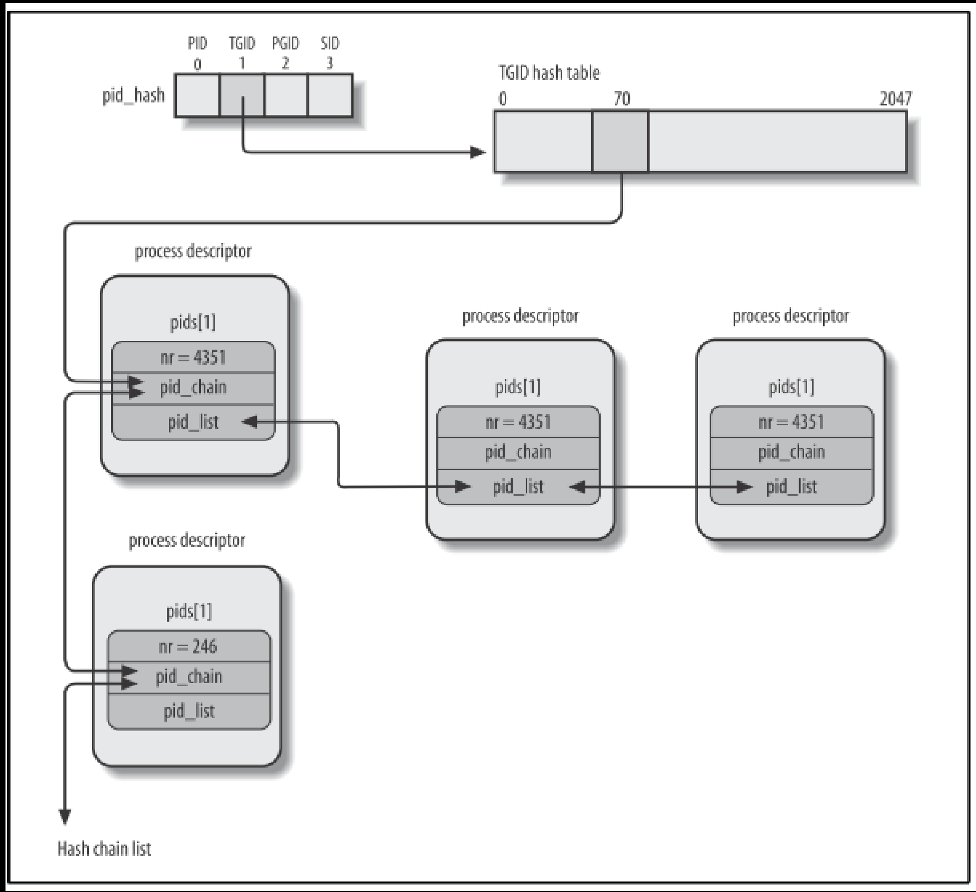
PIDhash

- Hash table with 2048 indexes + chaining
- struct of pid:

Type	Name	Description
int	nr	The PID number
struct hlist_node	pid_chain	The links to the next and previous elements in the hash chain list
struct list_head	pid_list	The head of the per-PID list

```
do_each_task_pid(nr, type, task)
while_each_task_pid(nr, type, task)
find_task_by_pid_type(type, nr)
find_task_by_pid(nr)
attach_pid(task, type, nr)
detach_pid(task, type)
next_thread(task)
```





Process Creation

- `fork() & exec()`
- fork(copy) a current process
  - only difference is pid & some resource
- Copy-on-Write
  - Share same process address space
  - When there is modify request, duplicate the parent content
  - very important optimization
- Loads a new into the address space then execute

Process Resource Limitation

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).

fork() vfork()

- do_fork()   // defined in kernel/fork.c
  - copy_process()
    - dup_task_struct()   // create a kernel stack for new process
    - Check not exceed the resource limits
    - Reset some of the item in process descriptor
    - child’s state is set to TASK_UNINTERRUPTIBLE
    - Calls copy_flags() to update the flags member of the task_struct.
    - alloc_pid
    - return