Precesses

Wang Xiaolin
wx672ster@gmail.com

June 13, 2013

Contents

Textbook:

- Chapter 3, *Processes*, [BC05]
- Chapter 3, *Process Management*, [Lov10]
- Chapter 2, *Process Management and Scheduling*, [Mau08]

References

[BC05] D.P. Bovet and M. Cesatí. *Understanding The Linux Kernel*. 3rd ed. O'Reilly, 2005.

[Lov10] R. Love. *Linux Kernel Development*. Developer's Library. Addison-Wesley, 2010.

[Mau08] W. Mauerer. *Professional Linux Kernel Architecture*. John Wiley & Sons, 2008.

[RFS05] C.S. Rodriguez, G. Fischer, and S. Smolski. *The Linux kernel primer: a top-down approach for x86 and PowerPC architectures*. Prentice Hall Professional Technical Reference, 2005.

Todo list

1 Processes, Lightweight Processes, and Threads

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.

2 Process Descriptor

Process Descriptor

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

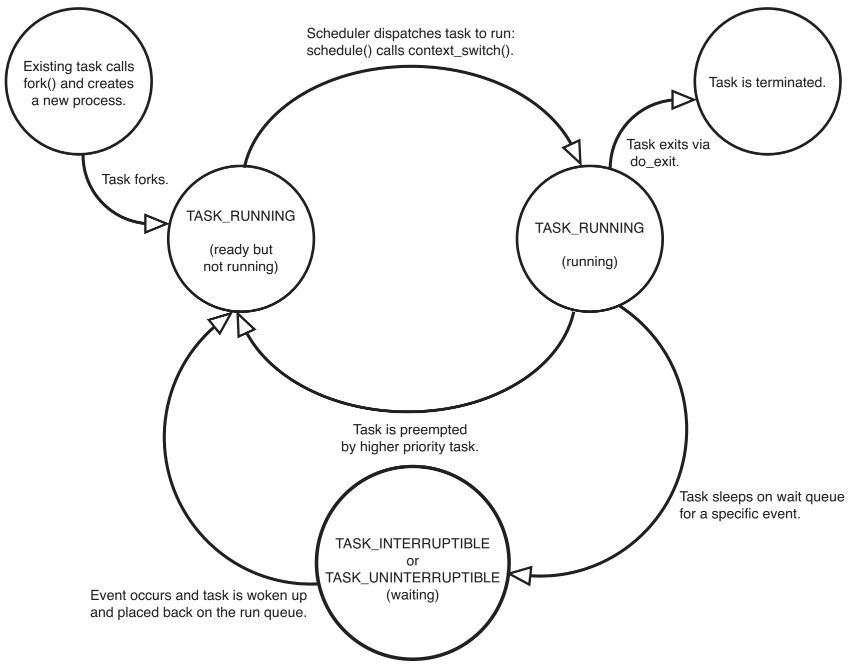
- the process's priority
- running or blocked
- its address space
- files it opened
- ...

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 */
};
```

2.1 Process State

Process State


Existing task calls fork() and creates a new process.
Scheduler dispatches task to run: schedule() calls context_switch().
Task is terminated.
Task forks.
Task exits via do_exit.
TASK_RUNNING (ready but not running)
TASK_RUNNING (running)
Task is preempted by higher priority task.
Task sleeps on wait queue for a specific event.
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE (waiting)
Event occurs and task is woken up and placed back on the run queue.

(sec 3.2.1 in [BC05]) The state field of task_struct is an array of flags, each of which describes a possible process state. In the current Linux version, these states are mutually exclusive, and hence exactly one flag of state always is set; the remaining flags are cleared. The following are the possible process states:

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. Raising a hardware interrupt, releasing a system resource the process is waiting for, or delivering a signal are examples of conditions that might wake up the process (put its state back to TASK_RUNNING).

TASK_UNINTERRUPTIBLE Like TASK_INTERRUPTIBLE, except that delivering a signal to the sleeping process leaves its state unchanged. This process state is seldom used. It is valuable, however, under certain specific conditions in which a process must wait until a given event occurs without being interrupted. For instance, this state may be used when a process opens a device file and the corresponding device driver starts probing for a corresponding hardware device. The device driver must not be interrupted until the probing is complete, or the hardware device could be left in an unpredictable state.

TASK_STOPPED Process execution has been stopped; the process enters this state after receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal.

TASK_TRACED Process execution has been stopped by a debugger. When a process is being monitored by another (such as when a debugger executes a ptrace() system call to monitor a test program), each signal may put the process in the TASK_TRACED state.

EXIT_ZOMBIE Process execution is terminated, but the parent process has not yet issued a wait4() or waitpid() system call to return information about the dead process. Before the wait()-like call is issued, the kernel cannot discard the data contained in the dead process descriptor because the parent might need it.

EXIT_DEAD The final state: the process is being removed by the system because the parent process has just issued a wait4() or waitpid() system call for it. Changing its state from EXIT_ZOMBIE to EXIT_DEAD avoids race conditions due to other threads of execution that execute wait()-like calls on the same process.

2.2 Identifying a Process

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

TODO: Draw a process relationship graph

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 */
};
```

