From the Abstract to the Concrete

- We've been discussing processes and threads at a high level
- We're now going to look at how they're implemented on Linux
- It's often helpful to read the appropriate Linux kernel files along with the text

What’s Different About Reality?

- Performance matters
- Therefore, data structures matter
- Provisions need to be made for things we haven’t talked about
- *Much less abstract*

High-Level Concepts

- Processes: creation, termination
- Threads: creation, termination
- Waiting for an event
- Interrupts and traps

Processes versus Threads

- To the kernel, threads are a lot like processes
- In Linux, threads are implemented as *light-weight proceses*

Lightweight Processes

- Similar to ordinary process, but share some resources
- Lighter-weight for the kernel because things like open file descriptors and virtual memory tables need not be copied
- Implication: open file table and virtual memory table can't be part of process structure

The Linux `task_struct`

- Stores per-process information
- As noted earlier, some of the data is a pointer to other data structures.
- Doesn't contain the kernel stack

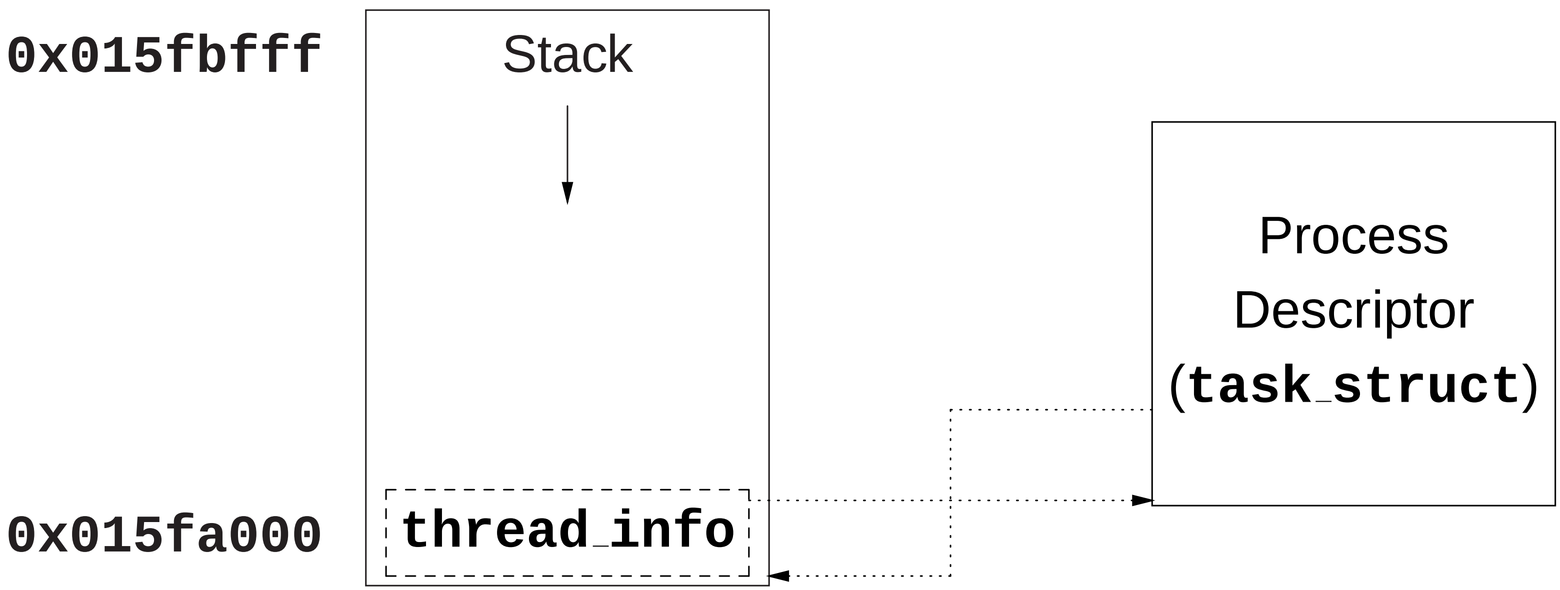
Indirect Data Structures

- `thread_info` (see below)
- Virtual memory — shared between threads
- Current directory — shared between threads
- Open files — shared between threads
- Signal information — shared between threads
- Current tty — shared in the process group

Kernel Stack

- Stack size is 8K — the kernel doesn't use many automatic variables
- Stored in the same data structure as the **`thread_info`** structure
- **`thread_info`** — at a known offset from the top of the stack — points to the **`task_struct`** entry

Stack Layout



The **current** macro points to the current process' descriptor.

