Linux Processes

COMS W4118

References: Operating Sys Concepts 9e, Understanding the Linux Kernel, previous W4118s
Copyright notice: care has been taken to use only those web images deemed by the instructor to be in the public domain. If you see a copyrighted image on any slide and are the copyright owner, please contact the instructor. It will be removed.
The major header files used for process management are:

- `include/linux/sched.h` – declarations for most task data structures
- `include/linux/threads.h` – some configuration constants (unrelated to threads)
- `include/linux/times.h` – time structures
- `include/linux/time.h` – time declarations
- `include/linux/timex.h` – wall clock time declarations
• The source code for process and thread management is in the `kernel` directory:
  `sched.c` – task scheduling routines
  `signal.c` – signal handling routines
  `fork.c` – process/thread creation routines
  `exit.c` – process exit routines
  `time.c` – time management routines
  `timer.c` – timer management routines

• The source code for the program initiation routines is in `fs/exec.c`. 
Linux: Processes or Threads?

• Linux uses a neutral term: tasks
  – Tasks represent both processes and threads

• Linux view
  – Threads: processes that share address space
  – Linux "threads" (tasks) are really "kernel threads"

• Lighter-weight than traditional processes
  – File descriptors, VM mappings need not be copied
  – Implication: file table and VM table not part of process descriptor
Stacks and task-descriptors

- To manage multitasking, the OS needs to use a data-structure which can keep track of every task’s progress and usage of the computer’s available resources (physical memory, open files, pending signals, etc.)
- Such a data-structure is called a ‘process descriptor’ — every active task needs one
- Every task needs its own ‘private’ stack
- So every task, in addition to having its own code and data, will also have a stack-area that is located in user-space, plus another stack-area that is located in kernel-space
- Each task also has a process-descriptor which is accessible only in kernel-space
Kernel Stacks

• Why need a special kernel stack?
  – Kernel can’t trust addresses provided by user
  – Address may point to kernel memory
  – Address may not be mapped
  – Memory region may be swapped out from physical RAM
  – Leftover data from kernel ops could be read by process

• Why a different stack for every process?
  – What to do if a process sleeps while executing kernel code?
  – Wasn’t a problem up to Linux 2.4
  – Kernel wasn’t pre-emptive
Pre-emptive Kernels

• Pre-emptive kernel different from process pre-emption
  – A non-preemptive kernel may not task switch while executing kernel code on behalf of a process
  – Up to Linux 2.4, implemented through BKL (big kernel lock)
  – Each syscall acquires BKL before execution
  – All other syscalls block. So, kernel code must run fast!
  – Inefficient on multicore architectures!
  – Finally removed in 2011

• Pre-emptive kernel: allow task switch while in kernel mode
  – What to do with kernel state?
  – Need per-process kernel stack!
  – What to do with interrupts?
  – Share process kernel stack (previously), or get their own (now)
  – All interrupts share single 4KB or 8KB kernel stack

• Which stack is being used determines kernel “context”
A task’s virtual-memory layout

Privilege-level 0

Kernel space

User-mode stack-area

User space

Shared runtime-libraries

Task’s code and data

Privilege-level 3

process descriptor and kernel-mode stack
Process Descriptor

• Process – dynamic, program in motion
  – Kernel data structures to maintain "state"
  – Descriptor, PCB (control block), task_struct
  – Larger than you think! (about 1K)
    – 160+ fields
    – Complex struct with pointers to others

• Type of info in task_struct
  – state, id, priorities, locks, files, signals, memory maps, locks, queues, list pointers, ...

• Some details
  – Address of first few fields hardcoded in asm
  – Careful attention to cache line layout
The Linux process descriptor

Each process descriptor contains many fields and some are pointers to other kernel structures which may themselves include fields that point to structures.

