Disclaimer: The contents of this document are scribe notes for The University of Texas at Austin EE360N Fall 2008, Computer Architecture[*]*. The notes capture the class discussion and may contain erroneous and unverified information and comments.*

Lecture 12: Virtual Memory Finale and Interrupts/Exceptions

Lecture #12: October 8, 2008
Lecturer: Derek Chiou
Scribe: Michael Sullivan and Peter Tran

Lecture 12 of EE360N: Computer Architecture summarizes the significance of virtual memory and introduces the concept of interrupts and exceptions. As such, this document is divided into two sections. Section 1 details the key points from the discussion in class on virtual memory. Section 2 goes on to introduce interrupts and exceptions, and shows why they are necessary.

1 Review and Summation of Virtual Memory

The beginning of this lecture wraps up the review of virtual memory. A broad summary of virtual memory systems is given first, in Subsection 1.1. Next, the individual components of the hardware and OS that are required for virtual memory support are covered in 1.2. Finally, the addressing of caches and the interface between the cache and virtual memory are described in 1.3.

1.1 Summary of Virtual Memory Systems

Virtual memory automatically provides the illusion of a large, private, uniform storage space for each process, even on a limited amount of physically addressed memory. The virtual memory space available to every process *looks* the same every time the process is run, irrespective of the amount of memory available, or the program's actual placement in main memory. Virtual memory can be broken up into three (barely overlapping) components.

[*]Copyright 2008 Michael Sullivan and Peter Tran and Derek Chiou, all rights reserved. This work may be reproduced and redistributed, in whole or in part, without prior written permission, provided all copies cite the original source of the document including the names of the copyright holders and "The University of Texas at Austin EE360N Fall 2008, Computer Architecture".

1. **Address Translation** Translates a virtual address to a physical memory address.

   Address translation:

   - Provides the ability for **dynamic relocation**, where the OS may move pages of a process in main memory while the process is running without affecting its functional behavior. Dynamic relocation may, however, affect the performance of a running process.

2. **Demand Paging** Provides processes with the (perceived) capacity of secondary memory at the speed of primary memory.

   Demand paging:

   - Is handled by the operating system.
   - Can be seen as analogous to caching disk space in main memory.
   - Involves the automatic and transparent swapping of program data between primary and secondary storage.

3. **Protection** Supports multiple users on the same system, each with his own private address space.

   Protection:

   - Multiple running processes may share data in a protected fashion.
   - Provides a strong level of security and privacy to users' data, built in at the system level.

1.2 Required Support for Virtual Memory

In order to implement virtual memory, a system needs cooperation between the operating system and CPU. Although there is no one way to implement virtual memory, there are common elements to almost all existing virtual memory implementations. Several structures and elements which are required for efficient virtual memory support follow.

1. **Page Table** A data structure that is indexed by the page number, and contains the physical address of a block.

   Some common page table properties:

   - The location of a particular page (in memory or on disk) is often described using a pair of *presence bits*. The location can take on one of three states:
     (a) **present** indicates that the page resides in a primary memory address
     (a) **on disk** indicates that the page resides in secondary memory
     (a) **missing** indicates that memory for the page has not yet been allocated

- *Protection bits* are often used to indicate how this page can be accessed (read/write/execute).
- Several *usage bits* may be used to indicate when a page was last accessed, and whether its contents were changed in main memory.
  - Bits that indicate when pages were accessed are primarily used to implement a pseudo-LRU page replacement algorithm.
  - Some operating systems have an "inclusive" view of the memory, where contents of the program data on the disk are the same as the pages stored in memory. Such a case is analogous to a *write-through* cache, and may not require dirty bits.
  - Most modern operating systems (such as Windows XP and Linux) have an "exclusive view" of memory, where the main memory functions like a *write-back* cache, and the contents of its pages must be written back to disk before they are replaced. In this case, dirty bits are often used to indicate which pages have been changed while they resided in memory.

2. **Page Faults** Throw a synchronous exception in case of a missing page or protection violation. The concept of a *synchronous exception* will be covered in Section 2.

3. **Fast Translation Mechanism** A hardware structure that assists in the fast translation of virtual addresses to physical addresses. A common example is a translation lookaside buffer (TLB), that caches recently accessed pages in the CPU for quick access.