- The kernel uses the alloc_thread_info and free_thread_info macros to allocate and release the memory area storing a thread_info structure and a kernel stack.
- Benefits of storing thread_info and kernel mode stack together (Sec 3.2.2.2, *Identifying the current process*, [BC05]):
  - 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.
  - For multi-processor systems: 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. Earlier versions of Linux did not store the kernel stack and the process descriptor together. Instead, they were forced to introduce a global static variable called current to identify the process descriptor of the running process. On multi-processor systems, it was necessary to define current as an array one element for each available CPU.

Kernel Stack

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

More about kernel stack:

- stackoverflow: **Does there exist Kernel stack for each process?** In Linux, each task (userspace or kernel thread) has a kernel stack of either 8kb or 4kb, depending on kernel configuration. There are indeed separate stack pointers, however, only one is present in the CPU at any given time; if userspace code is running, the kernel stack pointer to be used on exceptions or interrupts is specified by the task-state segment, and if kernel code is running, the user stack pointer is saved in the context structure located on the kernel stack.

- stackoverflow: **kernel stack for Linux process?**
  - There is just one common kernel memory. In it each process has it's own task_struct + kernel stack (by default 8K).
  - In a context switch
    1. the old stack pointer value is stored in the task_struct of the old process;
    2. the stack pointer for the new process is read from the task_struct of this new process.
  - The kernel stack is a non-shared space where system calls can put their data. If you'd share them between processes, several kernel routines could use the same stack at the same time ⇒ data corruption.

- stackoverflow: **kernel stack vs user-mode application stack?** Conceptually, both are the same data structure: a stack. *security concern*

  The reason why there are two different stack per thread is because in user mode, code must not be allowed to mess up kernel memory. When switching to kernel mode, a different stack in memory only accessible in kernel mode is used for return addresses an so on.

  If the user mode had access to the kernel stack, it could modify a jump address (for instance), then do a system call; when the kernel jumps to the previously modified address, your code is executed in kernel mode!

  Also, security-related information/information about other processes (for synchronisation) might be on the kernel stack, so the user mode should not have read access to it either.

