AOSV SBOB Lecture 3

Spring 2019

1 Front-Side Bus

Initially up to 2004 cores and processors where connected using the front side bus, that was the bus telling how fast multiple chips were able to load data from memory, since all components rely on this bus to read the memory. Of course there were some physical limitations and FSB became a bottolneck to the system' speed.

2 Dual Independent Buses

It was then replaced (2005) Dual Independent Buses. There where subset of processors which where using a shared bus and multiple busses started to appear. **NB** if we drop the common bus we must find the way to forward the chache coherence transaction to one bus to the other to allow CC and MMU to exchange transactions. An additional component was added: Snoop filter, an adaptor which forward messages from one bus to the other depending on the actual destintion of the transaction (it is possible only reling on directory based protocolls).

3 Dedicated High-Speed Interconnects

In 2007 Dedicated High-Speed Interconnects were introduced. They started to provide a dedicated bus from each processor to the chipset implementing the interface to memory. We are increasing the complexity of the snoop filter.

4 QuickPath Interconnects

On current architecture we have QuickPath Interconnect (an intel implementation in the pic. of slide pack 2 slide 105). It is intresting because it implements a shared memory architecture. Each processor has its own memory interface and its own bus that targets some phisycal memory, in addition each processor has a dedicated bus that directly connect it to the other processors. The total amount of avaliable memory is partitioned across multiple processors: physiscal memory is not fully reachable by any CPU. The situation now is: one dedicated bus from one processor to the other, but each processor has multiple cores competing to use the bus. We still need an arbiter which allows each core to access the QuickPath Interconnects: in each processor in then added a crossbar router. The idea is that we have multiple lines going out from the cores into Crossbar Router and a single output line connecting the router to the QuickPath Interconnects. An arbiter allows only a single core per time to access to the output line serializing the accesses, all is done with a single transistor. Since we are talking about routers we can imagine that a complete ISO stack is implemented in hardware with the following layers between processors:

1. Protocol
2. Transport
3. Routing
4. Link
5. Physical

any time that one core wants to send some kind of message an implementation of the above levels must be provided. A message is split into packets and forwarded trought the network according to some routing strategy. This is a **Distributed shared memory architecture**.

5 Dealing with locks

about split locks Intel documentation says:

> For the P6 and more recent processor families, if the area of memory being locked during a LOCK operation is cached in the processor that is performing the LOCK operation as write-back

> memory and is completely contained in a cache line, the processor may not assert the LOCK# signal on the bus. Instead, it will modify the memory location internally and allow it's cache coherency mechanism to ensure that the operation is carried out atomically. This operation is called "cache locking." The cache coherency mechanism automatically prevents two or more processors that have cached the same area of memory from simultaneously modifying data in that area.

A very important thing contained in this definition is: if we declare a lock in memory and the address of that variable is such that it can not be contained in a single cache line, in order to perform some read, modify, write on that lock requires to read two cache line, but is something that CacheCoherenceProtocols do not offer because we have one state machine for each line. To atomically update two cache lines, moderns cpus will start some kind of agreement upon Cache Controllers (CC). There is one controller in the hardware system, the Quiesce Master, that is a component of the distributed hardware architecture thaht eventually ask to all the cpu's cores to stop their activities on the memory. This is done in this way (slide pack 2 slide 109 for an example of execution of the protocol): processor that is going to perform some r-m-w (read modify write) instruction asks to the Quiesce Master to stop the actions on all cache controllers from all the cores because we can not grab the exclusive state for multiple cache lines. The master will then send messages asking to stop all the activities and is waiting to ack from all controllers. Then the LockPhase can start: multiple cache lines are updated atomically. At the end of the protocoll the Requestor is unlocking the Cache Lines and the Master is broadcasting to all cpu cores a message telling that the transaction is completed so that CC can continue to perform their actions on cache lines according to the scheme defined before (prev. lecture). Even if the protocoll is very intrusive there is no other way to manage this situation ensuring consistency. We can imagine that the kernell wants to avoid this situation (lock on multiple lines) as much as possible from a performance point of view, and we will see lots of facilities specifing that important variables must not be splitted on multiple lines.

6 UMA vs NUMA

Phisical mem partitioned per core is the implementation of what is called NonUniformMemoryArchitecture (NUMA). We can see principal difference between UMA and NUMA is that numa has multiple busses connected with a router and multiple cores connected to a specific bus and accessing only

to a portion of the available physical memory. If one core need to access memory that it is not able to address by itself then it will issue a request to the core that can access the required memory. Memory content is putted into a cache and sent to the cache of the core that has requested the memory. If the memory should be written back, the cache line is transparently sent back by the cache controller to the other cache controller that will write back the memory. Concept of close and distance memory are introduced since latency is now variable, depending on if we can or we can not directly access to that part of memory. In some acrchitecture is possible to have Numa Nodes or Numa Zones, that are set of processors (or cores) grouped together with a common dedicated Memory Controller wich allows them to reach a portion of memory in a uniform way. In some arch. there is not a direct connection between numa nodes, so to access some portion of memory request must be forwarded towards numa nodes to reach the destination of the request (high latency operation). Implementing an OS two important notions are required about numa arch:

1. addresses of physical memory accessible from each single numa node
2. scheduling a process i must carefully decide on which proccess it must be scheduled in order to decrease latency of memory accesses.