4. **Swapping Store** An area of secondary memory (disk) that is used to store pages that are swapped out of main memory. The swapping store is necessary in order to support demand paging, and give the illusion of a large virtual memory space, irrespective of the amount of physical memory in a machine.

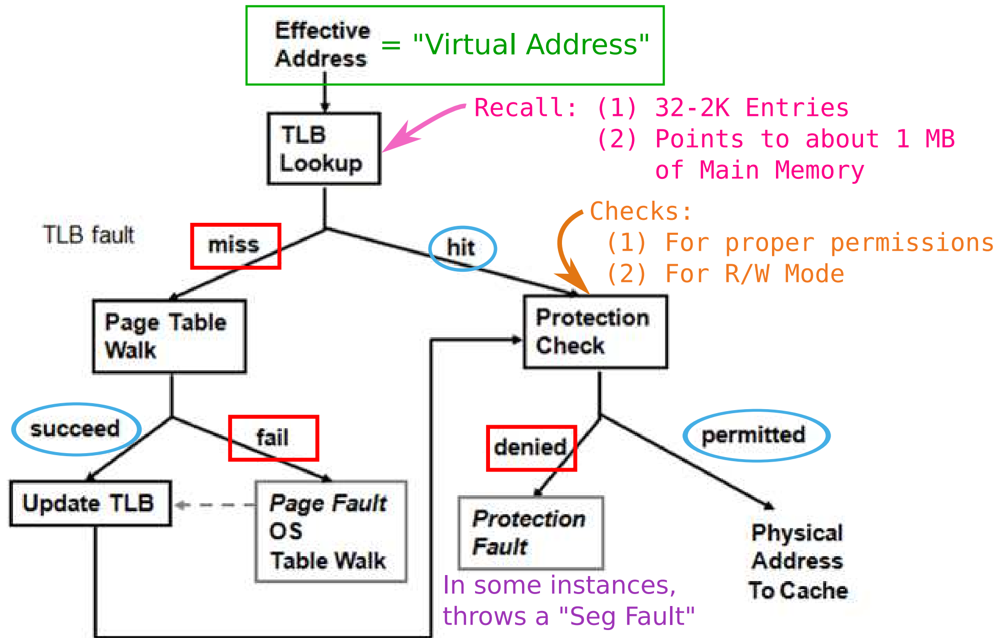
5. **Page Fault Handler** Code, managed by the operating system, that handles page faults. Some required properties of the way that operating systems handle page faults:

   - The operating system must keep two separate lists of the free pages inside of main memory and on the disk. The two lists are independent, and must be mapped separately.

1.3 Virtual Memory Flow Chart

An overview of the translation process can be seen in Figure 1. It can be seen that the effective (or "virtual") address is mapped to a physical address, assuming that the running process has permission to access the page in memory.

Figure 1: Virtual Memory Flow Chart

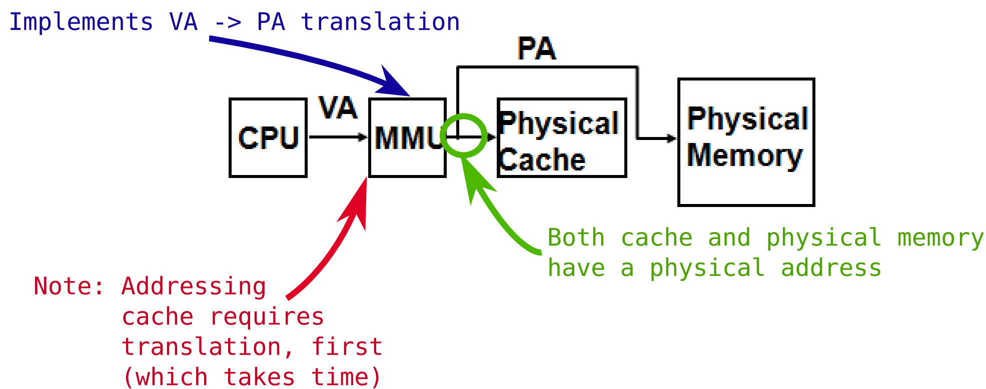


The TLB is used to speed up repeated requests to the same pages, and exploits temporal locality in the instruction stream. While the TLB is much quicker than doing a page table walk, its size is limited by the fact that it is often associative and resides on the CPU. For this reason, common table lookaside buffers can only index about 1MB of pages in memory at a time. Upon a TLB fault, the system throws a page fault. The operating system must then perform an expensive page table walk, where the page tables are traversed to translate the virtual address. The contents of the TLB are updated following a successful page table walk, and the translation process is retried as if the TLB fault had never happened.