- Kernel stack vs. User stack

- Kernel Stack Initialization

```
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

---- ℹ ----

More about thread_info:

- (www.linuxjournal.com: The Linux Process Model) Since 2.2.x, the task_struct is allocated at the bottom of the kernel stack. We can overlap the task_struct on the kernel stack because the task_struct is a per-task structure exactly as the kernel stack.
- Why are task_struct and thread_info separate?

  Quote from `http://www.spinics.net/lists/newbies/msg22259.html`:

  Q: There is one task_struct and thread_info for each process, and they link to each other, right? So why are they separate structs?

  A: thread_info is embedded in kernel stack, so we can easily get the task_struct using kernel stack.

  Quote from `http://www.spinics.net/lists/newbies/msg22263.html`

  Q: That is a fine workaround for the structs being separate. But why not keep all the data in one struct on the kernel stack, like it was done in 2.4?

  A: task_struct is huge. it's around 1.7KB on a 32 bit machine. on the other hand, you can easily see that thread_info is much slimmer.

  size matters

  kernel stack is either 4 or 8KB, and either way a 1.7KB is pretty much, so storing a slimmer struct, that points to task_struct, immediately saves a lot of stack space and is a scalable solution.

- supervisor_stack[0] in struct thread_info is a *flexible array member*.
  - Sec 6.17, *Arrays of Length Zero*, GCC Manual
  - What is the advantage of using zero-length arrays in C?
  - MEMxx-C. Understand how flexible array members are to be used
  - (Kerneltrap.org: Regarding thread_info) In the case of tread_info: supervisor_stack seems to be the kernel stack, in this case an integer number of pages (i.e. 4096 or 8192 bytes) is allocated for the rest of struct thread_info + the stack, so

a) you don't need the real size because you allocate the fixed amount anyway, and
b) you would have to declare an array of (4096 - offsetof(struct thread_info, supervisor_stack)) bytes inside the struct itself and that is just not possible.

Or not declare the stack inside the struct but do offset calculations, but the zero sized array is more elegant and maintainable. There is nothing to fear about that, this is perfectly normal C.

- process descriptors (task_structs) are stored in dynamic memory (ZONE_HIGHMEM) rather than in the memory area permanently assigned to the kernel
- Kernel stack size, 8K vs. 4K
  - (Sec 3.2.2.1, *Process descriptors handling*, [BC05]) In Sec 2.3, *Segmentation in Linux*, [BC05], we learned that a process in Kernel Mode accesses a stack contained in the kernel data segment, which is different from the stack used by the process in User Mode. Because kernel control paths make little use of the stack, only a few thousand bytes of kernel stack are required. Therefore, 8 KB is ample space for the stack and the thread_info structure. However, when stack and thread_info structure are contained in a single page frame, the kernel uses a few additional stacks to avoid the overflows caused by deeply nested interrupts and exceptions (see Chapter 4, *Interrupts and Exceptions*, [BC05]).
  - (lwn.net: 4K stacks in 2.6) Each process on the system has its own kernel stack, which is used whenever the system goes into kernel mode while that process is running. Since each process requires a kernel stack, the creation of a new process requires an order 1 allocation. So the two-page kernel stacks can limit the creation of new processes, even though the system as a whole is not particularly short of resources. Shrinking kernel stacks to a single page eliminates this problem and makes it easy for Linux systems to handle far more processes at any given time.

    problem in finding 2 consecutive page frames
  - in linux-2.6.11/include/asm-i386/thread_info.h

```
#ifdef CONFIG_4KSTACKS
#define THREAD_SIZE (4096)
#else
#define THREAD_SIZE (8192)
#endif
```

---

thread_info and task_struct are mutually linked

```
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
```

(Stackoverflow: Understanding the getting of task_struct pointer from process kernel stack) Each process only gets 8192 bytes of kernel stack, aligned to a 8192-byte boundary, so whenever the stack pointer is altered by a push or a pop, the low 13 bits are the only part that changes. $2^{13} == 8192$.

$$\begin{aligned}\sim(\texttt{THREAD\_SIZE}-1) &=\sim(\texttt{8k}-1)\\ &=\sim(\texttt{0x00002000}-1)\\ &=\sim\texttt{0x00001fff}\\ &=\texttt{11111111111111111110000000000000b}\end{aligned}$$

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.

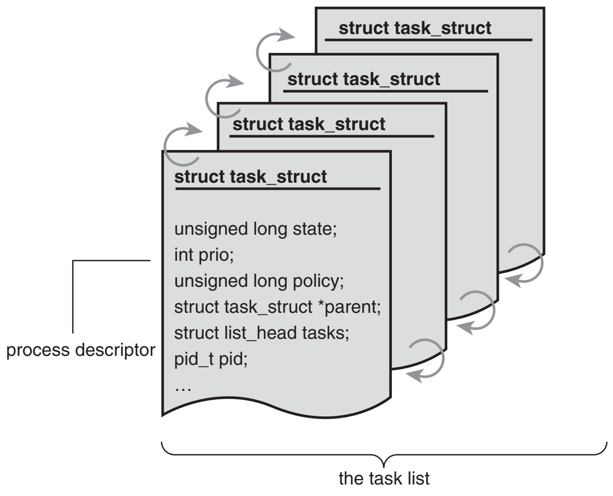
current — a marco pointing to the current running task

```
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.