Linux suppport numa starting from kernel version 2.6. There are facilities implemented as system call that allows a process to give hints about its memory demand and its scheduling requirements, so that we can ask the system to use specific policies to optimize numa system. libnuma is a user space support library that gives an organized access to all facilities supporting numa nodes. It is an abstract interface, thare are some heder that can be used in our programs and a runtime library thath can belinked in the application. A particular comand offred by libnuma is numactl, it allows to retrive information about the hardware organization in the machine and also to wrap all scheduling and and memory requirements of our proccess. Example of libnuma facilities are at slides pack 2 slide 115.

7 Slide pack 3

8 Booting

At different stages of execution hardware is configured in different ways and e have different support. The OS itself can not rely on its facilities at any time. Once we run the machine (press the power button) we can only run BIOS

os UEFI. BIOS itself provides services to let the machine at early stage to dialog with hardware. Only after the kernel inizialization it is able to directly interact with the hardware.

9 First step: BIOS

BIOS' role is to bring the machine into a state wich is uniform and usable by the bootloader stage1. At the very beginning we have Power Sequencing State: provide specific voltage to each component in a particular order done by specific hardware embedded on the motherboard. Then we generates clocks from a cuple of clock sources (note that in our macines each component may require different clock frequencies). Once clocks are initialized for each component and they are synchronized, the PSC (power sequencing control) will deassert the resert pin, that tells cpu to start executing instructions. At this point, the system is in a very basic state: Caches are disabled, The Memory Management Unit (MMU) is disabled, The CPU is executing in Real Mode (8086-compatible), Only one core can run actual code. BIOS' goal is to swich from this state to a state in which some code is executable.

9.1 8086 Compatibility: Real Mode

In 8086 compatible mode (Real Mode) we have one memory model called segmented memory. We have a linear addres space divied into segment. Different segment can have different priviledges and they were implemented to keep track of different portion of the programs on 8086 machines. Segments are identified with an ID that is later associated with the base addres of that segment. Any address in segment A is identified specifying ID,OFFSET. (segments where an initial attempt to differenciate among code data and the stack). On 8086, to make it easy to retrieve segment selectors quickly, the processor provides segmentation registers whose only purpose is to hold Segment Selectors; these registers are called cs, ss, ds, es (fs, and gs where added on 80386). Although there are only six of them, a program can reuse the same segmentation register for different purposes by saving its content in memory and then restoring it later. Three of the six segmentation registers have specific purposes:

- **cs:**The code segment register, which points to a segment containing program instructions
- **ss:**The stack segment register, which points to a segment containing the current program stack

- **ds:** The data segment register, which points to a segment containing global and static data

- **es, fs, gs:** The remaining three segmentation registers are general purpose and may refer to arbitrary data segments. Now those extra regsters are foundamental to implement some services and library: fs implements a thread local storage service, gs is used from linux kernel to manage "per cpu variables".

The segment identifier is a 16-bit field called the Segment Selector, while the offset is a 32-bit field. (in 8086 arch. segmentation was how the MMU actually worked)

Programs generates logical addresses. Segmentation Unit maps those logical addresses into physical addresses, that at the 8086 time where given to Northbridge chip (implementing the front side bus giving access to memory). This was the MMU at the 8086 time. Slide pack 3 slide 10 for a picture. Referring to the image here there are some definitions (from "Understanding the Linux Kernell"):

- Logical address: Included in the machine language instructions to specify the address of an operand or of an instruction. This type of address embodies the well-known 80 x 86 segmented architecture that forces MS-DOS and Windows programmers to divide their programs into segments . Each logical address consists of a segment and an offset (or displacement) that denotes the distance from the start of the segment to the actual address.