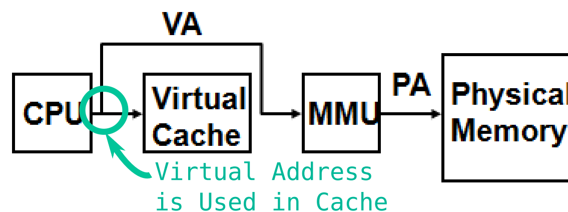
1.4 Caches and Virtual Memory

When combining a cache hierarchy with a virtual memory system, one must decide how the two interact with one another. Contents are inserted into the cache, and are tagged, according to their memory address. However, it is unclear (upon first glance) whether it is better to insert values into the cache using virtual addresses or physical addresses. An illustration of *physical* and *virtual* caches can be found in Figure 1.4.

Figure 2: Virtual and Physical Cache Addressing



(a) Physical Cache



(b) Virtual Cache

When caching values with their physical addresses, the CPU must first translate from the virtual to the physical address. The translation process, together with the handling of page faults if a page is not resident in memory (see Figure 1 for details) must be completed before any access to the cache can be made. This added latency will result in a performance hit, and may reduce the clock rate of the system.

Virtual caches do *not* require that the virtual address be translated before accessing the cache. Rather, the virtual address is used directly to index the cache contents. While this results in faster access for memory residing in the cache, there are several inherent problems with virtually addressed caches, which can be found below.

1. **Guaranteeing Uniqueness** All userspace programs operate using the same virtual address space. Therefore, identical virtual addresses (in use by different programs) may point to different areas of physical memory. Unless some solution is implemented, switching between multiple processes with virtually-tagged cache contents may violate the correctness of CPU operation. Two solutions to this problem follow:

   (a) The most obvious way to guarantee unique cache contents is to flush the cache upon every context switch.

   - However, this approach may result in a performance loss, since a program will always have to repopulate the cache once swapped in. Thus, a newly swapped-in process cannot exploit locality in any cache blocks that would have otherwise still be in the cache.
   - Also, the act of flushing the cache may be expensive, since every block needs to be invalidated. Using this technique to guarantee uniqueness, the cost of flushing the cache will be incurred at every context switch.

   (b) Another way to guarantee that each process accesses only its entries into the cache is to tag every cache entry with a unique process ID (PID). While this approach has an associated overhead (to store every PID tag), it avoids the need to flush the cache upon every context switch.

