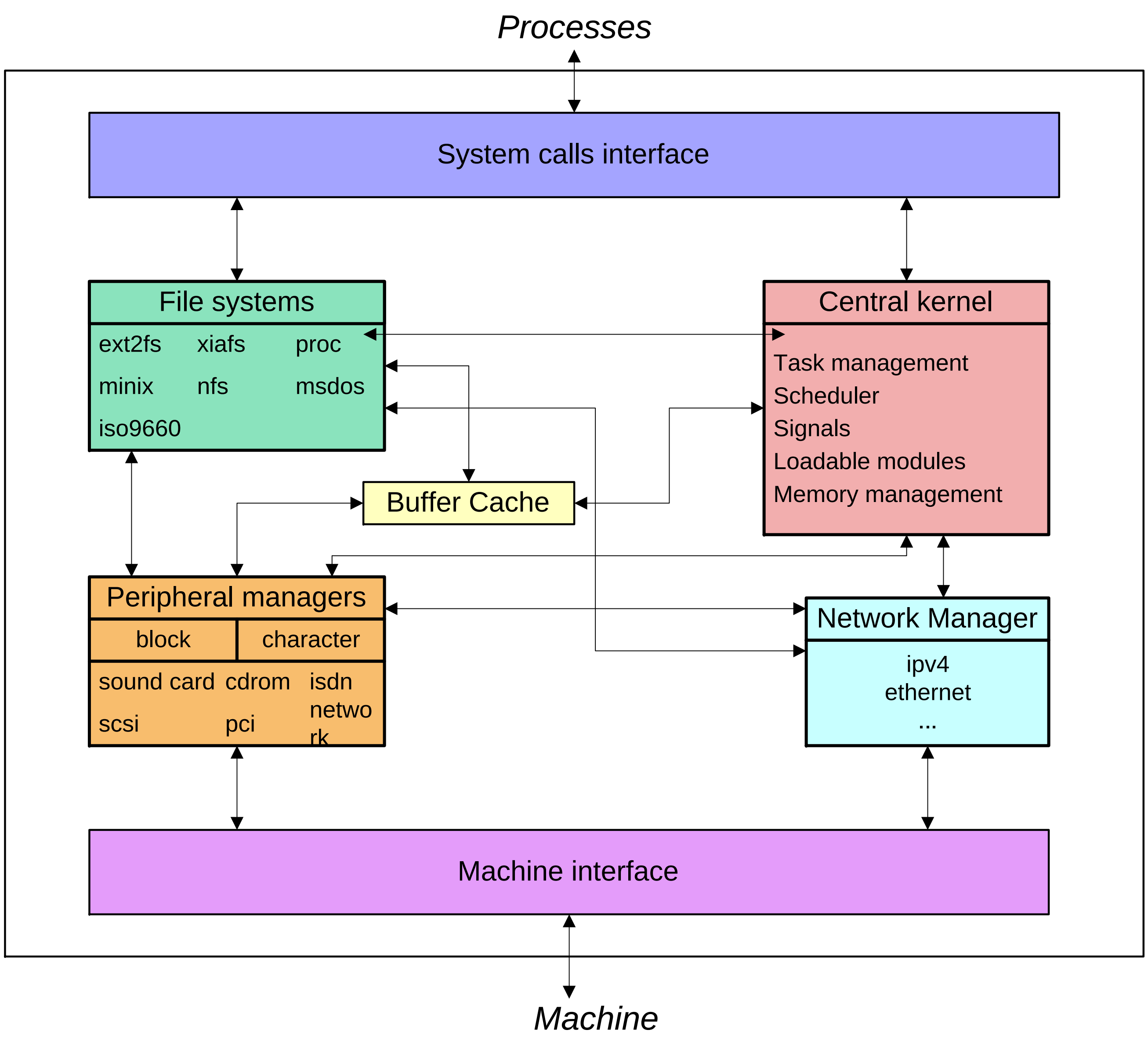
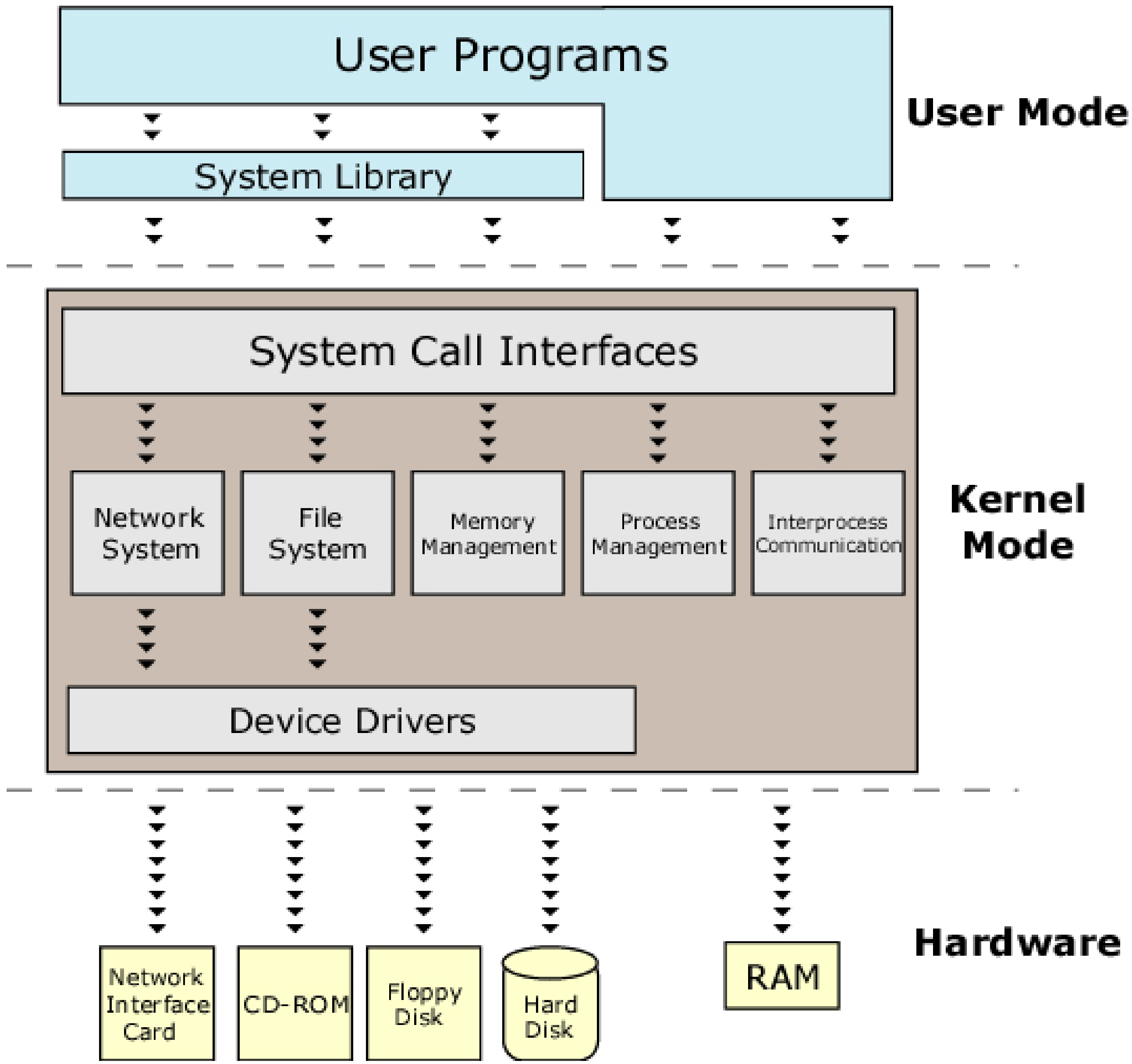
Linux Internals: Process and Process Management

