

Processes

- Process descriptor (`task_struct`)
 - Static properties of processes
 - State, id, relationships, wait queue, limits
- Process switching (context switch)
 - Hardware context, TSS, `switch_to()`
 - Saving floating-point registers
- Creating processes
 - `clone()`, `fork()`, `vfork()`
 - Kernel threads (vs. user threads)
- Destroying processes
 - Termination vs. removal

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)
 - Complex struct with pointers to others
- Type of info in `task_struct`
 - Registers, 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

Process State

- Traditional (textbook) view
 - Blocked, runnable, running
 - Also initializing, terminating
 - UNIX adds "stopped" (signals, ptrace())
- Linux (TASK_whatever)
 - Running, runnable (RUNNING)
 - Blocked (INTERRUPTIBLE, UNINTERRUPTIBLE)
 - Interruptible – signals bring you out of syscall block (EINTR)
 - Terminating (ZOMBIE)
 - Dead but still around – "living dead" processes
 - Stopped (STOPPED)

Process Identity

- Users: pid; Kernel: address of descriptor
 - Pids dynamically allocated, reused
 - 16 bits – 32767, avoid immediate reuse
 - Pid to address hash
 - 2.2: static task_array
 - Statically limited # tasks
 - This limitation removed in 2.4
- `current->pid` (macro)

Descriptor Storage/Allocation

- Descriptors stored in kernel data segment
 - Each process gets a 2 page (8K) "kernel stack" used while in the kernel (security)
 - `task_struct` stored here; rest for stack
 - Easy to derive descriptor from `esp` (stack ptr)
 - Implemented as union `task_union { }`
- Small (16) cache of free `task_unions`
 - `free_task_struct()`, `alloc_task_struct()`

Descriptor Lists, Hashes

- Process list
 - init_task, prev_task, next_task
 - for_each_task(p) iterator (macro)
- Runnable processes: runqueue
 - init_task, prev_run, next_run, nr_running
 - wake_up_process()
 - Calls schedule() if "preemption" is necessary
- Pid to descriptor hash: pidhash
 - hash_pid(), unhash_pid()
 - find_hash_by_pid()

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

Wait Queues

- Blocking implementation
 - Change state to TASK_(UN)INTERRUPTIBLE
 - Add node to wait queue
 - All processes waiting for specific "event"
 - Usually just one element
 - Used for timing, synch, device i/o, etc.
 - Structure is a bit optimized
 - struct wait_queue usually allocated on kernel stack

sleep_on(), wake_up()

- sleep_on(), sleep_on_interruptible()
 - See code on LXR
- wake_up(), wake_up_interruptible()
 - See code on LXR
 - Process can wakeup with event not true
 - If multiple waiters, another may have resource
 - Always check availability after wakeup
 - Maybe wakeup was in response to signal
- 2.4: wake_one()
 - Avoids "thundering herd" problem
 - A lot of waiting processes wake up, fight over resource; most then go back to sleep (losers)
 - Bad for performance; very bad for bus, cache on SMP machine

Process Limits

- Optional **resource** limits (accounting)
 - getrlimit(), setrlimit() (user control)
 - Root can establish rlim_min, rlim_max
 - Usually RLIMIT_INFINITY
- Resources (RLIMIT_whatever)
 - CPU, FSIZE (file size), DATA (heap), STACK,
 - CORE, RSS (frames), NPROC (# processes),
 - NOFILE (# open files), MEMLOCK, AS

Process Switching - Context

- Hardware context
 - Registers (including page table register)
 - Hardware support but Linux uses software
 - About the same speed currently
 - Software might be optimized more
 - Better control over validity checking
 - prev, next task_struct pointers
- Linux TSS (thread_struct)
 - Base registers, floating-point, debug, etc.
 - I/O permissions bitmap
 - Intel feature to allow userland access to i/o ports!
 - ioperm(), iopl() (Intel-specific)

