University of BRISTOL

Computer Systems B

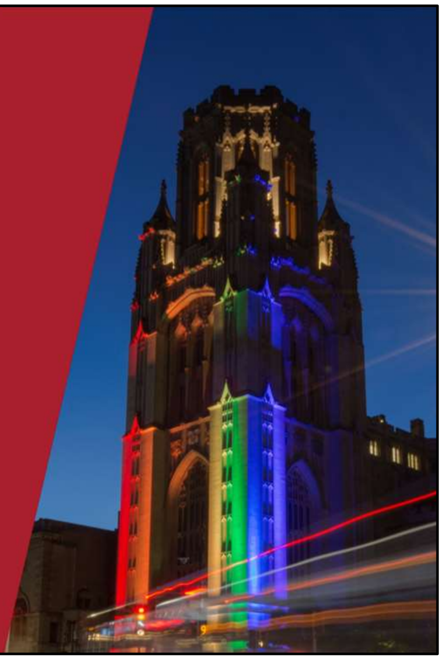
COMS20012

Introduction to Operating Systems and Security

bristol.ac.uk

University of BRISTOL

Trap Handling, Domain Crossing and System Calls



bristol.ac.uk

Terminology

- **Thread switch:** change from one thread of execution to another
  - Does not require a change of protection domain
  - Continue running in the same address space
  - Can change threads in usermode or kernel mode
- **Domain crossing:** change the privilege at which the processor is executing
  - Can change from user to kernel
  - Can change from kernel to user
  - Requires a trap or a return from trap
  - Requires an address space change (user to kernel or kernel to user)
- **Process Switching:** Change execution in one (user) process to execution in another (user) process
  - Requires two domain crossing + a thread switch in the kernel
- **Context Switching:** usage varies
  - Sometimes used for either thread or process switch
  - Sometimes means to mean only process switch
  - Rarely used to mean domain crossing

What can cause a trap?

- The thread request a trap: **System Call**
  - Every system call require domain crossing
- The thread does something bad: **Exception**
  - e.g., accessing invalid memory
- An external event happens: **Interrupt**
  - e.g., timer, disk operation completes, network packet arrived etc.
- **The kernel handles all trap, regardless of the cause!**
  - If the trap occurs during the execution of a user thread domain crossing happens
  - If the kernel is already running there is no domain crossing

bristol.ac.uk

What does the kernel do on a trap?

- The kernel has to find a stack on which to run:
  - If already in the kernel, it uses the same stack
  - If you were in userland, you need to find another stack to run on
  - This means that every userland thread as a corresponding kernel stack
- Before doing anything else, the kernel save the states
  - We will saw how in a few slides
- Figure out what caused the trap
- Do what need to be done

bristol.ac.uk

Process switch

- Change protection domain (user -> kernel)
- Change stack: switch from user stack to kernel stack
- Save execution state (on the kernel stack)
- Do kernel stuff
- Switch (kernel) thread (we saw how in the previous video)
- Restore user stack (belongs to the new process)
- Change protection domain (kernel -> user)

bristol.ac.uk

Handling a Trap MIPS Hardware

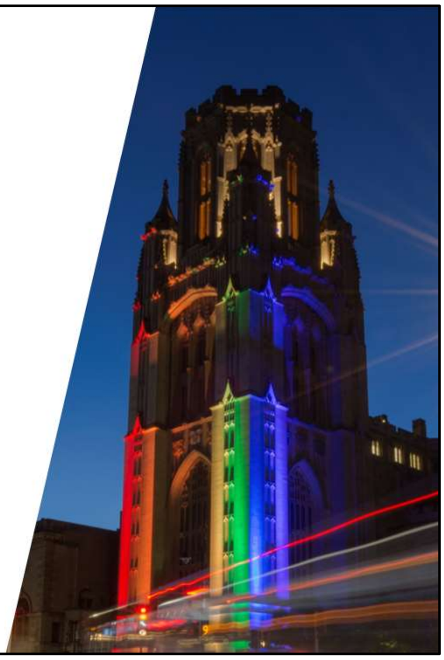
- Upate status register (CP0 $12):
  - turn off interrupts
  - put processor in kernel mode
  - indicate prior state (interrupt on/off, previous mode user/kernel)
- Set cause register (CP0 $13):
  - what trap happened
- Set the exception PC (CP0 $14):
  - The address to return to after the trap has been handled
- Set the PC to the address of the appropriate handler

bristol.ac.uk

University of BRISTOL

- Pause the video
- Open *kern/arch/mips/locore/exception-mips1.S*
- Resume the video

bristol.ac.uk



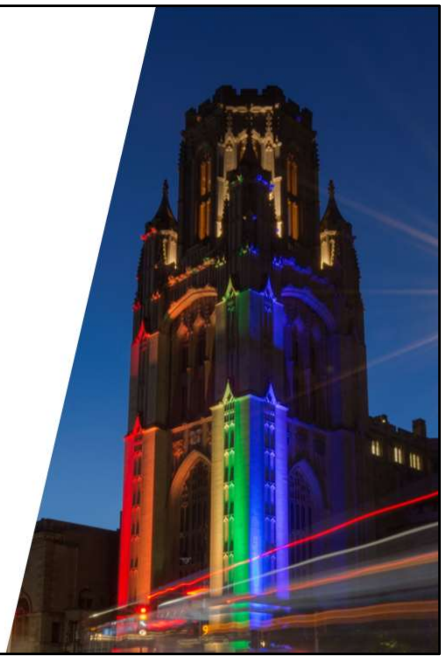
Handling a Trap MIPS Software

- In assembly (see *kern/arch/mips/locore/exception-mips1.S*)
  - Get status register (line 107)
  - Find kernel stack if needed (line 108-123)
  - Allocate trap frame (line 139-140)
  - Save states (line 166-264)
  - Load pointer to kernel global variables (line 270)
  - Call the trap handler function
- From now on things are coded in C