System Structure



Linux Kernel Architecture



Entering Kernel Mode

A user process will enter kernel-mode

- When it decides to execute a system-call
- When it is interrupted (e.g. by the timer)
- When ‘exception’ occurs (e.g. divide by 0)

Switching to Kernel Stack

- Entering kernel involves a stack-switch
- Necessary for robustness
  - Ex. user-mode stack might be exhausted
- Desirable for security
  - Ex. illegal parameters might be supplied

Location of User-mode Stack

- Each task has a private user-mode stack
- The user-mode stack grows downward from the highest address in user space
  - From (0xBFFFFFFF), top 1 GB reserved for kernel space

What's on the User Stack?

Upon entering 'main()':

- A program's exit-address is on user stack
- Command-line arguments on user stack
- Environment variables are on user stack

During execution of 'main()':

- Function parameters and return-addresses
- Storage locations for local variables

What’s on the Kernel Stack?

Upon entering kernel-mode:

- task’s registers are saved on kernel stack (e.g., address of task’s user-mode stack)

During execution of kernel functions

- Function parameters and return addresses
- Storage locations for local variables

Outline of Things to Come Now

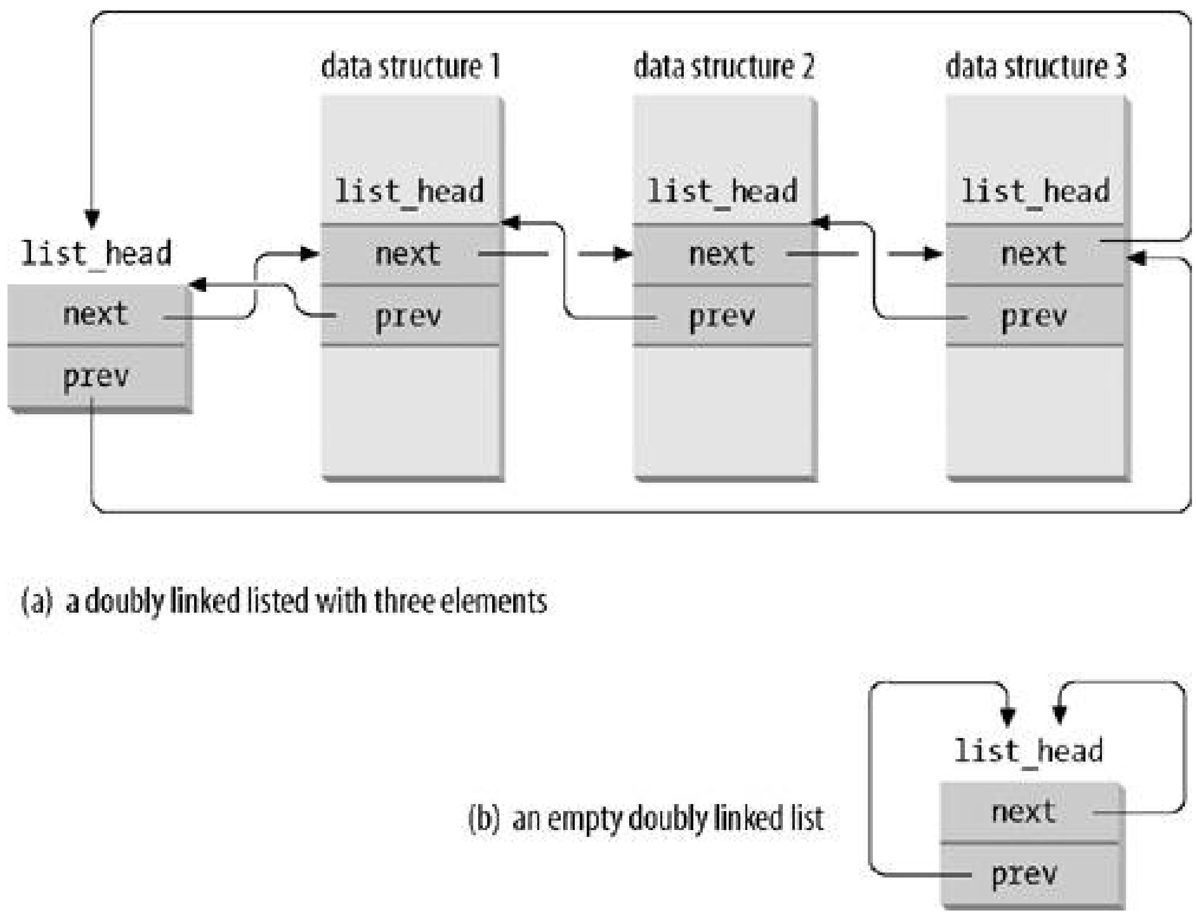
- Process Descriptors
- Identifying the Current Process
- Process States
- Process Lists and management
- Context Switch

Process Descriptors

Linux Processes

- Each process is represented by a *task_struct* data structure, containing:
  - Process State
  - Scheduling Information
  - Identifiers
  - Inter-Process Communication
  - Times and Timers
  - File system
  - Virtual memory
  - Processor Specific Context
  - Others…
- This is the generic PCB we studied earlier…
- In /include/linux/sched.h