Efficiency and the Current Process Descriptor

- Masking off 13 bits of the stack address points to the **`thread_info`** structure
- The first 4 bytes of **`thread_info`** point to **`task_struct`**
- Thus, **`current`** is efficiently calculable from the stack pointer
- The current process descriptor is *not* in a static variable — very useful for multiprocessors. (Why?)
- (Why can every stack be at the same address?)

Other Process Data Structures

- A **`pid`** field for each process — and a **`tgid`** field for the *thread group leader*'s **`pid`**
- Lots of doubly-linked lists (why doubly-linked?):
  - All processes
  - Array of runnable queues, one for each priority level
  - Family relationships: parent, siblings, child
  - Wait queues
- Hash table to convert from **`pid`** to **`task_struct`**

Wait Queues

- Hold list of processes that are not runnable
- Many wait queues, one for each resource
- Two types of wait queue:
  - Sharable resource ready — wake up all processes
  - Exclusive resource ready — wake only one process

Sleeping Processes

- Processes put themselves to sleep
- Create and initialize a **`waitqueue`** variable
- Add this process to a wait queue
- Call the scheduler

☞ It will run another process

- When awakened, remove the entry from the wait queue and return to the caller
- (The **`waitqueue`** variable is a local variable — why is that legitimate?)

Sleeping is Really Much More Complex

- Many different ways for processes to sleep
- Interruptible and non-interruptible sleeps
- Race conditions
- Timeouts
- Multiprocessor complexity

Waking a Process

- Some other context wakes a process
  - An interrupt
  - Another thread in that process
  - Another process
  - A kernel thread or process
- Note well: an interrupt does not run in a process' context — we'll see much more on this later

Switching Processes

- Very complex
- Very machine-dependent
- See the book for the gory details

High-Level View

- Save registers and other state for one process
- Load registers and other state for the next process
- Important state: address space

Where is Stuff Saved?

- Save general registers on the kernel stack
- Save other hardware context in **`thread_struct`** (part of **`task_struct`**)
- Also have a hardware-defined *Task State Segment*; some per-process fields are stored there

Important Principles

- One process must save state where another can find it
- When the new state is loaded, the CPU *is* running another process — the state *is* the process
- The stack pointer determines most of the state

