Precesses

Wang Xiaolin

June 13, 2013

✉ wx672ster+os@gmail.com

Processes

A process is

- an instance of a program in execution
- a dynamic entity (has lifetime)
- a collection of data structures describing the execution progress
- the unit of system resources allocation

The Linux kernel internally refers to processes as *tasks*.

When A Process Is created

The child

- is almost identical to the parent
  - has a logical copy of the parent's address space
  - executes the same code
- has its own data (stack and heap)

Multithreaded Applications

Threads

- are execution flows of a process
- share a large portion of the application data structures

Lightweight processes (LWP) — Linux way of multithreaded applications

- each LWP is scheduled individually by the kernel
  - no nonblocking syscall is needed
- LWPs may share some resources, like the address space, the open files, and so on.

Process Descriptor

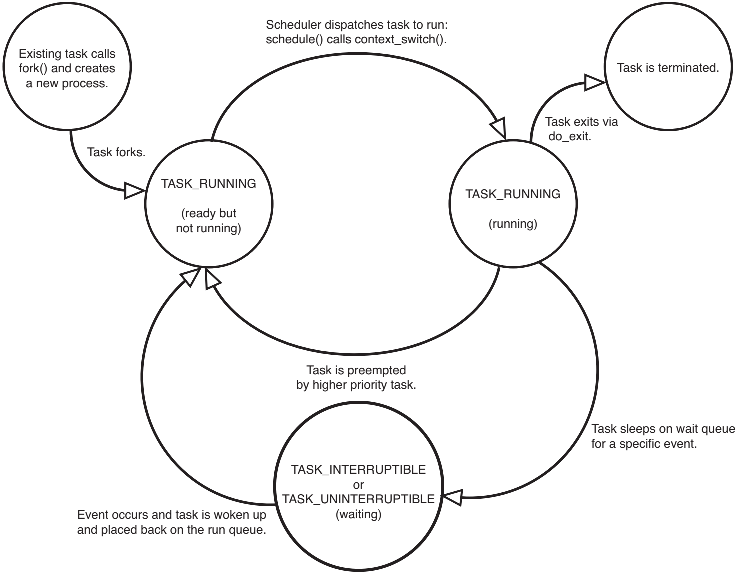
To manage processes, the kernel must have a clear picture of what each process is doing.

- the process's priority
- running or blocked
- its address space
- files it opened
- ...

Process descriptor: a task_struct type structure containing all the information related to a single process.

```
struct task_struct {
  /* 160 lines of code in 2.6.11 */
};
```

Process State



PID AND TGID

- kernel finds a process by its *process descriptor pointer* pointing to a task_struct
- users find a process by its PID
- all the threads of a multithreaded application share the same identifier

  tgid: the PID of the thread group leader

```
struct task_struct {
  ...
  pid_t pid;
  pid_t tgid;
  ...
};
```

~$ ps -eo pgid,ppid,pid,tgid,tid,nlwp,comm –sort pid

How many PIDs can there be?

- #define PID_MAX_DEFAULT 0x8000
- Max PID number = PID_MAX_DEFAULT - 1 = 32767
- $ cat /proc/sys/kernel/pid_max

Which are the free PIDs?

```
static pidmap_t pidmap_array[PIDMAP_ENTRIES] =
  {
    [ 0 ... PIDMAP_ENTRIES-1 ] =
    { ATOMIC_INIT(BITS_PER_PAGE), NULL }
  };
```

pidmap_array consumes a single page.

