Protection and the Kernel

Review: Thread-Structured Proxy Server

- network driver
- HTTP request handler
- distill, encrypt, distill, file/cache manager, scrubber
- stats, logging

- main thread; waiting for child termination
- periodic threads; waiting for timer to fire
- server threads; waiting on queues of data messages or pending requests (e.g., device interrupts)
- worker threads; waiting for data to be produced/consumed
Adding Protection

If modules are mutually distrusting, separating them and their threads into separate processes or protection domains prevents them from accessing each other’s memory.

A protected kernel creates/manages protected execution contexts for all processes/domains: all interactions among threads in different processes/domains must go through a set of shared kernel/executive objects.

Our interaction primitives may be supported at the kernel system call interface.

A Protected Kernel

CPU mode (a field in some status register) indicates whether the CPU is running in a user program or in the protected kernel.

Some instructions or data accesses are only legal when the CPU is executing in kernel mode.
The Kernel

- Today, all “real” operating systems have protected kernels. The kernel resides in a well-known file: the “machine” automatically loads it into memory (boots) on power-on/reset.
  
  Our “kernel” is called the executive in NT.

- The kernel is (mostly) a library of service procedures shared by all user programs, but the kernel is protected:
  
  User code cannot access internal kernel data structures directly, and it can invoke the kernel only at well-defined entry points (system calls).

- Kernel code is like user code, but the kernel is privileged:
  
  The kernel has direct access to all hardware functions, and defines the machine entry points for interrupts and exceptions.

Kernel-Supported Threads

Most newer OS kernels have kernel-supported threads.

- thread model and scheduling defined by OS
  
  Nachos kernel can support them: extra credit in Labs 4 and 5 NT, advanced Unix, Linux, etc.

  New kernel system calls, e.g.:
  - thread_fork
  - thread_exit
  - thread_block
  - thread_alert
  - thread_join
  - etc...

  Threads must enter the kernel to sleep: no blocking in user space

  Kernel scheduler (not a library) decides which thread to run next.

  Threads can block independently in kernel system calls.
Nachos as a Thread Library

The Nachos library implements concurrent threads.

- no special support needed from the kernel (use any Unix)
- thread creation and context switch are fast (no syscall)
- defines its own thread model and scheduling policies
- library threads are sometimes called coroutines, lightweight threads, or fibers in NT.

```
while(1) {
    t = scheduler->FindNextToRun();
    scheduler->Run(t);
}
```

Threads vs. Processes

1. The process is a kernel abstraction for an independent executing program.
   - includes at least one “thread of control”
   - also includes a private address space (VAS)
     - VAS requires OS kernel support
   - often the unit of resource ownership in kernel
     - e.g., memory, open files, CPU usage

2. Threads may share an address space.
   - Threads have “context” just like vanilla processes.
     - thread context switch vs. process context switch
   - Every thread must exist within some process VAS.
   - Processes may be “multithreaded” with thread primitives supported by a library or the kernel.
Two Views of Threads in Nachos

1. Nachos is a thread library running inside a Unix (Solaris) process, with no involvement from the kernel.
   SPARC interrupts and Solaris timeslicing are invisible.
   the Nachos scheduler does its own pseudo-random timeslicing.

2. Nachos is a toolkit for building a simulated OS kernel.
   Threads are a basis for implementing Nachos processes; when running in kernel mode they interact/synchronize as threads.
   Nachos kernel’s timeslicing is implemented in the scheduler.
   - driven by timer interrupts on the “simulated machine”
   A Nachos kernel could provide a kernel interface for threads.
Nachos Thread States and Transitions

In Labs 1-3 we are only concerned with the states in this box.

When running in user mode, the thread executes within the SPIM machine simulator.

Mode, Space, and Context
Protecting Entry to the Kernel

Protected events and kernel mode are the architectural foundations of kernel-based OS (Unix, NT, etc).

- The machine defines a small set of exceptional event types.
- The machine defines what conditions raise each event.
- The kernel installs handlers for each event at boot time.
  
  e.g., a table in kernel memory read by the machine

  The machine transitions to kernel mode only on an event.

  The kernel defines the event handlers.

  Therefore the kernel chooses what code will execute in kernel mode, and when.

CPU Events: Interrupts and Exceptions

- an “unnatural” change in control flow
- an interrupt is caused by an external event
device requests attention, timer expires, etc.
- an exception is caused by an executing instruction
CPU requires software intervention
- kernel handler routine for each event type

<table>
<thead>
<tr>
<th></th>
<th>unplanned</th>
<th>deliberate</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync</td>
<td>fault</td>
<td>syscall trap</td>
</tr>
<tr>
<td>async</td>
<td>interrupt</td>
<td>AST</td>
</tr>
</tbody>
</table>
The Virtual Address Space

A typical process VAS space includes:
- user regions in the lower half
  V->P mappings specific to each process
  accessible to user or kernel code
- kernel regions in upper half
  shared by all processes
  accessible only to kernel code
- **Nachos**: process virtual address space includes only user portions.
  mappings change on each process switch

A VAS for a private address space system (e.g.,
Unix) executing on a typical 32-bit architecture.

