Operating Systems

Tutorial 1

Michael Tänzer

os-tut@nhng.de
http://os-tut.nhng.de

16th November 2010

Outline

How can the kernel stack be located?

- Global kernel internal variable or
- Entry in the TCB (if one kernel stack per thread)
- Need to save user stack without overwriting any registers and load the kernel stack pointer
- Most architectures provide hardware support for stack switch on mode change:

  IA32 Loads the kernel stack pointer (`esp0`) from a hardware-known segment called TSS (thread state segment)

  MIPS Switches the entire register bank on mode change. User mode bank may be accessed in kernel mode but not the other way around

What state has to be saved by the hardware and the OS when handling interrupts, exceptions and traps?

Traps

- Explicitly called by the running thread

$\Rightarrow$ Application can save most of its state it wants to preserve

- At least the instruction pointer and probably also the stack pointer have to be saved by the hardware

Interrupts and Exceptions

- Thread is not prepared for them to happen

$\Rightarrow$ All accessible state has to be saved

- If some state is guaranteed not to be changed by the event handler, it may be excluded (e. g. floating point registers, application memory)

Step-by-step: interrupt/exception handling on stack-based architectures

1. Switch to kernel mode
2. Save current IP and SP in processor internal registers
3. Set up the kernel stack pointer (KSP)
4. Push saved IP, SP and possibly an exception reason
5. Load the interrupt vector (KIP, kernel instruction pointer)

---

6. Push remaining registers
7. Inspect exception reason and branch to specific handler

How do register-based architecture differ?

Only for your information

1. Switch to kernel mode
2. Disable interrupts
3. Save return address in special exception register
4. Save current IP in return address register
5. Switch register banks or have scratch registers
6. Load the interrupt vector (KIP)

---

7. Push user SP on kernel stack (using scratch registers)
8. Load the kernel stack pointer (KSP, unless we have switched register bank)
9. Save register state on stack
10. Inspect exception reason
11. Enable interrupts
12. Branch to specific handler

What happens when an interrupt occurs while executing in kernel mode?

- Don’t load the KSP again
- If we did we would reset it although there still is valid data on top of it
- Just add a new frame

Where might it be beneficial to use threads?

- Video games: One thread for graphics, physics, AI, ...
- Web server: One thread listens for incoming connections and dispatches them each to a new thread
- Image processing: Partition the image into *n* parts and process each part in a separate thread

Multi-threading vs. multiple processes

+ Lower overhead
  - Thread creation
  - Switching between threads of the same process (no address space switch needed)
+ Data sharing: Threads can easily work on the same data and communicate via shared memory (most operating systems also allow sharing of memory between processes but that's more complicated)
– Less isolation: Need to be careful when determining which variables should be synchronised

Discuss the three threading models

- Many-to-One
- One-to-One
- Many-to-Many

With regard to

- Performance
- Behaviour on blocking syscalls
- Utilisation of multi processor systems
- Kernel awareness

Many-to-One

+ Fast thread switch and creation (kernel not involved)
– A blocking syscall in one thread will block the whole process
– Can only use one processor
- Kernel does not know about threads (i. e. will also work on kernels which don't have a thread implementation)
+ Fully customizable (e. g. might use a specialised scheduling algorithm optimized for the application)

One-to-One

- Each thread creation and switch requires kernel invocation
+ A blocking syscall will only affect the thread doing the syscall
+ Can use as many processors as there are threads
- Kernel does all the bookkeeping and management of threads $\Rightarrow$ the kernel needs to have a thread implementation

Many-to-Many

- Creation and switch of kernel threads needs kernel invocation but on user threads it doesn't
- A blocking syscall in one thread will block those threads that are mapped to the same kernel thread
- Can use as many processors as there are kernel threads
- Kernel needs to know about threads. Also the two schedulers will want to exchange data to make informed decisions (even more complex)

Which types of events can trigger a One-to-One thread switch?

Voluntary

- Calling `yield()`
- Make a blocking syscall

Involuntary

- End of time-slice
- High priority thread becoming ready
- Device interrupt
- Exception that can't be handled immediately
- Exception that leads to aborting the thread/process

Which types of events can trigger a Many-to-One thread switch?

“Jobs are either I/O- or compute-bound. In neither case would user-level threads be a win.

Why would one go for pure user-level threads at all?”

What is the purpose of kernel-mode threads?

Would it make sense to clone threads on `fork`?

Questions & Comments

Any questions or comments?

Politically Correct UNIX

System VI Release notes

- `man` pages are now called `person` pages
- Similarly `hangman` is now the `person_executed_by_an_oppressive_regime`
- The `more` command reflects the materialistic philosophy of the Reagan era. System VI uses the environmentally preferable `less` command
- There will no longer be a invidious distinction between ‘dumb’ and ‘smart’ terminals. All terminals are equally valuable
- No longer will it be permissible for files and processes to be ‘owned’ by users. All files and processes will own themselves, and decide how (or whether) to respond to requests from users.