Process Switching – `switch_to()`

- Invoked by `schedule()`
 - Very Intel-specific (mostly assembly code)
 - GCC magic makes for tough reading
- Some highlights
 - Save basic registers
 - Switch to kernel stack of next
 - Save fp registers if necessary
 - Unlock TSS
 - Load `ldtr`, `cr3` (paging)
 - Load debug registers (breakpoints)
 - Return

Process Switching – FP Registers

- This is pretty weird
- Pentium – on chip FPU
 - Backwards compatibility, ESCAPE prefix
 - Not saved by default
 - MMX Instructions use FPU
- FP registers
 - Saved "on demand", reload "when needed" (lazily)
- TS Flag set on context switch
 - FP instructions cause exception (device unavailable)
 - Kernel intervenes by loading FP regs, clearing TS
 - `unlazy_fpu()`, `math_state_retstore()`

Creating Processes

- `clone()`, `fork()`, `vfork()`
 - Fork duplicates (most) parent resources
 - Exec tears down old address space and installs a new one (corresponding to process image on disk)
 - Most forks are part of a fork-exec sequence
 - Wasteful to copy resources that are then overwritten
- Old solution (hack): `vfork`
 - Parent/child share; parent blocks until child ends
- New solution: COW copy-on-write
 - Share r/w pages r/o until write (fault), then copy
- Linux solution: `clone()`
 - Specify what resources to share/duplicate
 - `CLONE_VM`, `_FS`, `_FILES`, `_SIGHAND`, `_PID`

Processes vs. Threads

- Traditional processes – big, "heavy"
 - Lot's of data, takes time to startup
- Newer idea: threads – small, "lightweight"
 - Share address space, files, sockets, etc.
 - Basically context + stack
- Usually "user-level", many per process
- But kernel can benefit from threads as well
 - Kernel threads: in the kernel address space

Benefits of Threading

- Lightweight context switch, blocking
 - Increases CPU (quantum) utilization
- Logically structures concurrent activities
 - Avoids uglier "async" code: e.g. sig handlers
- Possible to exploit parallelism (SMP)
 - If you have **kernel** threads!
 - Kernel doesn't "know about" user threads
 - Kernel threads are "unit of scheduling"

Linux: Processes or Threads?

- Linux uses a neutral term: tasks
- Traditional view
 - Threads exist "inside" processes
- Linux view
 - Threads: processes that share address space
 - Linux "threads" (tasks) are really "kernel threads"

Thread Models

- Many-to-one
 - User-level threads; kernel doesn't know about them
- One-to-one
 - Linux standard model; each user-level thread corresponds to a kernel thread
- Many-to-many (m-to-n; $m \geq n$)
 - Solaris, Next Generation POSIX Threads
 - Large number of user threads corresponds to a smaller number of kernel threads
 - More flexible; better CPU utilization

clone()

- fork() is implemented as a wrapper around clone() with specific parameters
- __clone(fp, data, flags, stack)
 - "__" means "don't call this directly"
 - fp is thread start function, data is params
 - flags is or of CLONE_ flags
 - stack is address of user stack
 - clone() calls do_fork() to do the work

do_fork()

- Highlights
 - alloc_task_struct()
 - Copy current into new
 - find_empty_process()
 - get_pid()
 - Update ancestry
 - Copy components based on flags
 - copy_thread()
 - Link into task list, update nr_tasks
 - Set TASK_RUNNING
 - wake_up_process()

Kernel threads

- Linux has a small number of kernel threads that run continuously in the kernel (daemons)
 - No user address space (only kernel mapped)
- Creating: `kernel_thread()`
- Process 0: idle process
- Process 1
 - Spawns several kernel threads before transitioning to user mode as `/sbin/init`
 - `kflushd` (`bdfush`) – Flush dirty buffers to disk under "memory pressure"
 - `kupdate` – Periodically flushes old buffers to disk
 - `kswapd` – Swapping daemon
 - `kpiod` – No longer used in 2.4

Destroying Processes

- Termination
 - kill(), exit()
- Removal
 - wait()