Process Descriptor Handling

thread_union: 2 consecutive page frames (8K) containing

- a process kernel stack
- a thread_info structure

```
union thread_union {
  struct thread_info thread_info;
  unsigned long stack[2048]; /* 1024 for 4KB stacks */
};
```

Kernel Stack

```
0xC15FBFFF +-------------+
           |    stack    |
           |             |
           |      |      |
           |      |      |
           |      |      |
           |      |      |                    +-------------+
0xC15FB000 +-     |     -+        thread_info |             |
           |      v      |               .----|             |
0xC15FA878 |    -----    | <- esp        |    | task_struct |
           |             |               |    |             |
0xC15FA034 |-------------|  task         |    |             |
           | thread_info |------------------> +-------------+
           |  structure  | <-- current   |
0xC15FA000 +-------------+ <-------------'
```

```c
struct thread_info {
        struct task_struct      *task;          /* main task structure */
        struct exec_domain      *exec_domain;   /* execution domain */
        unsigned long           flags;          /* low level flags */
        unsigned long           status;         /* thread-synchronous flags */
        __u32                   cpu;            /* current CPU */
        __s32                   preempt_count; /* 0 => preemptable, <0 => BUG */


        mm_segment_t            addr_limit;     /* thread address space:
                                                   0-0xBFFFFFFF for user-thead
                                                   0-0xFFFFFFFF for kernel-thread
                                                */
        struct restart_block    restart_block;

        unsigned long           previous_esp;   /* ESP of the previous stack in case
                                                   of nested (IRQ) stacks
                                                */
        __u8                    supervisor_stack[0];
};
```

Why both task_struct and thread_info?

- There wasn't a thread_info in pre-2.6 kernel
- Size matters

thread_info and task_struct are mutually linked

```c
struct thread_info {
  struct task_struct *task; /* main task structure */
  ...
};

struct task_struct {
  ...
  struct thread_info *thread_info;
  ...
};
```

Identifing The Current Process

Efficiency benefit from thread_union

- Easy get the base address of thread_info from esp register by masking out the 13 least significant bits of esp

current_thread_info()

```
/* how to get the thread information struct from C */
static inline struct thread_info *current_thread_info(void)
{
  struct thread_info *ti;
  __asm__("andl␣%%esp,␣%0;" :"=r" (ti) :"0" (~(THREAD_SIZE - 1)));
  return ti;
}
```

Can be seen as:

```
movl $0xffffe000,%ecx /* or 0xfffff000 for 4KB stacks */
andl %esp,%ecx
movl %ecx,p
```

To get the process descriptor pointer

current_thread_info()->task

```
movl $0xffffe000,%ecx /* or 0xfffff000 for 4KB stacks */
andl %esp,%ecx
movl (%ecx),p
```

Because the task field is at offset 0 in thread_info, after executing these 3 instructions p contains the process descriptor pointer.

current — a marco pointing to the current running task

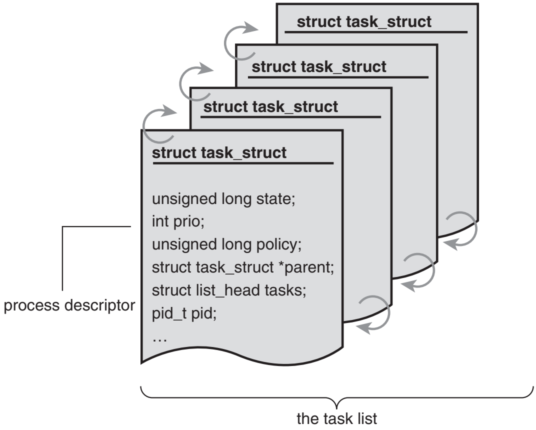
```
static inline struct task_struct * get_current(void)
{
        return current_thread_info()->task;
}

#define current get_current()
```

Task List

The kernel stores the list of processes in a circular doubly linked list called the task list.

Swapper The head of this list, init_task, process 0.



