Traps and Faults

Review: Mode and Space
Review: the Role of Events

A CPU event is an “unnatural” change in control flow.
Like a procedure call, an event changes the PC.
Also changes mode or context (current stack), or both.
Events do not change the current space!

The kernel defines a handler routine for each event type.
Event handlers always execute in kernel mode.
The specific types of events are defined by the machine.
Once the system is booted, every entry to the kernel occurs as a result of an event.
In some sense, the whole kernel is a big event handler.

Categorizing Events

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 to handle a fault or trap.

<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>

AST: Asynchronous System Trap
Also called a software interrupt or an Asynchronous or Deferred Procedure Call (APC or DPC)

Note: different “cultures” may use some of these terms (e.g., trap, fault, exception, event, interrupt) slightly differently.
System Call Traps

User code invokes kernel services by initiating system call traps.

- Programs in C, C++, etc. invoke system calls by linking to a standard library of procedures written in assembly language.
  - the library defines a stub or wrapper routine for each syscall
  - stub executes a special trap instruction (e.g., chmk or callsys)
  - syscall arguments/results passed in registers or user stack

### Alpha CPU architecture

```plaintext
(read()) in Unix libc.a library (executes in user mode):
```

```
#define SYSCALL_READ 27  # code for a read system call
move arg0...argn, a0...an  # syscall args in registers A0..AN
move SYSCALL_READ, v0  # syscall dispatch code in V0
callsys  # kernel trap
move r1, _errno  # errno = return status
return
```

“What Bullet-Proofing” the Kernel

System calls must be “safe” to protect the kernel from buggy or malicious user programs.

1. System calls enter the kernel at a well-known safe point.
   - Enter at the kernel trap handler; control transfers to the “middle” of the kernel are not permitted.

2. The kernel validates all system call arguments before use.
   - Kernel may reject a request if it is meaningless or if the user process has inadequate privilege for the requested operation.

3. All memory used by the system call handler is in kernel space, so it is protected from interference by user code.

*What stack does the system call execute on?*
Kernel Stacks and System Call Handling

Processes execute user code on a user stack in the user portion of the process virtual address space.

Each process has a second kernel stack in kernel space (the kernel portion of the address space).

System calls run in kernel mode on the process kernel stack.

System calls run in the process space, so copyin and copyout can access user memory.

The syscall trap handler makes an indirect call through the system call dispatch table to the handler for the specific system call.

Example: Mechanics of an Alpha Syscall Trap

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 process kernel stack
   
   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 (retsyst)
   
   machine restores SP, GP, PC and sets user mode
   
   emerges at user instruction following the callsys
Safe Handling of Syscall Args/Results

1. Decode and validate by-value arguments.
   Process (stub) leaves arguments in registers or on the stack.

2. Validate by-reference (pointer) IN arguments.
   Validate user pointers and copy data into kernel memory with a special safe copy routine, e.g., \textit{copyin}().

   Copy OUT results into user memory with special safe copy routine, e.g., \textit{copyout}().

4. Set up registers with return value(s); return to user space.
   Stub may check to see if syscall failed, possibly raising a user program exception or storing the result in a variable.

Questions About System Call Handling

1. Why do we need special \textit{copyin} and \textit{copyout} routines? validate user addresses before using them

2. What would happen if the kernel did not save all registers?

3. Where should per-process kernel global variables reside? syscall arguments (consider size) and error code

4. What if the \textit{kernel} executes a \textit{callsys} instruction? What if user code executes a \textit{retsy} instruction?

5. How to pass references to kernel objects as arguments or results to/from system calls? pointers? \textbf{No}: use integer \textit{object handles} or \textit{descriptors} (also sometimes called \textit{capabilities}).
Kernel Object Handles

Instances of kernel abstractions may be viewed as “objects” named by protected handles held by processes.