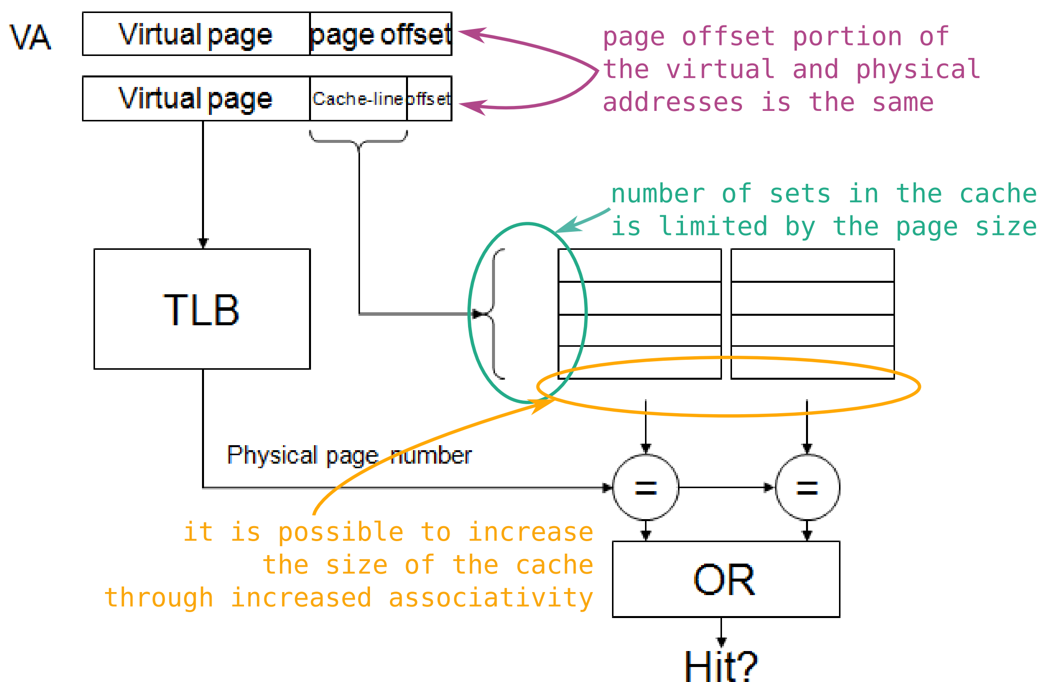
2. **Aliasing** It is possible for programs to use multiple different virtual addresses for the same physical address. Unhandled, these duplicate virtual addresses (called *aliases*) could create multiple copies of the same memory location in a virtual cache. Without some method of forcing coherence upon the aliases in a virtual cache, different programs could maintain different values for the same memory location.

   Hardware solutions to the aliasing problem (called *antialiasing*) exist, and guarantee that every cache block has a unique physical address. Thus, only one cache block exists for a set of aliases pointing to the same memory location. Typically, antialiasing schemes operate by checking all possible virtual addresses on a cache miss in order to prevent multiple copies of the same physical data from being brought into the cache. Antialiasing hardware can be fairly complex, which may result in increased area, power usage, and may impact the performance of the cache.

1.4.1 Is the "Best of Both Worlds" Possible?

It is possible to exploit some of the benefits of both physical and virtual addressing by using a *virtually-addressed, physically-tagged cache*. This possibility arises due to the fact that part of the page offset is identical in both the virtual and physical addresses corresponding to a location in memory. Exploiting this fact, it is possible to use the offset bits to index the cache, and therefore to begin physically indexing the cache before virtual address translation has finished.

Figure 3: A Virtually-Addressed, Physically-Tagged Cache



A limitation of this approach is that the number of sets that are possible in the cache is fixed by the number of offset bits in the virtual and physical addresses. In turn, the number of offset bits is determined by the page size of the system. Therefore, a virtually-addressed physically-tagged cache may limit the possible size of a cache. Recall, however, that the total size of a cache is affected not only by the number of sets, but also by the associativity of the cache. Thus, in practice, a larger cache size may be achieved with a virtually-addressed, physically-tagged cache by increasing the associativity of the cache.

There is also the possibility of "giving" one or more of the virtual page address bits to the offset field. However, this requires a restriction be placed upon the virtual address to physical mapping in order to prevent multiple virtual addresses with different targets from hitting on the same cache block. For example, if one extra bit is required, this restriction will constrain the OS to allocate even pages to even frames, and odd pages to odd frames, which keeps the least significant bit of the VPN and PFN the same. This restriction requires operating system support.

2 Interrupts & Exceptions

Interrupts and exceptions are, for this class, defined as follows[1].

- **Interrupt:** an event external to the processor that alters the normal flow of instructions. Examples include:
  - Hard disk accesses
  - Network events
  - Keyboard input
- **Exception:** an event internal to the processor that alters the normal flow of instructions. Examples include:
  - Page faults
  - Arithmetic exceptions, such as division by 0

2.1 Common Terms Associated with Interrupts & Exceptions

Some common terms associated with interrupts and exceptions follow.