Q: Stackoverflow: Why do we need a swapper task in linux? *swapper*

A: The reason is historical and programatic. The idle task is the task running, if no other task is runnable, like you said it. It has the lowest possible priority, so that's why it's running of no other task is runnable.

Programatic reason: This simplifies process scheduling a lot, because you don't have to care about the special case: "What happens if no task is runnable?", because there always is at least one task runnable, the idle task. Also you can count the amount of cpu time used per task. Without the idle task, which task gets the cpu-time accounted no one needs?

Historical reason: Before we had cpus which are able to step-down or go into power saving modes, it HAD to run on full speed at any time. It ran a series of NOP-instructions, if no tasks were runnable. Today the scheduling of the idle task usually steps down the cpu by using HLT-instructions (halt), so power is saved. So there is a functionality somehow in the idle task in our days.

In Windows you can see the idle task in the process list, it's the idle process.

Q: superuser.com: What is the main purpose of the swapper process in Unix?

A: It hasn't been a swapper process since the 1990s, and swapping hasn't really been used since the 1970s.

Unices stopped using swapping a long time ago. They've been demand-paged operating systems for a few decades — since System V R2V5 and 4.0BSD. The swapper process, as was, used to perform process swap operations. It used to swap entire processes — including all of the kernel-space data structures for the process — out to disc and swap them back in again. It would be woken up, by the kernel, on a regular basis, and would scan the process table to determine what swapped-out-and-ready-to-run processes could be swapped in and what swapped-in-but-asleep processes could be swapped out. Any textbook on Unix from the 1980s will go into this in more detail, including the swap algorithm. But it's largely irrelevant to demand-paged Unices, even though they retained the old swap mechanism for several years. (The BSDs tried quite hard to avoid swapping, in favour of paging, for example.)

Process #0 is the first process in the system, hand-crafted by the kernel. It fork()s process 1, the first user process. What it does other than that is dependent from what Unix the operating system actually is. As mentioned, the BSDs before FreeBSD 5.0 retained the old swapper mechanism, and process #0 simply dropped into the swapper code in the kernel, the scheduler() function, after it had finished system initialization. System V was much the same, except that process #0 was conventionally named sched rather than swapper. (The names are pretty much arbitrary choices.) In fact, most — possibly all — Unices had a (largely unused) old swapper mechanism that hung around as process #0.

Linux, traditionally, has been somewhat different to the Unices, in that process #0 is the idle process, running cpu_idle(). It simply does nothing, in an infinite loop. It exists so that there's always a task ready to be scheduled.

Even this is an out-of-date description. The late 1980s and early 1990s was the advent of multi-threading operating systems, as a result of which process #0 became simply the system process. In the world of old single-threaded Unices, one could only get a separate flow of execution to do a continuous kernel task by fork()ing a process. So all of the kernel tasks (vmdaemon, pagedaemon, pagezero, bufdaemon, syncer, ktrace, and so forth on FreeBSD systems, for example) were low-numbered processes, fork()ed by process #0 after it fork()ed init. In multiple-threaded Unices, it makes no sense to create a whole new process context for something that runs entirely in kernel space, and doesn't need an address space, file descriptor table, and what not all to itself. So all of these tasks became (essentially) threads, sharing the address space of a single system process.

Along the way, several Unices finally lost the old swapper mechanism, that they were trying their utmost to avoid ever using anyway. OpenBSD's initialization code now simply drops into a while(1) tsleep(···); loop, for example, with no swapping mechanism anywhere.

So nowadays process #0 on a Unix is the system process, which effectively holds a number of kernel threads doing a number of things, ranging from page-out operations, through filesystem cache flushes and buffer zeroing, to idling when there's nothing else to run.