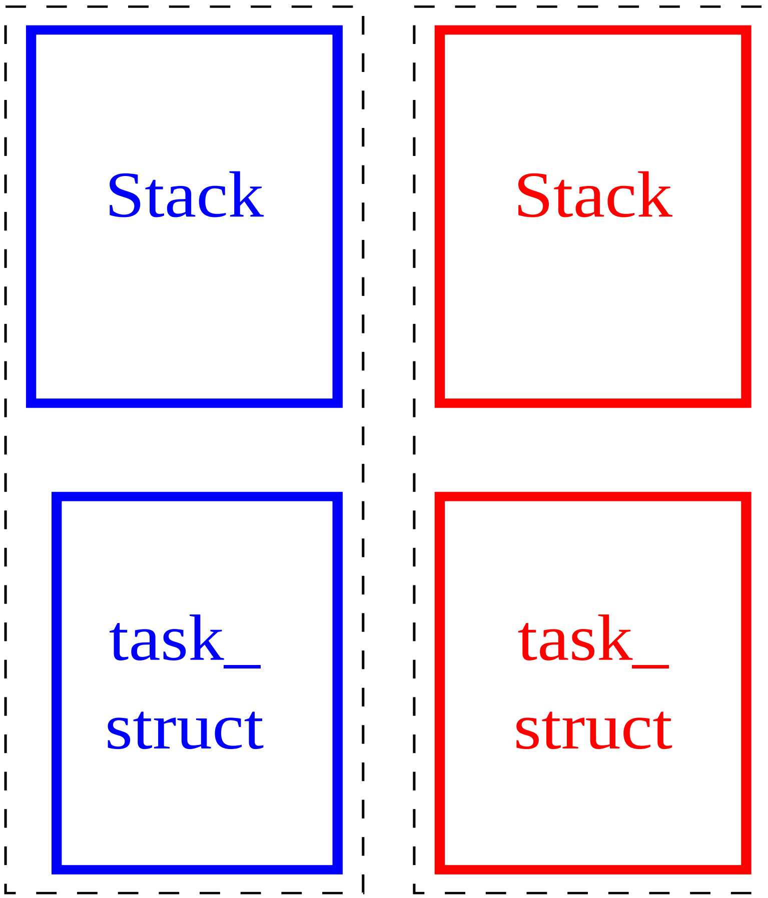
The Stack Pointer and the State

- Some of the registers are on the stack
- The stack pointer determines the location of **`thread_info`**
- **`thread_info`** points to **`task_struct`**
- *Changing the stack pointer changes the process*

Switching Processes

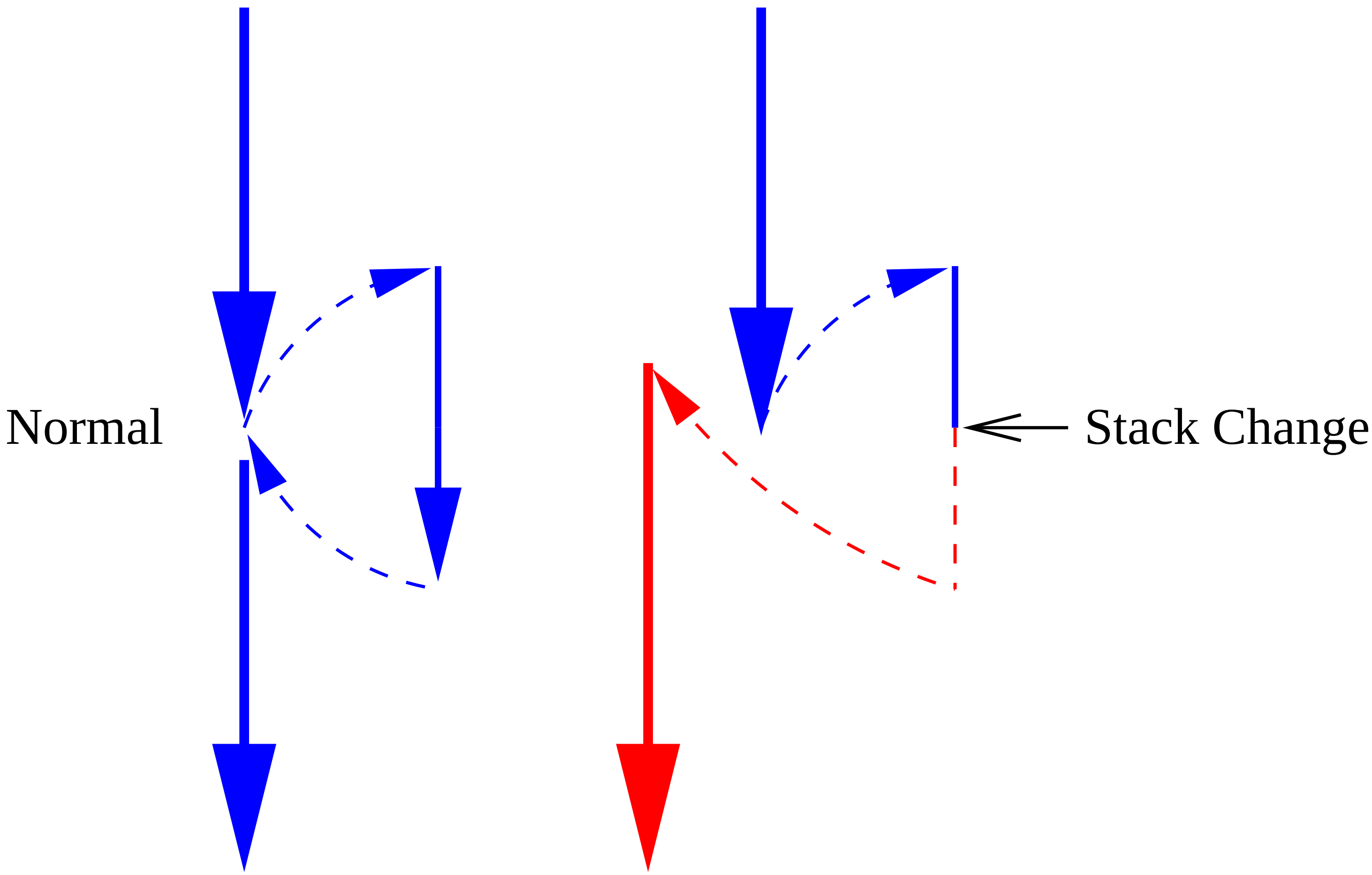
- Enter a subroutine; push registers onto the stack
- Save other state in **`thread_struct`**
- *Change the stack pointer*
- Restore state from the *new* **`thread_struct`**
- Restore registers from the *new* stack
- Return to the *new* process' caller