- Handles are obtained by create/open calls, subject to security policies that grant specific rights for each handle.
- Any process with a handle for an object may operate on the object using operations (system calls).
  Specific operations are defined by the object’s type.
- The handle is an integer index to a kernel table.

Faults

Faults are similar to system calls in some respects:

- Faults occur as a result of a process executing an instruction. Fault handlers execute on the process kernel stack; the fault handler may block (sleep) in the kernel.
- The completed fault handler may return to the faulted context.

But faults are different from syscall traps in other respects:

- Syscalls are deliberate, but faults are “accidents”.
  divide-by-zero, dereference invalid pointer, memory page fault
- Not every execution of the faulting instruction results in a fault. may depend on memory state or register contents
Options for Handling a Fault (1)

1. Some faults are handled by “patching things up” and returning to the faulted context.
   
   Example: the kernel may resolve an address fault (virtual memory fault) by installing a new virtual-physical translation.
   
   The fault handler may adjust the saved PC to re-execute the faulting instruction after returning from the fault.

2. Some faults are handled by notifying the process that the fault occurred, so it may recover in its own way.

   Fault handler munges the saved user context (PC, SP) to transfer control to a registered user-mode handler on return from the fault.

   Example: Unix signals or Microsoft NT user-mode Asynchronous Procedure Calls (APCs).

Example: Unix Signals

Unix systems can notify a user program of a fault with a signal. The system defines a fixed set of signal types (e.g., SIGSEGV, SIGBUS, etc.).

A user program may choose to catch some signal types, using a syscall to specify a (user mode) signal handler procedure.

   system passes interrupted context to handler
   handler may munge and/or return to interrupted context

Signals are also used for other forms of asynchronous event notifications.

   E.g., a process may request a SIGALARM after some interval has passed, or signal another process using the kill syscall or command.
Options for Handling a Fault (2)

3. The kernel may handle unrecoverable faults by killing the user process.
   Program fault with no registered user-mode handler?
   Destroy the process, release its resources, maybe write the memory image to a file, and find another ready process/thread to run.
   In Unix this is the default action for many signals (e.g., SEGV).

4. How to handle faults generated by the kernel itself?
   Kernel follows a bogus pointer? Divides by zero? Executes an instruction that is undefined or reserved to user mode?
   These are generally fatal operating system errors resulting in a system crash, e.g., `panic()`!

Thought Questions About Faults

1. How do you suppose `ASSERT` and `panic` are implemented?
2. Unix systems allow you to run a program “under a debugger”. How do you suppose that works?
   If the program crashes, the debugger regains control and allows you to examine/modify its memory and register values!
3. Some operating systems allow remote debugging. A remote machine may examine/modify a crashed system over the network. How?
4. How can a user-mode fault handler recover from a fault? How does it return to the faulted context?
5. How can a debugger restart a program that has stopped, e.g., due to a fault? How are breakpoints implemented?
6. What stack do signal handlers run on?
Architectural Foundations of OS Kernels

- One or more privileged execution modes (e.g., *kernel mode*)
  - protected device control registers
  - privileged instructions to control basic machine functions
- System call *trap* instruction and protected fault handling
  - User processes safely enter the kernel to access shared OS services.
- Virtual memory mapping
  - OS controls virtual-physical translations for each address space.
- Device interrupts to notify the kernel of I/O completion etc.
  - Includes timer hardware and clock interrupts to periodically return control to the kernel as user code executes.
- Atomic instructions for coordination on multiprocessors

A Few More Points about Events

The *machine* may actually be implemented by a combination of hardware and special pre-installed software (firmware).

- PAL (Privileged Architecture Library) on Alpha
  - hides hardware details from even the OS kernel
  - some instructions are really short PAL routines
  - some special “machine registers” are really in PAL scratch memory, not CPU registers

Events illustrate hardware/software tradeoffs:

- how much of the context should be saved on an event or switch, and by whom (hardware, PAL, or OS)
- goal: simple hardware and good performance in common cases