Doubly Linked List

```
     .-----------------------------------------------------------------.
     |                                                                 |
     v                                                                 |
 list_head           data                                              |
+---------+<--.   structure 1           data                           |
|  next --|---|-->+=========+<---.   structure 2           data        |
+---------+   |   |  next --|----|-->+=========+<---.   structure 3    |
|  prev   |   |   +---------+    |   |  next --|----|-->+=========+<---|--.
+----o----+   `---|--prev   |    |   +---------+    |   |  next --|----'  |
     |            +=========+    `---|--prev   |    |   +---------+       |
     |            |         |        +=========+    `---|--prev   |       |
     |            +---------+        |         |        +=========+       |
     |                               +---------+        |         |       |
     |                                                  +---------+       |
     |                                                                    |
     `--------------------------------------------------------------------'
```

```
struct list_head {
        struct list_head *next, *prev;
};
```

```
struct task_struct {
  ...
  struct list_head tasks;
  ...
}
```

List operations

SET_LINKS insert into the list

REMOVE_LINKS remove from the list

for_each_process scan the whole process list

```
#define for_each_process(p) \
        for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```

list_for_each iterate over a list

```
#define list_for_each(pos, head)                                  \
  for (pos = (head)->next; prefetch(pos->next), pos != (head); \
        pos = pos->next)
```

Example: Iterate over a process' children

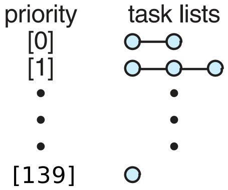
```
struct task_struct *task;
struct list_head *list;
list_for_each(list, &current->children) {
  task = list_entry(list, struct task_struct, sibling);
  /* task now points to one of current's children */
}
```

A task can be in multiple lists

```
struct task_struct {
  struct list_head run_list;
  struct list_head tasks;
  struct list_head ptrace_children;
  struct list_head ptrace_list;
  struct list_head children; /* list of my children */
  struct list_head sibling;  /* linkage in my parent's children list */
}
```

The List Of TASK_RUNNING Processes

- Each CPU has its own runqueue
- Each runqueue has 140 lists
- One list per process priority
- Each list has zero to many tasks




```c
struct task_struct {
  ...
  int prio, static_prio;
  struct list_head run_list;
  prio_array_t *array;
  ...
};
```

Each Runqueue Has A prio_array_t Struct

```
typedef struct prio_array prio_array_t;

struct prio_array {
        unsigned int nr_active;
        unsigned long bitmap[BITMAP_SIZE];
        struct list_head queue[MAX_PRIO];
};
```

nr_active: The number of process descriptors linked into the lists (the whole runqueue)

bitmap: A priority bitmap. Each flag is set if the priority list is not empty

queue: The 140 heads of the priority lists

To Insert A Task Into A Runqueue List

```
static void enqueue_task(struct task_struct *p, prio_array_t *array)
{
  ...
  list_add_tail(&p->run_list, &array->queue[p->prio]);
  __set_bit(p->prio, array->bitmap);
  array->nr_active++;
  p->array = array;
}
```

prio: priority of this process

array: a pointer pointing to the prio_array_t of this runqueue

- To removes a process descriptor from a runqueue list, use dequeue_task(p,array) function.

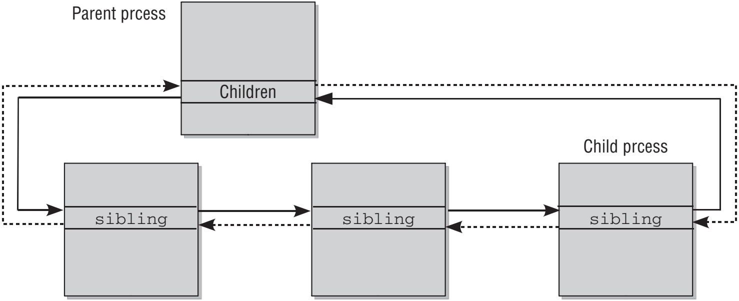
Relationships Among Processes

Family relationship

```
struct task_struct {
  ...
  struct list_head children; /* list of my children */
  struct list_head sibling;  /* linkage in my parent's children list */
  ...
};
```

children: is the list head for the list of all child elements of the process

sibling: is used to link siblings with each other



Other Relationships

A process can be:

- a leader of a process group or of a login session
- a leader of a thread group
- tracing the execution of other processes