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 */
}
```

Expend the macro list_for_each():

```
struct task_struct *task;
struct list_head *list;
for (list = (&current->children)->next;
             prefetch(list->next), list != (&current->children);
             list = list->next)
{
  task = list_entry(list, struct task_struct, sibling);
}
```

- (Linux Kernel Linked List Explained) we need to access *the item itself* not the variable "list" in the item! macro list_entry() does just that.

  tmp = list_entry(pos, struct kool_list, list);

  Given

  1. a pointer to struct list_head,
  2. type of data structure it is part of, and
  3. it's name (struct list_head's name in the data structure)

  it returns a pointer to the data structure in which the pointer is part of.

- Sec 3.2.6, *The Process Family Tree*, [Lov10].

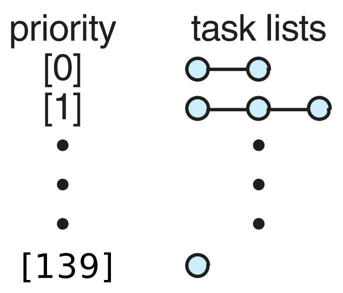
A task can be in multiple lists

```
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 */
}
```

ℹ

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



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

ℹ

prio (Sec 3.2.2.1, *prio*, [RFS05]) the dynamic priority of a process is a value that depends on the processes' scheduling history and the specified nice value. It is updated at sleep time, which is when the process is not being executed and when timeslice is used up. This value, prio, is related to the value of the static_prio field described next. The prio field holds +/- 5 of the value of static_prio, depending on the process' history; it will get a +5 bonus if it has slept a lot and a -5 handicap if it has been a processing hog and used up its timeslice.

static_prio (Sec 3.2.2.2, *static_prio*, [RFS05]) is equivalent to the nice value. The default value of static_prio is MAX_PRIO - 20. In our kernel, MAX_PRIO defaults to 140.

run_list (Sec 3.2.2.5, *The lists of TASK_RUNNING processes*, [BC05]) The trick used to achieve the scheduler speedup consists of splitting the runqueue in many lists of runnable processes, one list per process priority. 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.

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

- Each *runqueue (NOT each priority)* has a prio_array_t. (Sec 3.2.2.5, *The lists of TASK_RUNNING processes*, [BC05]) As we'll see, the kernel must preserve a lot of data for every runqueue in the system; however, the main data structures of a runqueue are the lists of process descriptors belonging to the runqueue; all these lists are implemented by a single prio_array_t data structure.

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.

2.3 Relationships Among Processes

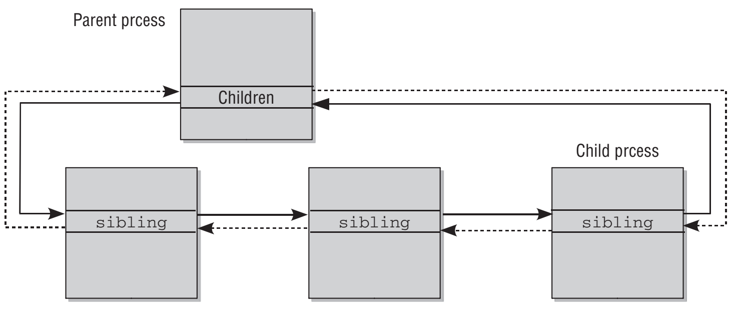
Relationships Among Processes

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

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

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

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 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)
```

The pidhash_shift variable stores the length in bits of a table index (11, in our example). The hash_long() function is used by many hash functions; on a 32-bit architecture it is essentially equivalent to:

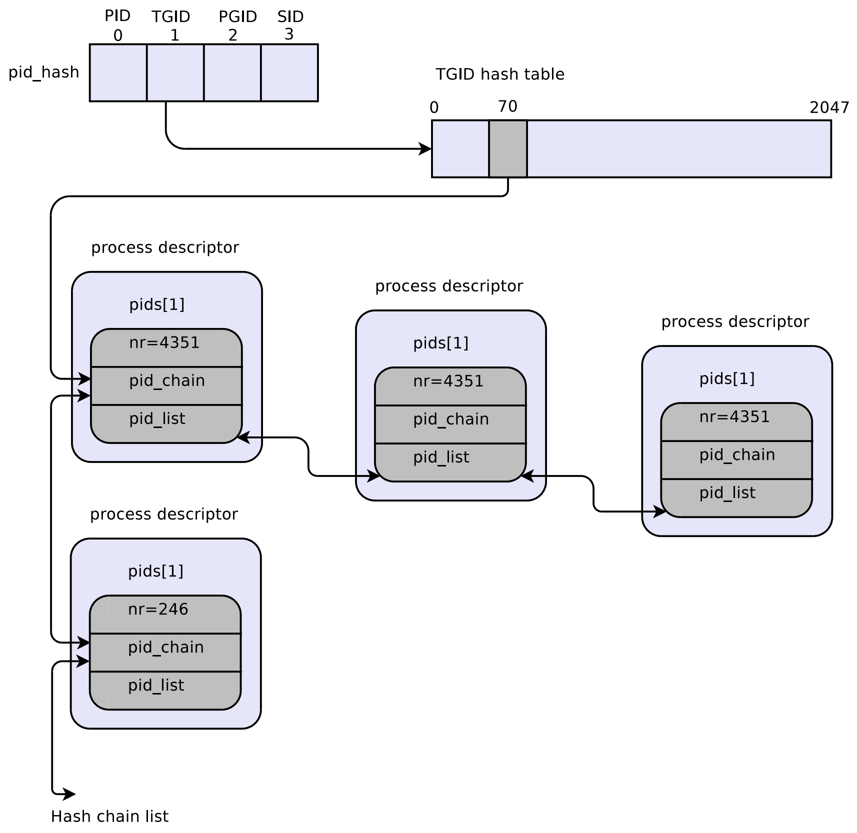
```
unsigned long hash_long(unsigned long val, unsigned int bits)
{
  unsigned long hash = val * 0x9e370001UL;
  return hash >> (32 - bits);
}
```

Because in our example pidhash_shift is equal to 11, pid_hashfn yields values ranging between 0 and $2^{11} - 1 = 2047$.

The pid data structure

```
struct pid                          struct task_struct{
{                                     ...
  int nr;                             struct pid pids[PIDTYPE_MAX];
  struct hlist_node pid_chain;        ...
  struct list_head pid_list;        }
};
```

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)

2.4 How Processes Are Organized

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?

2.5 Process Resource Limits

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

3 Process Switch

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

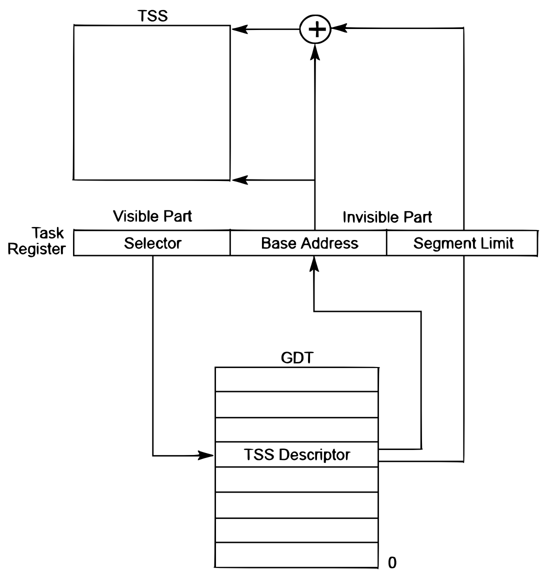
- One TSS for each CPU
  - The address of the kernel mode stack
  - I/O permission bitmap

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
-

Stackoverflow: **How does schedule() + switch_to() actually work?**

- When a process runs out of time-slice, the flag TIF_NEED_RESCHED is set by scheduler_tick(). (Called from the timer interrupt handler, set_tsk_need_resched())
- The kernel checks the flag, sees that it is set, and calls schedule() to switch to a new process. This flag is a message that schedule should be invoked as soon as possible because another process deserves to run. Upon returning to user-space or returning from an interrupt, the TIF_NEED_RESCHED flag is checked. If it is set, the kernel invokes the scheduler before continuing.

4 Creating Processes

REFs: [Mau08], [RFS05], [Lov10]

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;

- Sec *Forking* in chap. 3 of [Lov10]

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);
}
```

- Typically, a kernel thread continues executing its initial function forever (or at least until the system reboots, but with Linux you never know). The initial function usually implements a loop in which the kernel thread wakes up as needed, performs its duties, and then returns to sleep (Sec 3.4.2, *Kernel Threads*, [Lov10]).

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

5 Destroying Processes

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