Switching Stacks



- Called by someone
- Push registers
- Save state
- *Change to Red stack*
- Restore state
- Pop registers
- Return to Red caller

Flow of Control During Stack Change



Floating Point Registers

- Floating point (FPU) and MMX instructions use a separate set of registers
- SSE and SSE2 instructions use yet another set of registers
- FPU/MMX and SSE/SSE2 registers are *not* automatically saved on interrupts

Floating Point Registers

- Legacy issue: floating point originally handled by outboard (expensive) chip
- Expense: it takes a fair number of cycles to save and restore these registers
- Rarity: most processes don't use floating point

Lazy Save/Restore

- Hardware flag set on process switch
- If process issues floating point instruction and flag is set, trap
- Kernel then does a save/restore on the floating point registers
- A software flag is set for this process
- Any time it's set, floating point registers are restored for that process at switch time
- Bottom line: only done if needed; if only one process uses floating point, no save/restore needed

Creating a Process

- Three types: **`fork()`**, **`clone()`**, **`vfork()`**
- **`fork()`** is traditional: duplicate process

☞ Can be expensive

- **`clone()`** is used for lightweight processes
- **`vfork()`** is an efficiency hack

Create a New Process: `fork()`

- Allocate a new PID
- Save floating point registers if needed
- Allocate memory for a new **`task_struct`**
- Allocate memory for a new stack
- Copy the old **`task_struct`** and stack to the new ones, modifying the pointers appropriately
- Copy other data, such as address space and open files
- Put the new process on the run queue

Returning from `fork()`

- The new process has the same stack — and hence the same return address — as the old one
- It will therefore return to the same spot
- Very minor changes are made to variables, so that it returns a child process indication rather than a parent process (0 instead of the child's PID)
- Similar magic to process switching

Copying Indirect Data Structures

- Open files: new set of file descriptors point to shared open file table
- Virtual memory: copy virtual memory page table, but set up for *copy on write* semantics

Copy on Write

- Both page tables point to the same memory pages
- Mark all pages non-writable
- If a process writes to a page, it causes a page fault
- That page is copied, a new page table entry is created in one process for the copy; both copies are marked writable in both processes
- Usually, the child process will **`exec()`** a new program soon; not many pages are copied

Creating a Light-Weight Process

- Requires help from userland to create a thread: a new user-level stack needs to be allocated
- Process creation is similar to `fork()`, except for copying indirect data structures
- Page table is shared; nothing is marked read-only
- Open file table is shared, too

Efficiency Hack: `vfork()`

- Copying a page table can be expensive
- Write protection traps are expensive
- Most new processes execute a new program almost immediately anyway; there's no major need for a copy of the address space
- **`vfork()`** freezes the parent process and uses its address space for the child process
- When the child process **`exec()s`** a new program, the parent is released
- Saves a lot of data copying

Process Exit

- Difference between process and thread exit
- Remove process from lots of queues; free certain data structures if not in use by other processes
- For process exit, must close open files

☞ Note: this can block; on misbehaving systems, can block forever, leaving the process unkillable!

- Become a zombie process

Zombies

- Processes don't fully exit until the parent process issues a **`wait()`** system call
- Allows the parent to check exit status of child processes
- Data structure is finally cleaned up and freed after this happens
- If a parent exits before its children, the child processes become children of process 1
- Process 1 is always sitting in a **`wait()`** loop

Recap

- The entire mechanism is driven by the data structures
- Context switches happen by creating return values in the new data structure
- When the new structure is referenced, the new context is magically used