```
struct task_struct {
  ...
  pid_t tgid;
  ...
  struct task_struct *group_leader; /* threadgroup leader */
  ...
  struct list_head ptrace_children;
  struct list_head ptrace_list;
  ...
};
```

The Pid Hash Table And Chained Lists

PID ⇒ process descriptor pointer?

- Scanning the process list? — too slow
- Use hash tables

Four hash tables have been introduced

Why 4? For 4 types of PID

$$\left.\begin{array}{r}\text{PID}\\\text{TGID}\\\text{PGID}\\\text{SID}\end{array}\right\} \Rightarrow \text{task\_struct}$$

Collision

Multiple PIDs can be hashed into one table index

```
  +---+              Hash table
  |PID|             +----------+
  +---+           0 |          |
    |               +----------+       process
    v               |          |     descriptors
+------------+      +----------+   +---+     +---+
|pid_hashfn()|-->m |      <---|-->|885|<--->|169|
+------------+      +----------+   +---+     +---+
                    |          |
                    +----------+   +-----+
                    |      <---|-->|29385|
                    +----------+   +-----+
               2047 |          |
                    +----------+
```

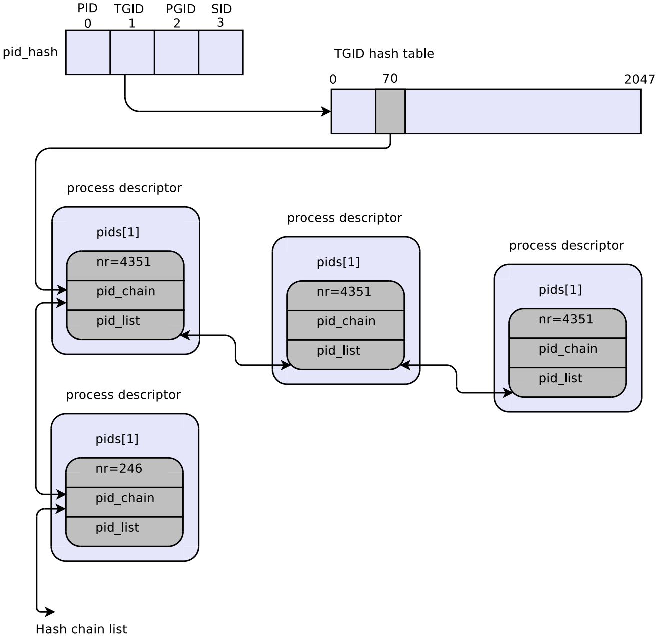
- Chaining is used to handle colliding PIDs
- No collision if the table is 32768 in size! But...

The pid data structure

```
struct pid
{
  int nr;
  struct hlist_node pid_chain;
  struct list_head pid_list;
};
```

```
struct task_struct{
  ...
  struct pid pids[PIDTYPE_MAX];
  ...
}
```

PID Hash Tables



kernel/pid.c — Operations

- 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)

Wait Queues

- A wait queue represents a set of sleeping processes, which are woken up by the kernel when some condition becomes true.
- Wait queues are implemented as doubly linked lists whose elements include pointers to process descriptors.

Each wait queue is identified by a __wait_queue_head

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

lock: avoid concurrent accesses.

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;
```

- task: address of this sleeping process
- task_list: which wait queue are you in?
- flags: 1 – exclusive; 0 – nonexclusive;
- func: how it should be woken up?

Process Resource Limits

Limiting the resource use of a process

- The amount of system resources a process can use are stored in the current->signal->rlim field.
- rlim is an array of elements of type struct rlimit, one for each resource limit.