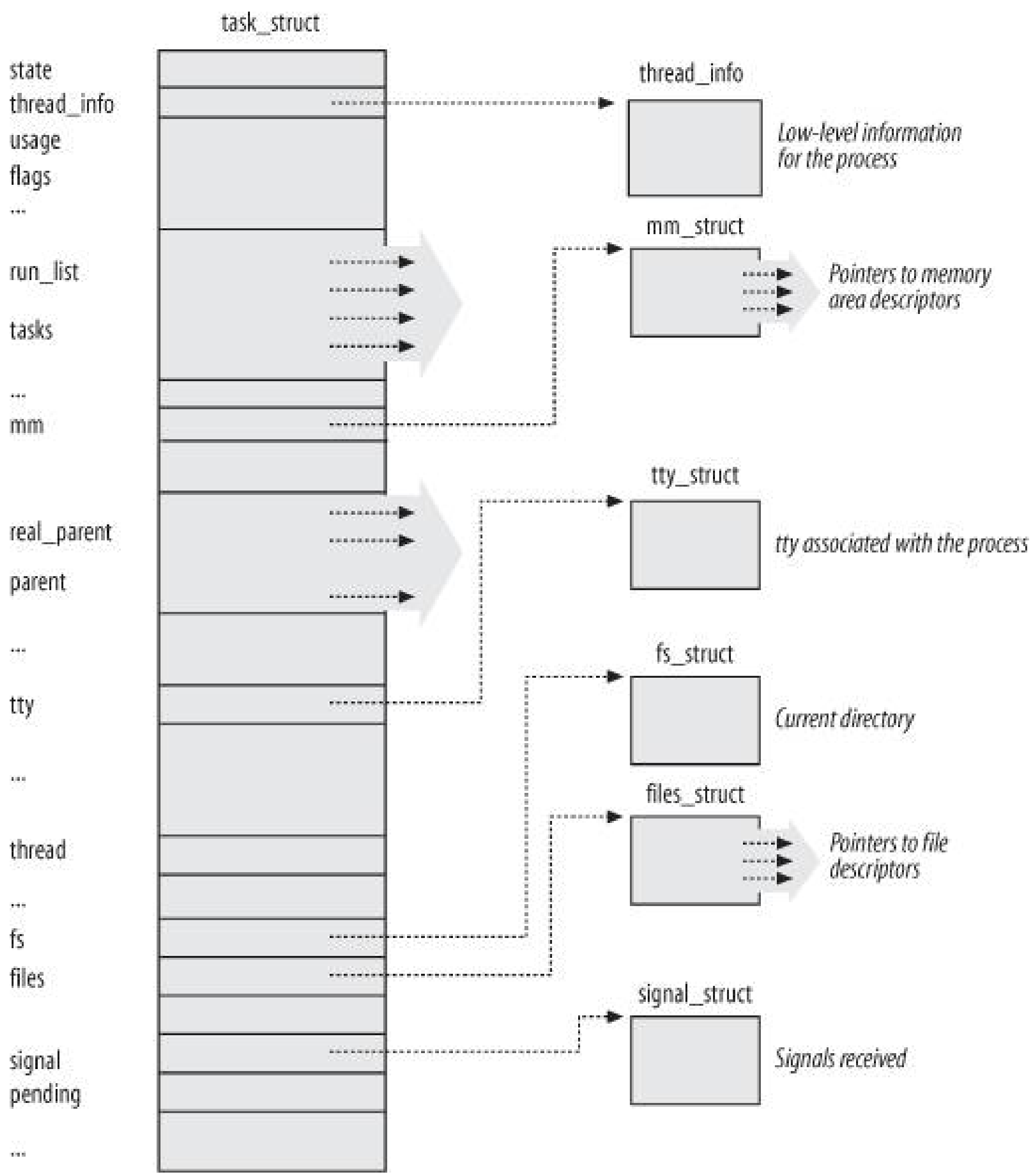
List of Processes: Doubly Linked List



task_struct: Some Fields

- state: process state
- thread_info: low-level information for the process
- mm: pointers to memory area descriptors
- tty: tty associated with the process
- fs: current directory
- files: pointers to file descriptors
- signal: signals received
- blocked: masked signals
