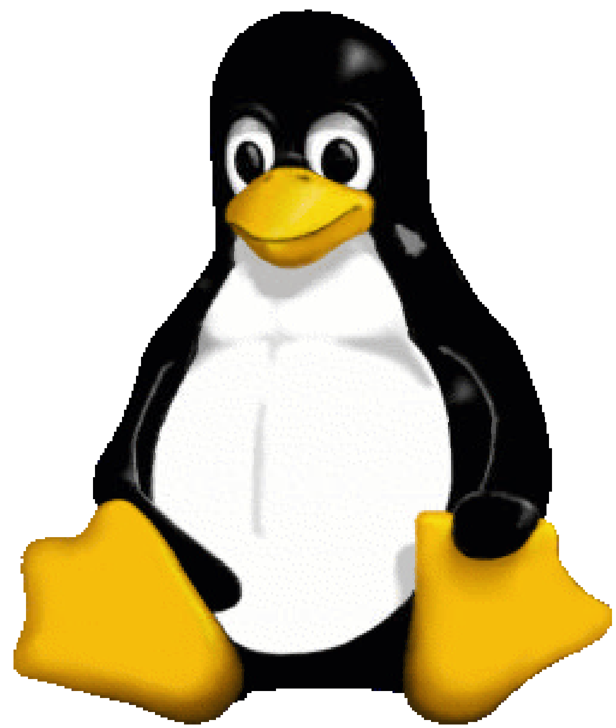
# The Linux Kernel: Process Management



# Process Descriptors

- The kernel maintains info about each process in a process descriptor, of type **`task_struct`**.
  - See **`include/linux/sched.h`**
  - Each process descriptor contains info such as run-state of process, address space, list of open files, process priority etc…

```
struct task_struct {
  volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
  unsigned long flags; /* per process flags */
  mm_segment_t addr_limit; /* thread address space:
                              0-0xBFFFFFFF for user-thead
                              0-0xFFFFFFFF for kernel-thread */
  struct exec_domain *exec_domain;
  long need_resched;
  long counter;
  long priority;
  /* SMP and runqueue state */
     struct task_struct *next_task, *prev_task;
     struct task_struct *next_run,  *prev_run;
     ...
  /* task state */
  /* limits */
  /* file system info */
  /* ipc stuff */
  /* tss for this task */
  /* filesystem information */
  /* open file information */
  /* memory management info */
  /* signal handlers */
     ...
};
```

**Contents of process descriptor**

# Process State

- Consists of an array of mutually exclusive flags*
  - *at least true for 2.2.x kernels.
  - *implies exactly one **`state`** flag is set at any time.
- **`state`** values:
  - **`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 debugger).
  - **`TASK_ZOMBIE`** (terminated before waiting for parent).

# Process Identification

- Each process, or independently scheduled execution context, has its own process descriptor.
- Process descriptor addresses are used to identify processes.
  - Process ids (or **PID**s) are 32-bit numbers, also used to identify processes.
  - For compatibility with traditional UNIX systems, LINUX uses PIDs in range 0..32767.
- Kernel maintains a **`task`** array of size **`NR_TASKS`**, with pointers to process descriptors. (Removed in 2.4.x to increase limit on number of processes in system).

# Process Descriptor Storage

- Processes are *dynamic*, so descriptors are kept in dynamic memory.
- An 8KB memory area is allocated for each process, to hold process descriptor *and* kernel mode process stack.
  - **Advantage**: Process descriptor pointer of **`current`** (running) process can be accessed quickly from stack pointer.
  - 8KB memory area = $2^{13}$ bytes.
  - Process descriptor pointer = **esp** with lower 13 bits masked.

# Cached Memory Areas

- 8KB (**`EXTRA_TASK_STRUCT`**) memory areas are cached to bypass the kernel memory allocator when one process is destroyed and a new one is created.
- **`free_task_struct()`** and **`alloc_task_struct()`** are used to release / allocate 8KB memory areas to / from the cache.

# The Process List

- The *process list* (of all processes in system) is a doubly-linked list.
  - **`prev_task`** & **`next_task`** fields of process descriptor are used to build list.
  - **`init_task`** (i.e., swapper) descriptor is at head of list.
    - **`prev_task`** field of **`init_task`** points to process descriptor inserted *last* in the list.
  - **`for_each_task()`** macro scans whole list.