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

rlim_cur: the current resource limit for the resource

e.g. current->signal->rlim[RLIMIT_CPU].rlim_cur — the current limit on the CPU time of the running process.

rlim_max: the maximum allowed value for the resource limit

Resource Limits

RLIMIT_AS	The maximum size of process address space
RLIMIT_CORE	The maximum core dump file size
RLIMIT_CPU	The maximum CPU time for the process
RLIMIT_DATA	The maximum heap size
RLIMIT_FSIZE	The maximum file size allowed
RLIMIT_LOCKS	Maximum number of file locks
RLIMIT_MEMLOCK	The maximum size of nonswappable memory
RLIMIT_MSGQUEUE	Maximum number of bytes in POSIX message queues
RLIMIT_NOFILE	The maximum number of open file descriptors
RLIMIT_NPROC	The maximum number of processes of the user
RLIMIT_RSS	The maximum number of page frames owned by the process
RLIMIT_SIGPENDING	The maximum number of pending signals for the process
RLIMIT_STACK	The maximum stack size

Process Switch

Process execution context: all information needed for the process execution

Hardware context: the set of registers used by a process

Where is the hardware context stored?

- partly in the process descriptor (PCB)
- partly in the Kernel Mode stack

Process switch

- saving the hardware context of `prev`
- replacing it with the hardware context of `next`

Process switching occurs only in Kernel Mode.

Task State Segment (TSS)

- For storing hardware contexts
- One TSS for each process (Intel's design)
- Hardware context switching
  - far jmp to the TSS of next

Linux doesn't use hardware context switch

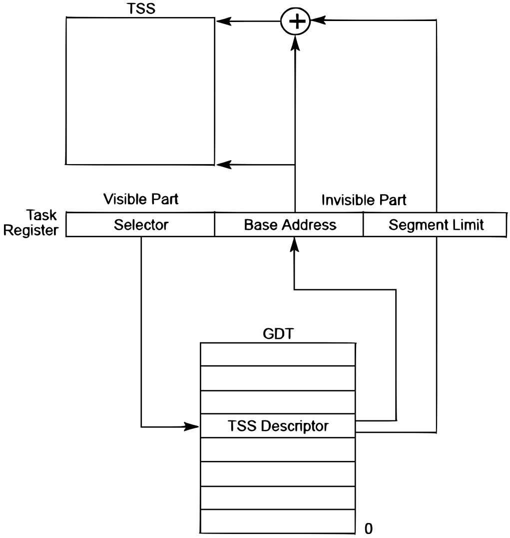
- One TSS for each CPU
  - The address of the kernel mode stack
  - I/O permission bitmap

Task State Segment Descriptor (TSSD)

```
  |31              24|23    20|19    16|15              8|7                 0|
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
7 |  BASE 31..24   |1|1|0|0| LIMIT |1|DPL|0 1 0 1 1|   BASE 23..16   | 4
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
3 |      SEGMENT BASE 15..0        |      SEGMENT LIMIT 15..0        | 0
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- S bit set to 0;
- Type bits set to 9/11;
- Busy bit set to 1.

The Task Register (tr)



Where to save the hardware context?

```
struct task_struct{
  ...
  struct thread_struct thread;
  ...
}
```

- thread_struct includes fields for most of the CPU registers, except the general-purpose registers such as eax, ebx, etc., which are stored in the Kernel Mode stack.

Performing The Process Switch

— schedule()

Two steps:

1. Switching the Page Global Directory
2. Switching the Kernel Mode stack and the hardware context

switch_to(prev,next,last)

- in any process switch three processes are involved, not just two
-

Creating Processes

The clone() system call

```
int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
          /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );
```

The traditional fork() system call

clone(func, child_stack, SIGCHLD, NULL);

- child_stack: parent stack pointer (copy-on-write)

vfork()

clone(func, child_stack, CLONE_VM|CLONE_VFORK|SIGCHLD, NULL);

- child_stack: parent stack pointer (copy-on-write)

The do_fork() function does the real work

```
 Program 1   Program 2   Program 3
+---------+ +---------+ +---------+
| fork()  | | vfork() | | clone() |
+---------+ +---------+ +---------+
     |           |           |
     V           V           V
+---------------------------------+
|            C Library            |
+----------------o----------------+
                 |                     User space
-----------------|----------------------------------
                 |                     Kernel space
                 |
                 |               +---------+
                 V               :   ...   :
          +-------------+      3 +---------+
          | System call |--o---->| fork()  |--> sys_fork() ---.
          +-------------+  |     +---------+                  |
                           |     :   ...   :                  |
                           | 120 +---------+                  |
                           |---->| clone() |--> sys_clone() --|
                           |     +---------+                  |
                           |     :   ...   :                  |
                           | 289 +---------+                  |
                           `---->| vfork() |--> sys_vfork() --|
                                 +---------+                  |
                                 :   ...   :                  V
                                 +---------+             do_fork()
                              System Call Table
```

do_fork() calls copy_process() to make a copy of process descriptor