Processes and the Kernel

n-bit virtual address space

32-bit virtual address space

system call traps

...and upcalls (e.g.,
signals)
System Call Traps

User code initiates system call traps to invoke kernel services.

- procedural interface for user code (through standard library)
  syscall stub or wrapper routine for each syscall
  executes a special trap instruction (e.g., chmk or callsys)
  syscall arguments/results passed in registers or user stack

<table>
<thead>
<tr>
<th>Alpha CPU architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>read() in libc.a (executes in user mode):</td>
</tr>
<tr>
<td>move arg0...argn, a0...an # syscall args in registers A0..AN</td>
</tr>
<tr>
<td>move SYSSTOP_READ, v0 # syscall dispatch code in V0</td>
</tr>
<tr>
<td>callsys # kernel trap</td>
</tr>
<tr>
<td>move r1, _errno # (return unless high-order R0 bit set)</td>
</tr>
<tr>
<td># errno = return status</td>
</tr>
</tbody>
</table>

Handling a System Call Trap (Alpha)

1. **Machine** saves return address and switches to kernel stack.
   - save user SP, global pointer(GP), PC on kernel stack
   - set kernel mode and transfer to a syscall trap handler (entSys)

2. **Trap handler** saves software state, and dispatches.
   - save some/all registers/arguments on kernel stack for this thread
   - vector to syscall routine through sysent[v0: dispatchcode]

3. Trap handler returns to user mode.
   - when syscall routine returns, restore user register state
   - execute privileged return-from-syscall instruction (e.g., retsys)
   - machine restores SP, GP, PC and sets user mode
   - emerges at user instruction following the callsys
Questions About System Call Handling

1. Why not just use `bcopy` instead of `copyin` and `copyout`?
2. What should `copyin` and `copyout` do?
3. What stack should the system call handler use?
   Why not use the user stack? A global kernel stack?
4. What would happen if the kernel did not save all registers?
5. Where should per-process kernel global variables reside?
   syscall arguments (consider size) and error code
   How should the system handle multiple threads per-process?
6. What if the kernel executes a `callsys` instruction? What if user code executes a `retsys` instruction?

Handling Interrupts and Faults

At a high level, interrupts and faults are similar to syscalls.
   Machine saves user context (SP, PC, GP), changes mode to kernel, and transfers to handler.
   Event handler saves other machine state as needed, etc.
   Handle event; restore context/mode in effect when it occurred.

But there are some key differences:
   Interrupts and faults are not requested by the user program.
     faults are specific to a thread/process: execute on that thread’s kernel stack
   Interrupts occur asynchronously.
     typically execute on a shared system-wide interrupt stack
   Faults may be resolved by re-executing the faulting instruction.
     touch the saved PC to re-execute faulting instruction after return from fault
Questions About Interrupts and Faults

1. What stack should an interrupt handler use? fault handler?
   user stack? process kernel stack? global kernel stack?
2. What happens if an interrupt handler does not save and
   restore all registers?
   (ask Drew)
3. How can we report an error from an interrupt handler?
4. Do nested interrupts cause any special problems?
5. How about nested faults?

Architectural Foundations of OS Kernels

- A privileged execution context (*kernel mode*)
- System call *trap* instruction
  User processes enter the kernel to access OS services
- Virtual memory mapping
  OS control of virtual-physical translations
- Device interrupts to notify the kernel of I/O completion etc.
  Timer hardware and clock interrupts to periodically return
  control to the kernel
- Protected device control registers
- Privileged instructions to control basic machine functions
- Atomic instructions for coordination on multiprocessors
Summary: Mode, Space, and Context

<table>
<thead>
<tr>
<th></th>
<th>Process context</th>
<th>System context</th>
</tr>
</thead>
<tbody>
<tr>
<td>User mode</td>
<td>Application</td>
<td>N/A</td>
</tr>
<tr>
<td>Kernel mode</td>
<td>Syscall or</td>
<td>Interrupt or</td>
</tr>
<tr>
<td></td>
<td>exception</td>
<td>System task</td>
</tr>
</tbody>
</table>

- **User mode**: Application
- **Kernel mode**: Syscall or exception
- **Interrupt or System task**: System context

![Diagram showing mode, space, and context]