- *next_task, *prev_task: links to next and previous tasks in the list of tasks
- priority
- policy: scheduling policy for this process (sched_FIFO, sched_RR, sched_others…)
- utime, stime, cutime, cstime, start_time: various time info
- Many other fields (total 100+)

Linux Process Descriptor



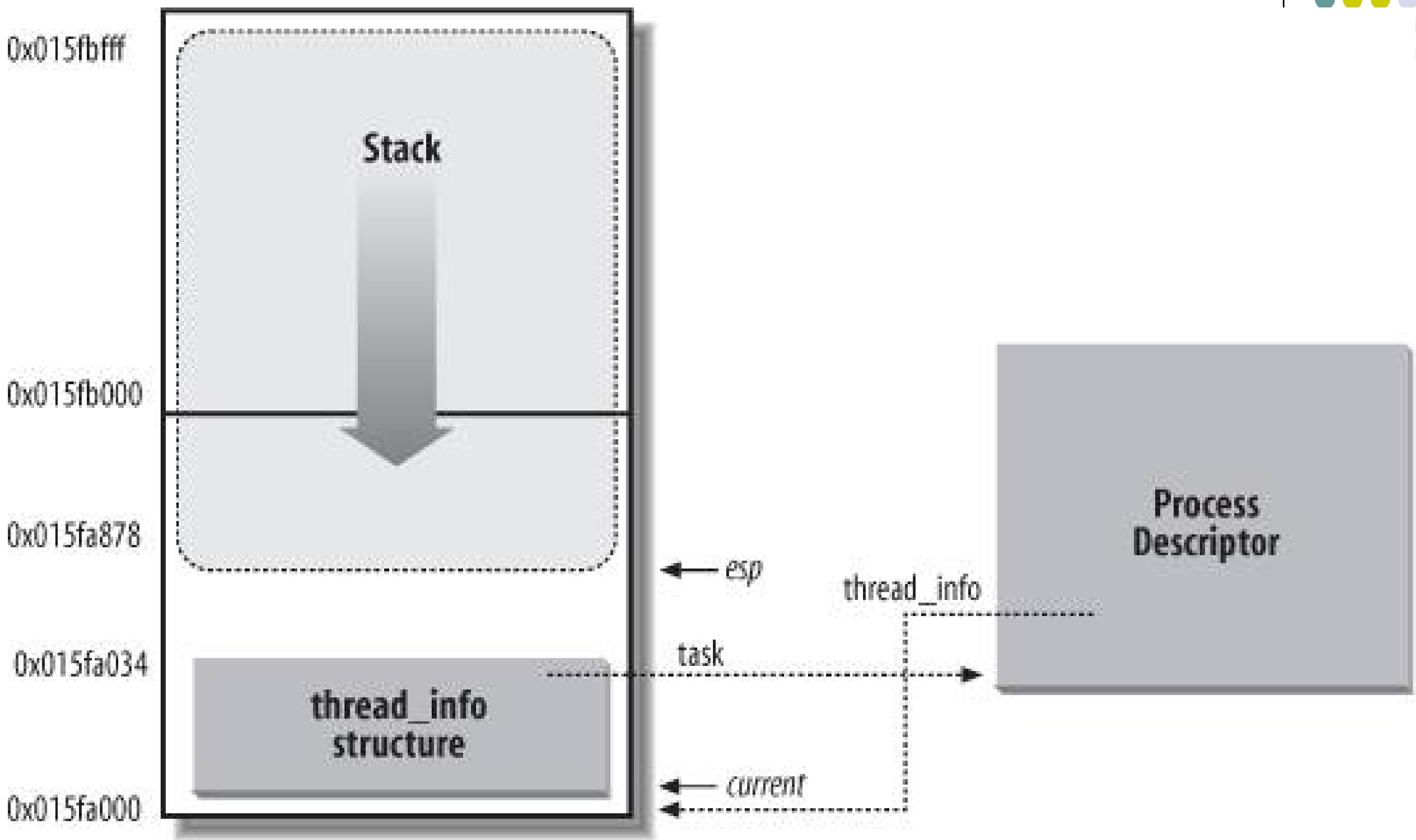
Where is the Descriptor Stored?