- **Synchronous Events:** occur in the same place in an instruction stream every time
  - Essentially synonymous with exceptions
- **Asynchronous Events:** occur independent of the location in the instruction stream
  - Essentially synonymous with interrupts
- **Maskable Events:** may be enabled or disabled by the operating system. An example:
  - Misaligned accesses may be maskable in order to cause an exception or not
- **Unmaskable Events:** cannot be disabled, and always raise an exception
- **User Requested Events:** exceptions which are requested by userspace programs. Examples include:
  - I/O and HALT functionality, called through TRAP instructions

[1] These definitions are not universally accepted. White papers for different architectures will use different nomenclature to talk about equivalent concepts. See Section 3

2.2 Why are Interrupts and Exceptions Needed?

Many events occur simultaneously in a computer system. Input may come from external devices, such as the hard disk or network connection, and the timing of this input is highly unpredictable. Because most current processors have a single program counter, handling asynchronous, unpredictable input without extra hardware raises a problem. In addition, errors in execution (such as an arithmetic divide by 0 or a page fault) may interrupt the normal flow of program execution.

While input from external devices and periodic errors in execution must be handled, they are relatively rare. Therefore, designing specialized hardware to handle every single type of event that may occur in a system would be a waste of resources. Rather, interrupts and exceptions offer a way to handle unpredictable and uncommon events in software, which offers more flexibility and cost savings over hardware solutions. Interrupts and exceptions switch the running program (in a transparent manner) to software "handlers", that perform appropriate actions to resolve an event. More details on interrupt and exception handlers follow.

2.2.1 Handlers: Interrupt and Exception "Subroutines"

A **handler** is a specific subroutine that is called after an event has occurred during the instruction stream. Examples include:

- Interrupt Handlers
- Exception Handlers
- Page Fault Handlers

After a handler completes its tasks, the program must be able to restart the original instruction stream in order to continue execution. This requires that the PC of the interrupted instruction stream be saved by hardware for use after the handler is done. The need to store the interrupted PC in a hardware location is due to the fact that the PC will be overwritten once the processor jumps to the software handler.

2.2.2 When is a handler called and where is the PC restored once the handler is done?

Because of the relative priority differences between exceptions (which must be handled immediately) and interrupts (whose execution may be delayed), this answer depends on the type of event. In general, the following rules apply to interrupts and exceptions.

- **Interrupts:** Servicing of interrupts may be delayed until it is convenient for the system. For this reason:
  - The instruction is allowed to complete even when an interrupt is detected

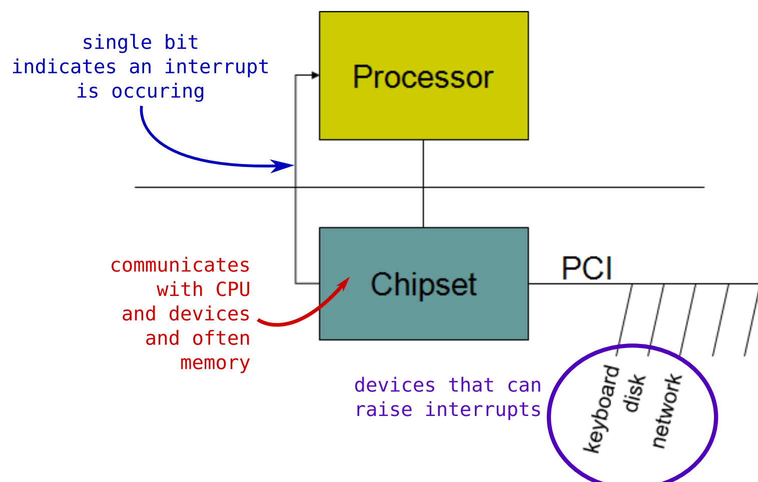
    * The interrupt handler is executed after the instruction is complete
  - Interrupt handler needs to return to the following instruction after the handler is finished executing

- **Exceptions:** Exceptions signal an internal failure, and so continuing to execute makes no sense. In the case of an exception:
  - The instruction where the exception occurs can not continue without help
    * The exception handler must to be called before the instruction completes
  - Exception handler needs to return to the instruction where the exception occurred and the entire instruction needs to be executed again.

2.3 Interrupts in a Processor

Interrupts are events that occur external to the processor, yet require some immediate action to be taken by the CPU. For this reason, there needs to be some channel of communication between the system and the processor in order for the CPU to detect and service interrupt requests. Figure 4 shows a diagram depicting the most common way of communicating between external devices and the CPU. Devices (which are located in PCI slots in this example) communicate with the CPU through a chip on the motherboard. When an interrupt is raised, a single wire indicates to the CPU that an external event has occurred.