- Linear address (also known as virtual address): A single 32-bit unsigned integer that can be used to address up to 4 GB that is, up to 4,294,967,296 memory cells. Linear addresses are usually represented in hexadecimal notation; their values range from 0x00000000 to 0xffffffff. (not used in Real Mode)

- Physical address: Used to address memory cells in memory chips. They correspond to the electrical signals sent along the address pins of the microprocessor to the memory bus. Physical addresses are represented as 32-bit or 36-bit unsigned integers. The Memory Management Unit (MMU) transforms a logical address into a linear address by means of a hardware circuit called a segmentation unit ; subsequently, a second hardware circuit called a paging unit transforms the linear address into a physical address

Today we still have segmentation but implicitly (a jump instruction uses CS a push instruction uses SS). We can load segment values with a simple "mov" instruction. NOTE: only OS can perform this kind of operatin, if we try to do that in user space the system will crash or return an error. cs can not be updated explicitly, mov is not allowed on it. it is uploaded one calling a special "jmp" or "call" instruction specifing the address to put into cs.

9.2 Memory Management in Real Mode

x86 Real Mode gave us the possibility to use 16-bit instruction execution mode, 20-bit segmented memory address space, that is 2^20 possible addresses, ie 1 MB of total addressable memory. A Real Mode address is composed of a seg segment and an off offset; the corresponding physical address is given by $seg * 16 + off$ (the multiplication gives a simple shift of one position to the left in exadecimal notation). If for example we have a jump instruction that whant to reach address 0x6025 and the CS contains 0x1000 the result will be:
segment 0x1000
Memory address (offset from the current instruction) 0x6025
$seg * 16 + off = 0x10000 + 0x06025 = 0x16025$ (phisical address).
Note that a segment can be reched eithr with base addres or with another base + offset.

If we define a base addres as $< ID : OFFSET >$ of another segment we can reach a location with two different segment: programmer must be careful to give a perfect separation to segments.

There are some addresses that can be represented with segmentation but can not be handled by memory in this first stage. The 16-bit segment selector in the segment register is interpreted as the most significant 16 bits of a linear 20-bit address, called a segment address, of which the remaining four least significant bits are all zeros. The segment address is always added to a 16-bit offset in the instruction to yield a linear address, which is the same as physical address in this mode. Because of the way the segment address and offset are added, a single linear address can be mapped to up to $2^12 =$ 4096 distinct segment:offset pairs. Since there is no protection or privilege limitation in real mode, even if a segment could be defined to be smaller than 64 KB, it would still be entirely up to the programs to coordinate and keep within the bounds of their segments, as any program can always access any memory (since it can arbitrarily set segment selectors to change segment addresses with absolutely no supervision). Therefore, real mode can just as well be imagined as having a variable length for each segment, in the range 1 to 65,536 bytes, that is just not enforced by the CPU.

10 Back to BIOS

Once we deassert the resect line in the intel cpu the program counter does not start at zero but at the resect vector, configured such that the segment and offset form a specific addres that is F000:FFF0 (16 bytes before the end of avaliable the memory). At this address we find the first fetched instruction, that is generally on a Read Only piece of Memory. It is convenitionally fixed at F000:FFF0 and is a jump instruction to the first instruction of bios code. From now on the BIOS try to bring the organization of the machine into a state that is as uniform as possible to load the actual OS.
The first thing the bios actually does is to memory map a portion of the routins implementing graphic cards (each graphic card provides the assembly code within the card that allows to write characters on the screen). After that POST (Power-On Self-Test) phase can start to simply check correctness of devices initialization, it is strictly bios dependent. BIOS loads its own code into memory to give faster access to its code and it may try to load the OS from a memorized location according to device booting order. Next step is to try to load the stage 1 bootloader (only if not using UEFI). It will load the MBR (Master Boot Record, 512 bytes) into memory using a specified boot order and a fixed address: 0000:7c00. The controll to bootloader is then given thanks to a longjump instruction (ljump $0x0000 $0x7c00). After BIOS has executed its routines the only memory avaliable is 640Kb that must be sufficient to load the entire system. The MBR contains a mixture of data-structures (signature to validate bootsector, partition table) and a tiny amount of executable code (MBR bootstrap) that allows us to initialize bootloader. It may contain an addictional data-structure called BIOS Parameter Blcok (BPB) describing the physical geometry of the devices; problems: space is reduced. We can solve the problem simply using another jump instruction that allows us to skip the datastructures in the MBR. Here we can execute some code: 1) we disable interrupst because we can not handle them yet 2) we set all segments (ds es gs fs ss) to 0, but not cs because we can not copy a value into cs (note that the activation od MBR already sets cs to zero). Segmentation register will be set to zero to the while execution time starting from now, because segmentation in no longer used 3) sets up the Real Mode stack, loads the second part of the bootloader into RAM starting from a specified address and jumps into it. Next step are

- to load stage 2 bootloader (thanks to bios facilities that allowa to load data from storage to memory);
- to Enable address A20;

- Switch to 32-bit protected mode;
- Setup a stack;
- Load the kernel.