- Linux uses part of a task's kernel-stack to store that task's process descriptor
  - Actually *thread_info* structure is stored which has a task filed that points to the process descriptor
- The stack and descriptor are overlayed

```
union task_union {
    unsigned long    stack[INIT_THREAD_SIZE/sizeof(long)  ];
    struct  thread_info          task;
};
```

- Two pages (8KB assuming 4KB page size) allocated for stack and the *thread_info* structure
- Task's process descriptor is 1696 bytes
- So kernel stack can grow to about 6.5KB

  8192 bytes – 1696 bytes = 6496 bytes
- Each *task_union* object is '8KB-aligned'


0x015fbfff
Stack
0x015fb000
0x015fa878
esp
thread_info
Process
Descriptor
0x015fa034
task
thread_info
structure
current
0x015fa000

Why is this done like this?

- Identifying current process is now easy
- Linked to *esp*: current stack pointer
  - *thread_info* pointer = *esp* with lower 13 bits masked
- Obtain the address of *thread_info* structure from the *esp* register
  - AND the *esp* register value with 0xFFFFE000

```
movl %esp, %ebx
andl $0xFFFFE000, %ebx
```

(Now %ebx = thread_info's base-address)

- Kernel-headers define useful macros

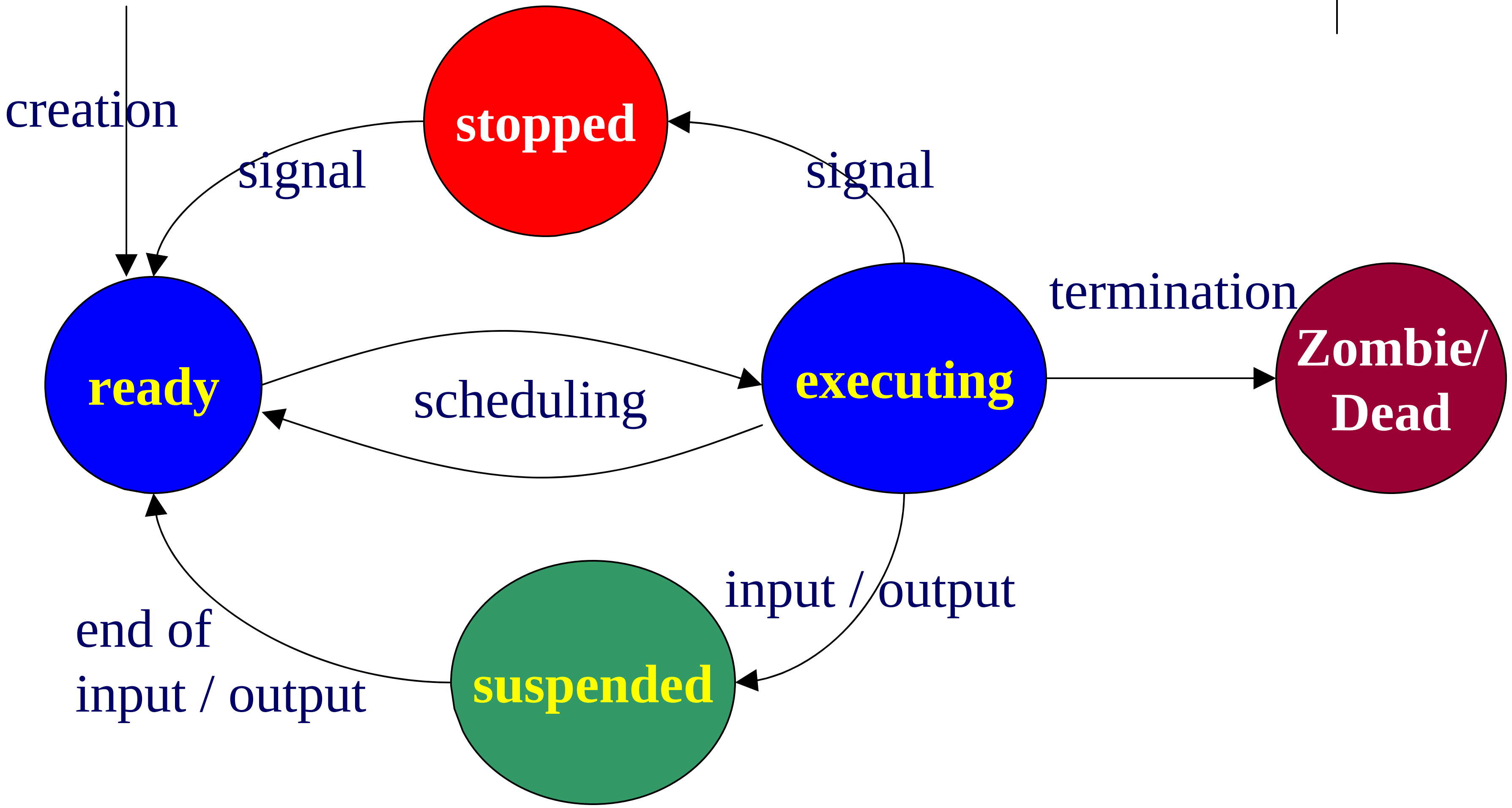
```
static inline struct task_struct *
        get_current( void )
        {
                struct task_struct      *current;
                __asm__( " andl %%esp, %0 ; " \
                : "=r" (current) : "0" (~0x1FFF)  );
                return   current;
        }
#define current get_current()
```