# The Run Queue

- 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?

# Chained Hashing of PIDs

- 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`**.

# Managing the `task` Array

- The **`task`** array is updated every time a process is created or destroyed.
- A separate list (headed by **`tarray_freelist`**) keeps track of free elements in the **`task`** array.
  - When a process is destroyed its entry in the **`task`** array is added to the head of the freelist.

# 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 Queue Example

```
void sleep_on(struct wait_queue **wqptr) {
    struct wait_queue wait;
    current->state=TASK_UNINTERRUPTIBLE;
    wait.task=current;
    add_wait_queue(wqptr,&wait);
    schedule();
    remove_wait_queue(wqptr,&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 Switching

- Part of a process's execution context is its *hardware context* i.e., register contents.
  - The task state segment (**`tss`**) and kernel mode stack save hardware context.
    - **`tss`** holds hardware context not automatically saved by hardware (i.e., CPU).
- *Process switching* involves saving hardware context of **`prev`** process (descriptor) and replacing it with hardware context of **`next`** process (descriptor).
  - Needs to be fast!
  - Recent Linux versions override hardware context switching using software (sequence of **`mov`** instructions), to be able to validate saved data and for potential future optimizations.

# The switch_to Macro

- **switch_to()** performs a process switch from the **prev** process (descriptor) to the **next** process (descriptor).
- **switch_to** is invoked by **schedule()** & is one of the most hardware-dependent kernel routines.
  - See **kernel/sched.c** and **include/asm-*/system.h** for more details.

# Creating Processes

- Traditionally, resources owned by a parent process are duplicated when a child process is created.
  - *It is slow* to copy whole address space of parent.
    - *It is unnecessary*, if child (typically) immediately calls **`execve()`,** thereby replacing contents of duplicate address space.
- Cost savers:
  - *Copy on write* – parent and child share pages that are read; when either writes to a page, a new copy is made for the writing process.
  - *Lightweight processes* – parent & child share page tables (user-level address spaces), and open file descriptors.

# Creating *Lightweight* Processes

- LWPs are created using **`__clone()`**, having 4 args:
  - **`fn`** – function to be executed by new LWP.
  - **`arg`** – pointer to data passed to **`fn`**.
  - **`flags`** – low byte=sig number sent to parent when child terminates; other 3 bytes=flags for resource sharing between parent & child.
    - **`CLONE_VM`**=share page tables (virtual memory).
    - **`CLONE_FILES, CLONE_SIGHAND, CLONE_VFORK`** etc…
  - **`child_stack`** – user mode stack pointer for child process.
- **`__clone()`** is a library routine to the **`clone()`** syscall.
  - **`clone()`**takes **`flags`** and **`child_stack`** args and determines, on return, the id of the child which executes the **`fn`** function, with the corresponding **`arg`** argument.

# `fork()` and `vfork()`

- **`fork()`** is implemented as a **`clone()`** syscall with **`SIGCHLD`** sighandler set, all clone flags are cleared (no sharing) and **`child_stack`** is 0 (let kernel create stack for child on copy-on-write).
- **`vfork()`** is like **`fork()`** with **`CLONE_VM`** & **`CLONE_VFORK`** flags set.
  - With **`vfork()`** child & parent share address space; parent is blocked until child exits or executes a new program.

# do_fork()

- **do_fork()** is called from **clone()**:
  - **alloc_task_struct()** is called to setup 8KB memory area for process descriptor & kernel mode stack.
  - Checks performed to see if user has resources to start a new process.
  - **find_empty_process()** calls **get_free_taskslot()** to find a slot in the **task** array for new process descriptor pointer.
  - **copy_files/fs/sighand/mm()** are called to create resource copies for child, depending on **flags** value specified to **clone()**.
  - **copy_thread()** initializes kernel stack of child process.
  - A new PID is obtained for child and returned to parent when **do_fork()** completes.

# 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.
- 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.
- Kernel threads use linear addresses greater than PAGE_OFFSET – normal processes can access 4GB range of linear addresses.

# Kernel Thread Creation

- Kernel threads created using:
  - **`int kernel_thread(int (*fn)(void *), void *arg, unsigned long flags);`**
  - **`flags`**=**`CLONE_SIGHAND`**, **`CLONE_FILES`** etc.

# 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.