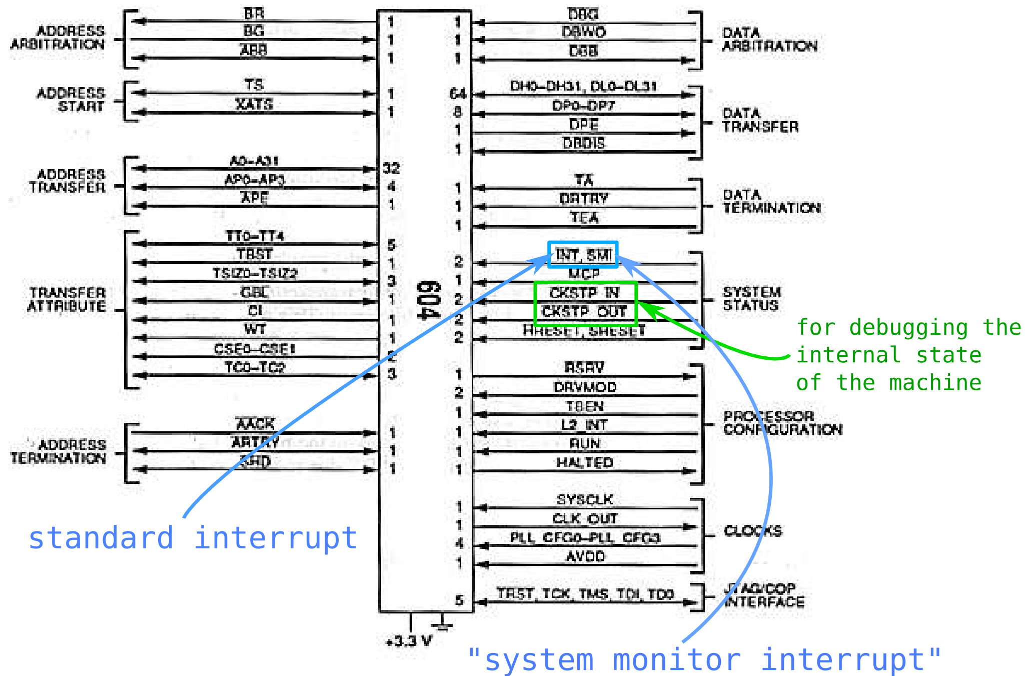
Figure 4: System I/O Interrupts



2.3.1 Interrupts in the PowerPC 604

As a specific example of how a real-world CPU handles interrupts and exceptions, we consider the PowerPC 604. Figure 5 shows the PowerPC 604 pin out, including those lines which are used for general and specialized interrupts. These interrupt lines signal to the processor that an event has occurred, and may be used for input from external devices or for precise debugging of the chip itself during runtime.

Figure 5: Interrupts in the PowerPC 604



2.3.2 Types of Handlers

- **DSI/ISI Handler:** Several exceptions may be triggered by memory accesses. The most common is an exception raised upon a page fault. The page fault handler:
  - Allocates a page frame
  - Reads data from disk if necessary
  - Swaps page back
  - Updates page table
- **External Interrupt:** External interrupts occur when another device in the system requires assistance from or communication with the CPU. A depiction of the way that external interrupts trigger the CPU can be found in Figure 4. In

order to discover which device triggered the interrupt, an interrupt handler may:

- Poll all possible sources of the interrupt
  - Keyboard
  - Disk
  - Network
- Copy the source of the interrupt into an appropriate buffer location (in the chipset)

- **Decrementer (Timer):** Used by the operating system for context switches.
  - Switches to new process
  - Does a Least Recently Used (LRU) bit flipping for page retrieval

- **Program:** A debugger may replace program instructions with TRAPs
  - Uses the saved PC to determine location within the program

2.4 Multiple Interrupts/Exceptions

There is always a chance that multiple interrupts or exceptions will occur simultaneously. A problem arises when deciding which of the handlers should be called first. The simplest way to resolve this issue is to assign priorities to every interrupt and exception. Then the handler of the interrupt or exception with the highest priority is called first. This method insures that the most important issues are dealt with first in order for the program to operate with minimum problems.