Process State

Process State

- **TASK_RUNNING** (executing on CPU or runnable)
- **TASK_INTERRUPTIBLE** (waiting on a condition: interrupts, signals and releasing resources may "wake" process)
- **TASK_UNINTERRUPTIBLE** (Sleeping process cannot be woken by a signal)
- **TASK_STOPPED** (stopped process e.g., by a SIGSTOP)
- **EXIT_ZOMBIE** (terminated before waiting for parent)
- **EXIT_DEAD** (terminated)

Process State



Identifying and Retrieving Processes

Identifying a Process

- Process descriptor pointers: 32-bit
- Process ID (PID): 16-bit (~32767 for compatibility)
- Each process, or independently scheduled execution context, has its own process descriptor
- Programmers expect threads in the same group to have a common PID
- *Thread group*: a collection of LWPs (Light Weight Processes)
- The PID of the first LWP in the group is the process id returned for all threads in the group
    - *tgid* field in process descriptor: using getpid() system call

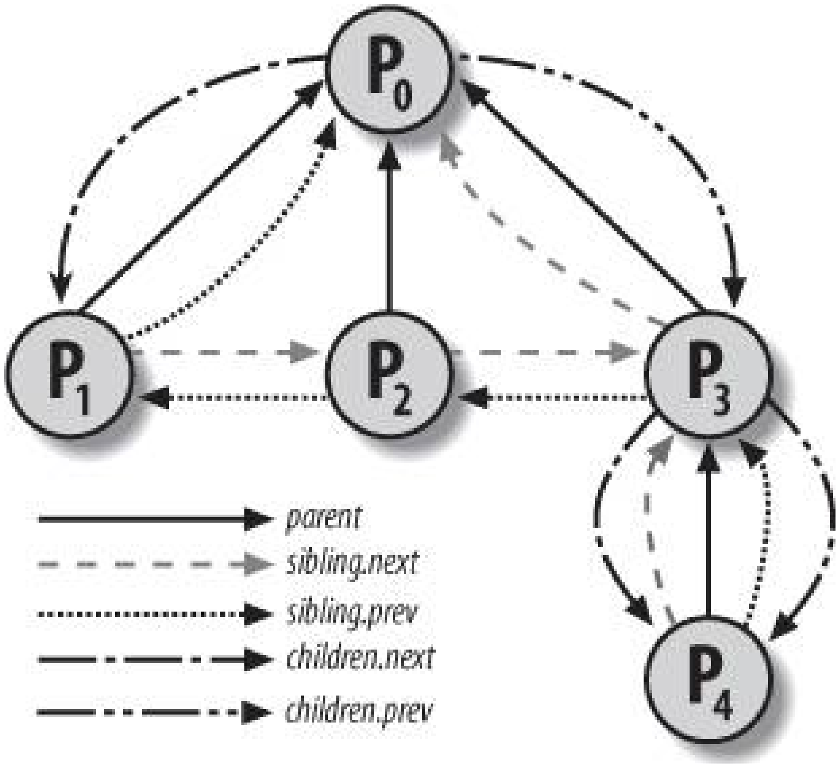
Parenthood

- New tasks get created by calling ‘fork()’
- Process 0 and 1: created by the kernel
    - Process 1 (*init*): the ancestor of all processes
- Old tasks get terminated by calling ‘exit()’
- When ‘fork()’ is called, two tasks return
- One task is known as the ‘parent’ process
- And the other is called the ‘child’ process
- The kernel keeps track of this relationship

Parenthood Relationships among Processes

- Process 0 and 1: created by the kernel
  - Process 1 (*init*): the ancestor of all processes
- Fields in process descriptor for parenthood relationships
  - real_parent
  - parent
  - children
  - sibling