11 About A20

The 80286 could address up to 16 MB of system memory in protected mode. However, the CPU was supposed to emulate an 8086's behavior in real mode, its startup mode, so that it could run operating systems and programs that were not written for protected mode. The 80286 did not force the A20 line to zero in real mode, however. Therefore, the combination F800:8000 would no longer point to the physical address 0x00000000, but to the correct address 0x00100000. As a result, some DOS programs would no longer work. To remain compatible with such programs, IBM decided to correct the problem on the motherboard. That was accomplished by inserting a logic gate on the A20 line between the processor and system bus, which got named Gate-A20. Gate-A20 can be enabled or disabled by software to allow or prevent the address bus from receiving a signal from A20. It is set to non-passing for the execution of older programs that rely on the wrap-around. At boot time, the BIOS first enables Gate-A20 when it counts and tests all of the system memory, and then disables it before transferring control to the operating system. Originally, the logic gate was a gate connected to the Intel 8042 keyboard controller. Controlling it was a relatively slow process. Other methods have since been added to allow more efficient multitasking of programs that require this wrap-around with programs that access all of the system memory. There are multiple methods to control the A20 line.
(TAKEN FROM WIKI "A20 LINE" TO MAKE STUFF MORE CLEAR AND SHORT)

12 Protected Mode

Now we need to enable line A20 and switch to Protected mode to enable paging. CPUs have lots of registers. Some CR (Control Registers) are fondamental to tell in wich mode we are running. CR0 is the first introduced, it allows programmers to interact with the hardwer in order to activate or deactivate specific facilities thaks to a bitmap. For example bit 0 enables protected mode, bit 16 enables write protect: in kernel mode we can write

even into memory protected pages, but we can disable this facility setting CR0's bit 16. Or bit 30 enabling/disabling caches, 31 enabling paging etc. In order to enter into protected mode we must set PE bit and PG bit, that respectively enables protected mode and paging. Even if we set PE bit we are not entering immediately in protected mode, because we have to explicitely update CS to do not bring our system into an inconsistent state. This is because between real mode an protected mode we still have segment register but the bit in those registers are interpreted in a completely different way. Hence protected mode expect specific informations into CS thaht are the basis to implement syscall access from user space. We update CS using a long jump.

Now on system can run into 32 or 64 bit mode.

Protected mode keeps data into SEGMENT DESCRIPTOR'S TABLE. We can have multpile time of segment descriptors: data, code, system. System is used to implement proviledge levels in protected mode. Descriptor table is composed of descriptor table entries (DTE), each DTE is a a 64 bit data structure in which we have:

- base addres of the segment (we can still use segmentatio with base addres of 32 bit)
- limit tells the size of the segment, to put some restrictions since now we are in 32bit addressing mode
- granularity tell whether size must be multiplied per page size
- DPL is 2 bit field that identifies a value between 0-3 representing the descriptor priviledge level: is telling the priviledge that an operation accessing this segment should have before actually accessing it.

PM introduce the concept of rings. Linux only uses two: ring 0 (kernell) and 3 (user). Relying on dpl we can implemnt part of the ring protection mechanism. At ring 3 we can not execuse some machine instructions, to perform them we must give the controll to OS at ring 0.

There are 2 descriptor tabel: Global and Local. The Global is shared among cpus. The Local is associated with specific core or with specific application running on a specific core. The idea is to give the os the opportunity to implement different access priviledges to its facilities wrt each process running. They are also used by cpu to implement memory protection some mecanisms. LDT is no more used. We have LDTR and GDTR which are two registers containing pointer to those tables.

Segment selectors have an index wich allows to tell what is the segment selector in the specific table, TI is set to 0 for the GDT, set to 1 for the LDT, RPL is requested priviledge level.

Segmented address resolution in protected mode: we have an instruction with a memory destination (an offset wrt to base of the segment). Suppose we have a jump instruction, it is associated with cs in which we now have an index telling what is the entry in one of the two tables in order to find out the segment descriptor, a bit specifing the table, a CPL. $Index*8$+gdtr gives us the exact segment descriptor into the gdt giving us the base addres of the segment. Since our instruciotn is a jump, it specifies an offset that is added to the base addres obtained from the segment descriptor in the GDT to give us the linear addres generated from MMU. Remeber: we have explicitely written 0 into all segments. We are in what is called the flat memory model in wich, since base=0, the offset is no longer an offset but is the linear addres generated from MMU.

Segment caching: since gdt is in memory we can speed up the access implementing a caching mechanism. We have a segment register with a programmable selector and a non programmable descriptor. Any time we use a segment selector we store a shadow copy into the segment register of the accessed descriptor. Problem if we modify the table and the previous value is cached we do not see the update. If we want to handle the problem if we modify the descriptor table we reload the segment selector in the segment register also with the same value in ordet to flush the cache in the segment register and forces MMU to reload the segment descriptor. NOTE it is very rare that segment's content changes, we change it only once we switch from user mode to kernel mode.