- state
- *stack
- flags
- *mm
- exit_code
- *user
- pid
- *files
- *parent
- *signal

- mm_struct
  - *pgd

- user_struct

- files_struct

- signal_struct

- pagedir[]
The Task Structure

- The `task_struct` is used to represent a task.
- The `task_struct` has several sub-structures that it references:
  - `tty_struct` – TTY associated with the process
  - `fs_struct` – current and root directories associated with the process
  - `files_struct` – file descriptors for the process
  - `mm_struct` – memory areas for the process
  - `signal_struct` – signal structures associated with the process
  - `user_struct` – per-user information (for example, number of current processes)
- Linux uses part of a task’s kernel-stack page-frame to store thread information
- The thread_info includes a pointer to the task’s process-descriptor data-structure
Finding a task’s ‘thread-info’

- During a task’s execution in kernel-mode, it’s very quick to find that task’s `thread_info` object
- Just use two assembly-language instructions:

```assembly
movl $0xFFFFF000, %eax
andl %esp, %eax
```

Ok, now `%eax` = the thread-info’s base-address

- Masking off 13 bits of the stack yields `thread_info`
- Macro `current_thread_info` implements this computation
- `thread_info` points to `task_struct`
- `current` macro yields the `task_struct`
- `current` is not a static variable, useful for SMP
Finding task-related kernel-data

• Use a macro ‘\texttt{task\_thread\_info(task)}’ to get a pointer to the ‘thread\_info’ structure:
  \begin{verbatim}
  struct thread_info *info = task_thread_info(task);
  \end{verbatim}

• Then one more step gets you back to the address of the task’s process-descriptor:
  \begin{verbatim}
  struct task_struct *task = info->task;
  \end{verbatim}
• PID: 16-bit process ID
• `task_structs` are found by searching for `pid` structures, which point to the `task_structs`. The `pid` structures are kept in several hash tables, hashed by different IDs:
  - process ID
  - thread group ID // pid of first thread in process
  - process group ID // job control
  - session ID // login sessions
    - (see `include/linux/pid.h`)

• Allocated process IDs are recorded in a bitmap representing around four million possible IDs.
• PIDs dynamically allocated, avoid immediate reuse
Process Relationships

• Processes are related
  – Parent/child (fork()), siblings
  – Possible to "re-parent"
    • Parent vs. original parent
  – Parent can "wait" for child to terminate

• Process groups
  – Possible to send signals to all members

• Sessions
  – Processes related to login
Task Relationships

• Several pointers exist between *task_structs*:
  - *parent* – pointer to parent process
  - *children* – pointer to linked list of child processes
  - *sibling* – pointer to task of "next younger sibling" of current process

• *children* and *sibling* point to the *task_struct* for the first thread created in a process.

• The *task_struct* for every thread in a process has the same pointer values.
Task States

From kernel-header: <linux/sched.h>

- #define TASK_RUNNING 0
- #define TASK_INTERRUPTIBLE 1
- #define TASK_UNINTERRUPTIBLE 2
- #define TASK_STOPPED 4
- #define TASK_TRACED 8
- #define EXIT_ZOMBIE 16
- #define EXIT_DEAD 32
- #define TASK_NONINTERACTIVE 64
- #define TASK_DEAD 128
Task States

- **TASK_RUNNING** – the thread is running on the CPU or is waiting to run
- **TASK_INTERRUPTIBLE** – the thread is sleeping and can be awoken by a signal (EINTR)
- **TASK_UNINTERRUPTIBLE** – the thread is sleeping and cannot be awakened by a signal
- **TASK_STOPPED** – the process has been stopped by a signal or by a debugger
- **TASK_TRACED** – the process is being traced via the `ptrace` system call
- **TASK_NONINTERACTIVE** – the process has exited
- **TASK_DEAD** – the process is being cleaned up and the task is being deleted
We have two separate sets of flags: task->state is about runnability, while task->exit_state are about the task exiting. Confusing, but this way modifying one set can't modify the other one by mistake.

- **EXIT_ZOMBIE** – the process is exiting but has not yet been waited for by its parent
- **EXIT_DEAD** – the process has exited and has been waited for