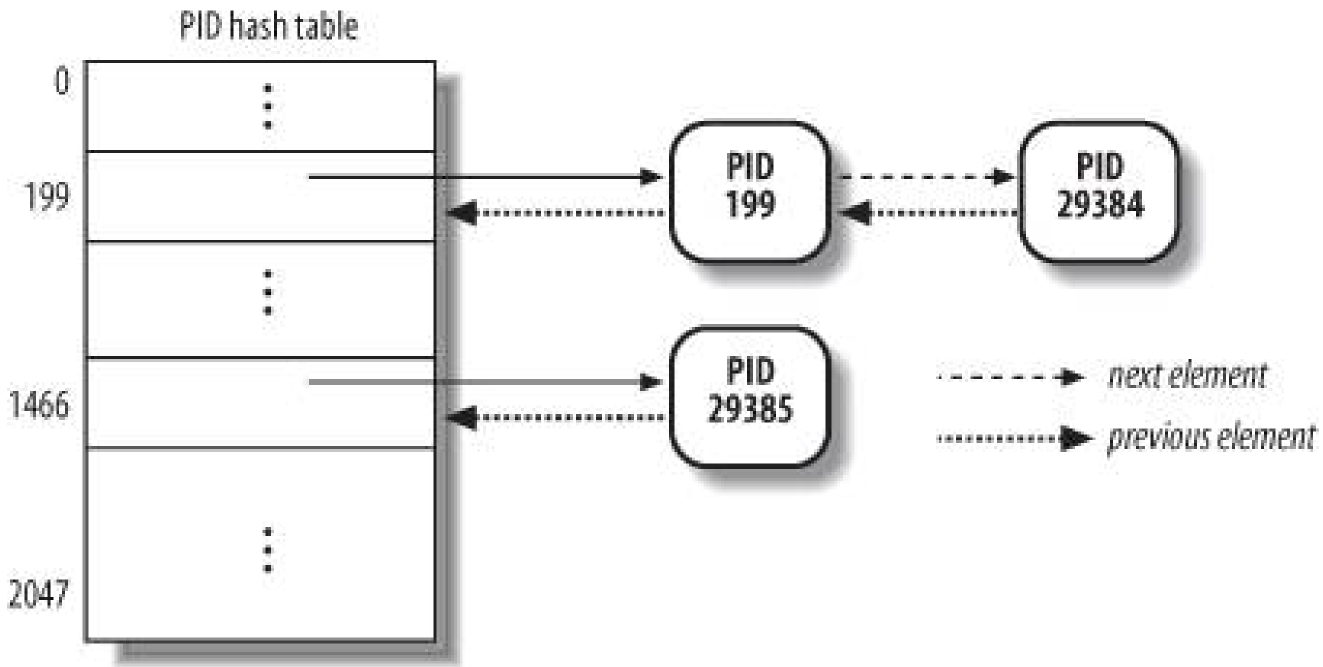
Parenthood Relationships among Five Processes



pidhash Table and Chained Lists

- User gives pid, need to map to process descriptor
- Sequential search in the process list by pid is inefficient
- PIDs are converted to matching process descriptors using a hash function.
  - A **`pidhash`** table maps PID to descriptor.
  - Collisions are resolved by chaining.
  - **`find_task_by_pid()`** searches hash table and returns a pointer to a matching process descriptor or **`NULL`**.

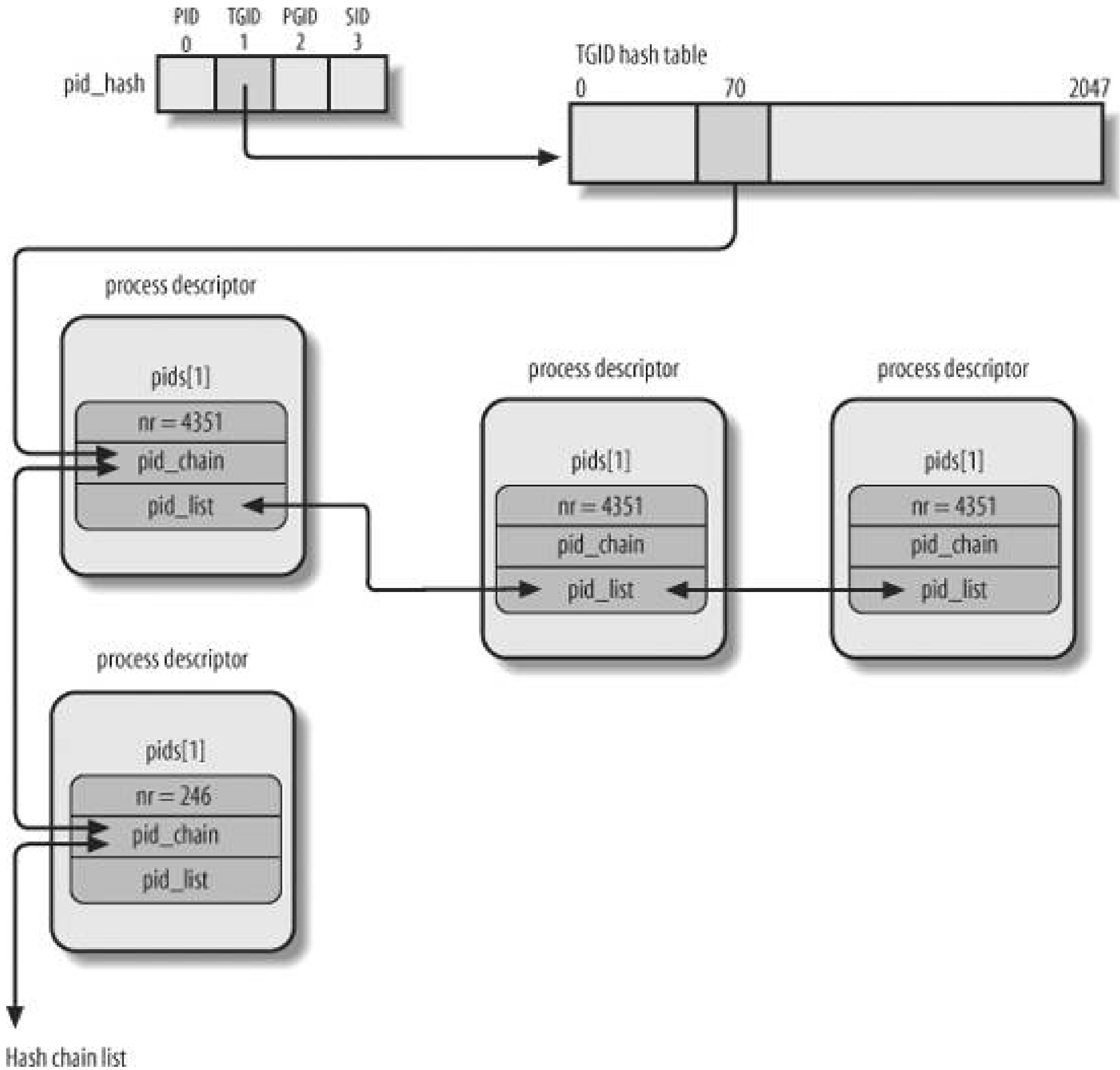
A Simple Example PID Hash Table and Chained Lists