```
long do_fork(unsigned long clone_flags,
             unsigned long stack_start,
             struct pt_regs *regs,
             unsigned long stack_size,
             int __user *parent_tidptr,
             int __user *child_tidptr)
{
  struct task_struct *p;
  ...
  long pid = alloc_pidmap();
  ...
  p = copy_process(clone_flags, stack_start, regs,
                   stack_size, parent_tidptr,
                   child_tidptr, pid);
  ...
  return pid;
}
```

copy_process()

1. dup_task_struct(): creates
   - a new kernel mode stack
   - thread_info
   - task_struct

   Values are identical to the parent
2. is current->signal->rlim[RLIMIT_NPROC].rlim_cur confirmed?
3. Update child's task_struct
4. Set child's state to TASK_UNINTERRUPTABLE
5. copy_flags(): update flags in task_struct
6. get_pid() (check pidmap_array bitmap)
7. Duplicate or share resources (opened files, FS info, signal, ...)
8. return p;

Creating A Kernel Thread

kernel_thread() is similar to clone()

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
{
  ...
  return do_fork(flags | CLONE_VM | CLONE_UNTRACED, 0, &regs, 0, NULL, NULL);
}
```

Process 0

Process 0 is a kernel thread created from scratch during the initialization phase.

- Also called *idle process*, or *swapper process*
- Its data structures are *statically* allocated

start_kernel()

- Initializes all the data structures
- Enables interrupts
- Creates another kernel thread — *process 1, the init process*

Call graph

```
start_kernel()
  `--> rest_init()
         |--> kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND)
         `--> cpu_idle()
```

- After having created the *init* process, *process 0* executes the cpu_idle() function.
- Process 0 is selected by the scheduler only when there are no other processes in the TASK_RUNNING state.
- In multiprocessor systems there is a process 0 for each CPU.

Process 1

- Created via
  `kernel_thread(init, NULL, CLONE_FS|CLONE_SIGHAND);`
- `PID` is 1
- shares all per-process kernel data structures with process 0
- starts executing the `init()` function
  - completes the initialization of the kernel
- `init()` invokes the `execve()` system call to load the executable program `init`
  - As a result, the *init kernel thread* becomes a regular process having its own per-process kernel data structure
- The init process stays alive until the system is shut down

Process Termination

- Usual way: call exit()
  - The C compiler places a call to exit() at the end of main().
- Unusual way: Ctrl-C ...

All process terminations are handled by do_exit()

- tsk->flags |= PF_EXITING; to indicate that the process is being eliminated
- del_timer_sync(&tsk->real_timer); to remove any kernel timers
- exit_mm(), exit_sem(), __exit_files(), __exit_fs(), exit_namespace(), exit_thread(): free pointers to the kernel data structures
- tsk->exit_code = code;
- exit_notify() to send signals to the task's parent
  - re-parents its children
  - sets the task's state to TASK_ZOMBIE
- schedule() to switch to a new process

Process Removal

Cleaning up after a process and removing its process descriptor are separate.

Clean up

- done in do_exit()
- leaves a zombie
  - To provide information to its parent
  - The only memory it occupies is its kernel stack, the thread_info structure, and the task_struct structure.

Removal

- release_task() is invoked by

either do_exit() if the parent didn't wait
or wait4()/waitpid()

- free_uid()
- unhash_process: to remove the process from the pidhash and from the task list
- put_task_struct()
  - free the pages containing the process's kernel stack and thread_info structure
  - de-allocate the slab cache containing the task_struct