bristol.ac.uk

University of BRISTOL

- Pause the video
- Open *kern/arch/mips/locore/trap.c*
- Resume the video



bristol.ac.uk

Handling a Trap MIPS Software

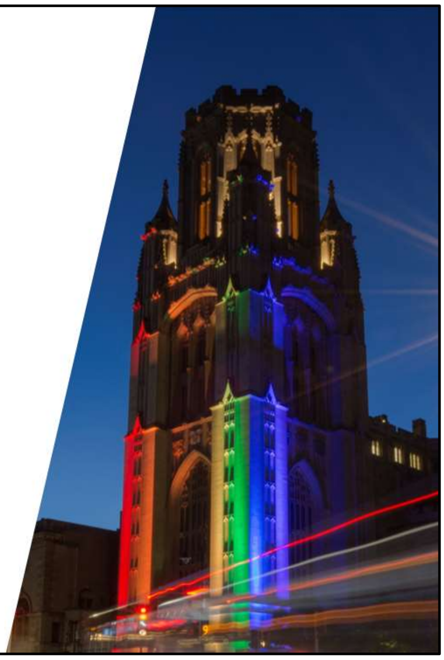
- In C (*kern/arch/mips/locore/trap.c*)
  - Line 126
  - Does error handling (line 134-150)
  - If it is an interrupt handle it (line 153-199)
  - Restore interrupt on/off states (line 212-213)
    - Previously we did not want to be interrupted while potentially handling another interrupt
  - Handle system call (line 215-226)
  - Handle exception cases (line 233-347)

bristol.ac.uk

University of BRISTOL

- Pause the video
- Open *kern/arch/mips/syscall/syscall.c*
- Resume the video

bristol.ac.uk



Handling a system call MIPS Software

- In C (*kern/arch/mips/syscall/syscall.c*)
  - Line 79
  - Figure which system call to execute (line 102-118)
  - Handle errors (line 121-134)
  - Update PC value (line 141)
  - We're done!
- **You will need to modify code here for lab 7.**

bristol.ac.uk

syscall details

- Upon entry in syscall
  - We are in supervisor mode
  - The process states have been saved
- System call details
  - Where did we leave the arguments?
  - How do we know which system call to execute?
  - Where do we return the error?
- Do we need to do anything with the arguments?
  - Where does data referenced by an argument lives?
  - How do we get to it?

bristol.ac.uk

syscall details

- Upon entry in syscall
  - We are in supervisor mode
  - The process states have

First 4 in a0-a3, rest on the stack
(e.g. line 104)

- System call details
  - Where did we leave the arguments?
  - How do we know which system call to execute?
  - Where do we return the error?
- Do we need to do anything with the arguments?
  - Where does data referenced by an argument lives?
  - How do we get to it?

bristol.ac.uk

syscall details

- Upon entry in syscall
  - We are in supervisor mode
  - The process states have been saved
- System call details
  - Where did we leave the arg
  - How do we know which system call to execute?
  - Where do we return the error?
- Do we need to do anything with the arguments?
  - Where does data referenced by an argument lives?
  - How do we get to it?

Syscall number in v0 (see line 89)

bristol.ac.uk

syscall details

- Upon entry in syscall
  - We are in supervisor mode
  - The process states have been saved
- System call detai
  - Where did we le
  - How do we kno
  - Where do we return the error?
- Do we need to do anything with the arguments?
  - Where does data referenced by an argument lives?
  - How do we get to it?

a3 indicate success/failure, v0 contains the error code (see 127 and 128)

bristol.ac.uk

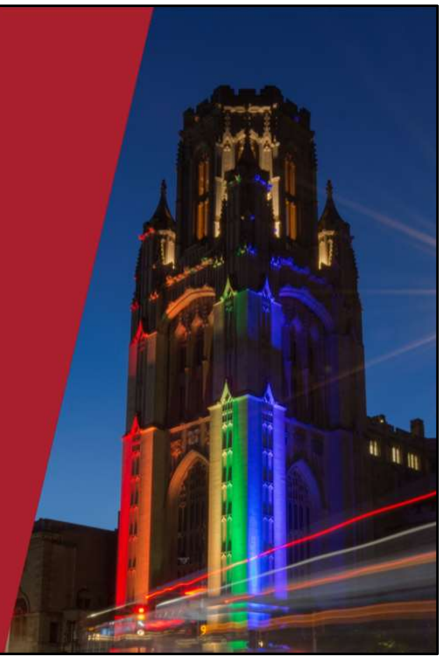
copyin and copyout

- We've seen this in lab 5!
- Process that issues system calls with pointer argument cause two problems:
  - The item references reside in the process address space (i.e. not in the kernel)
  - Those pointers could be bad addresses
- Most kernel have a pair of routines to copy from user or copy to user (*copy_from_user* and *copy_to_user* in Linux)
- In OS/161 they are called copyin and copyout
  - **copyin**: verify that the pointer is valid then copies data from a process address space to the kernel address space
  - **copyout**: verify that the pointer is valid then copies data from the kernel address space to a process address space

bristol.ac.uk

University of BRISTOL

Thank you



bristol.ac.uk