If only one signal is used to signify an interrupt or exception, the processor does not know which of the interrupts or exceptions have occurred. The processor will need to poll state to determine which interrupt or exception has occurred and then polls the handlers in priority order. A more efficient way of determining which handler to call is to vector to the appropriate handler based on the interrupt/exception.

2.4.1 Vectoring to Interrupts/Exception Handlers

An efficient way to service interrupts and exceptions is to use their unique numbers to index into a interrupt table, which contains jump statements to the handler subroutines. The specific steps in the vectoring process follow.

1. Handling an interrupt or exception first saves the PC of the interrupted stream into a special register. Other altered state must be saved and restored by the the handler being called.

2. The interrupt/exception number is then used as an index into the interrupt/exception vector. This creates a PC of the location that needs to be jumped to.

3. A jump is then made to the location in the interrupt/exception vector. Another jump usually occurs to the interrupt/exception handling code. This allows for the interrupt/exception vector to remain small.

4. At the jump to the handler, supervisor mode is enabled. This mode has special permissions that are needed to fix problems, such as the ability to load pages. Usually priority for exception handling is inherited from the original program.

For a real-world example of vectoring to interrupt and exception handlers, we once again consider the PowerPC604 processor. Table 1 shows the types of interrupts and exceptions which are handled by the PowerPC 604[2]. The second column shows the vector offset (into the interrupt vector) for the handler of each type of interrupt and exception.

Table 1: PowerPC 604 Exceptions & Interrupts

Exception Type	Vector Offset	Conditions
System Reset	0x0100	hard reset or soft reset asserted
Machine check	0x0200	TEA/MCP, problems in memory, can checkstop
DSI	0x0300	Data memory access
ISI	0x0400	Instruction memory access
External Interrupt	0x0500	some external interrupt
Alignment	0x0600	data access out of alignment
Program	0x0700	floating point, illegal, privileged, trap instructions
Decrementer	0x0900	a counter's MSB changes from 0 to 1
System call	0x0C00	when an system call instruction is executed
Trace	0x0D00	any instruction OR branch completes
Perf Monitor	0x0F00	when a performance counter goes negative
Inst Add Break	0x1300	inst address matches a special reg
System Manage	0x1400	SMI asserted

[2]For our purposes, Table 1 contains both interrupts and exceptions. IBM does not follow the same naming conventions. See Section 3 for details.

Table 2 shows the contents of the PowerPC 604 interrupt vector. Each entry corresponds to a jump instruction to a subroutine for a specific interrupt or exception handler.

Table 2: PowerPC 604 Interrupt Vector

Vector Index	Handler
0x00000100	System reset handler
0x00000200	Machine check handler
0x00000300	DSI handler
0x00000400	ISI handler
0x00000500	External interrupt handler
0x00000600	Alignment handler
0x00000700	Program handler
...	...

2.4.2 Nested Exceptions

A nested exception is when an interrupt or exception occurs when another is already active. Priority is maintained throughout the course of a nested exception to insure that the more important interrupts or exceptions are dealt with first. Handling nested exceptions is like calling a subroutine within another subroutine. The handlers will need to save the appropriate state in order for the program to return to the appropriate handler or the original instruction stream after the completion of each handler.

3 Definitions in Industry

While the concepts of interrupts and exceptions are pervasive across different architectures and platforms, the definition of the terms "interrupt" and "exception" vary widely. Companies in industry use different labels to describe interrupts and exceptions. The nomenclature used by several well-known companies follows.

- **Intel** *Local exceptions* are generated internally within the processor, and share a similar definition to our "exceptions". *External exceptions* are generated outside of the processor (by I/O devices) and are received at exception pins. External exceptions are also called *interrupts*.
- **IBM PowerPC Architecture** All errors, external signals, and unusual conditions are called *interrupts*.
- **IBM PowerPC 604 and Motorola 68K** All errors, external signals, and unusual conditions are called *exceptions*.
- **DEC VAX** Definitions of "interrupt" and "exception" similar to our own. Signals generated by external devices are classified as *interrupts*. Faults and events that must be executed immediately are called *exceptions*.