- The *pid_hash* array contains four hash tables and corresponding field in the process descriptor
    - pid: PIDTYPE_PID
    - tgid: PIDTYPE_TGID (thread group leader)
    - pgrp: PIDTYPE_PGID (group leader)
    - session: PIDTYPE_SID (session leader)
- *Chaining* is used to handle PID *collisions*
- Size of each pidhash table: dependent on the available memory

- *pids* field of the process descriptor: the pid data structures
  - nr: PID number
  - pid_chain: links to the previous and the next elements in the hash chain list
  - pid_list: head of the per-PID list (in thread group)

The PID Hash Tables



Different Process Lists

How Processes are Organized

- Processes in TASK_STOPPED, EXIT_ZOMBIE, EXIT_DEAD: not linked in lists
- Processes in TASK_RUNNING: *run queue*
- Processes in TASK_INTERRUPTABLE, TASK_UNINTERRUPTABLE: *wait queues*
- Two kinds of sleeping processes
  - *Exclusive process*
  - *Nonexclusive process*: always woken up by the kernel when the event occurs

The Runqueue

- Processes are scheduled for execution from a doubly-linked list of **`TASK_RUNNING`** processes, called the **`runqueue`**.
  - **`prev_run`** & **`next_run`** fields of process descriptor are used to build **`runqueue`**.
  - **`init_task`** heads the list.
  - **`add_to_runqueue()`**, **`del_from_runqueue()`**, **`move_first_runqueue()`**, **`move_last_runqueue()`** functions manipulate list of process descriptors.
  - **`NR_RUNNING`** macro stores number of runnable processes.
  - **`wake_up_process()`** makes a process runnable.

- **QUESTION:** Is a *doubly-linked list* the best data structure for a run queue?

- Linux 2.6 implements the runqueue differently
  - To achieve scheduler speedup, Linux 2.6 splits the runqueue into 140 lists of each priority!
  - *array* field of process descriptor: pointer to the *prio_array_t* data structure
    - nr_active: # of process descriptors in the list
    - bitmap: priority bitmap
    - queue: the 140 list_heads
  - enqueue_task(p, array), dequeue_task(p, array)

Wait Queues

- `TASK_(UN)INTERRUPTIBLE` processes are grouped into classes that correspond to specific events.
  - e.g., timer expiration, resource now available.
  - There is a separate wait queue for each class / event.
  - Processes are “woken up” when the specific event occurs.

Wait Queues

- struct _ _wait_queue_head {
  spinlock_t lock;
  struct list_head task_list;
  };
  typedef struct _ _wait_queue_head wait_queue_head_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;

Wait Queue Example

```
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)
   schedule();
   Remove_wait_queue(wq, &wait);
}
```

- **`sleep_on()`** inserts the current process, P, into the specified wait queue and invokes the scheduler.
- When P is awakened it is removed from the wait queue.

Process Resource Limits

- RLIMIT_AS
- RLIMIT_CORE
- RLIMIT_CPU
- RLIMIT_DATA
- RLIMIT_FSIZE
- RLIMIT_LOCKS
- RLIMIT_MEMLOCK
- RLIMIT_MSGQUEUE
- RLIMIT_NOFILE
- RLIMIT_NPROC
- RLIMIT_RSS
- RLIMIT_SIGPENDING
- RLIMIT_STACK

Miscelleneous

Kernel Threads

- Some (background) system processes run only in kernel mode.
  - e.g., flushing disk caches, swapping out unused page frames.
  - Can use *kernel threads* for these tasks.
  - Ex.
    - Process 0 (swapper process), the ancestor of all processes
    - Process 1 (init process)
    - Others: keventd, kapm, kswapd, kflushd (also bdflush), kupdated, ksoftirqd
- Kernel threads only execute kernel functions – normal processes execute these fns via syscalls.
- Kernel threads only execute in kernel mode as opposed to normal processes that switch between kernel and user modes.

Process Termination

- Usually occurs when a process calls **`exit()`**.
  - Kernel can determine when to release resources owned by terminating process.
    - e.g., memory, open files etc.
- **`do_exit()`** called on termination, which in turn calls **`__exit_mm/files/fs/sighand()`** to free appropriate resources.
- Exit code is set for terminating process.
- **`exit_notify()`** updates parent/child relationships: all children of terminating processes become children of **`init`** process.
- **`schedule()`** is invoked to execute a new process.