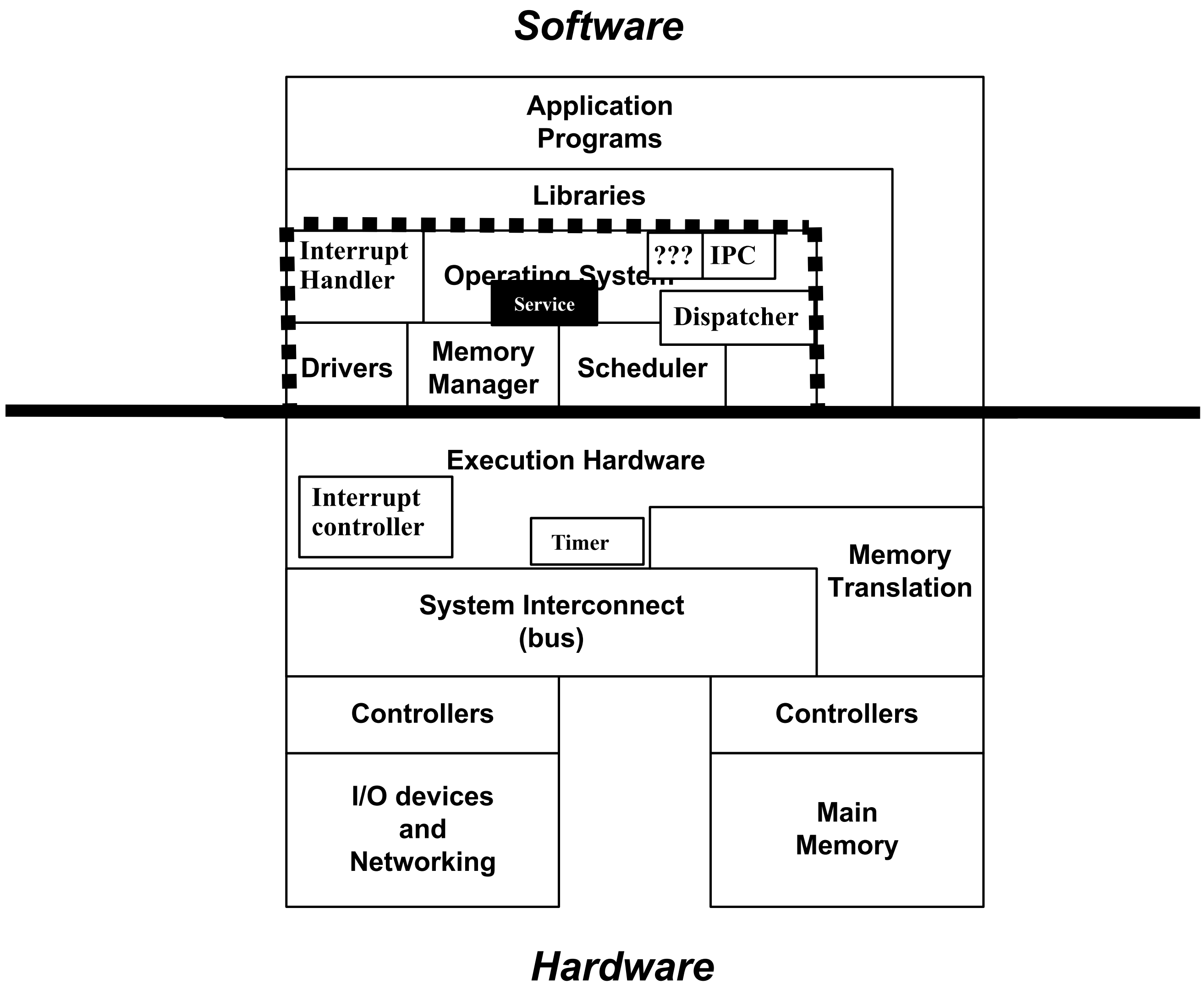
Operating Systems Structure

Otto J. Anshus

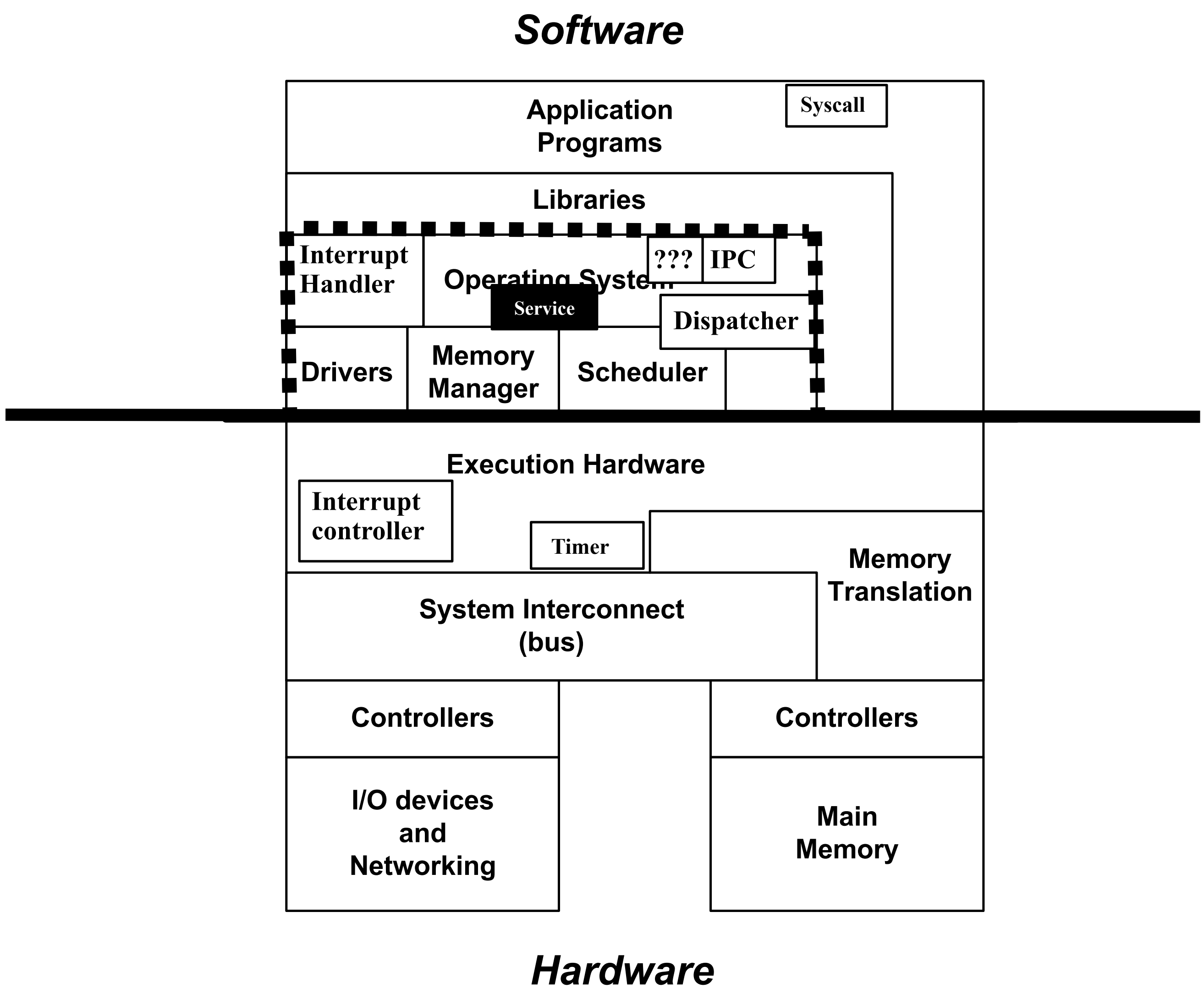
Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)



Border UL-KL

Border SW-HW

Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)

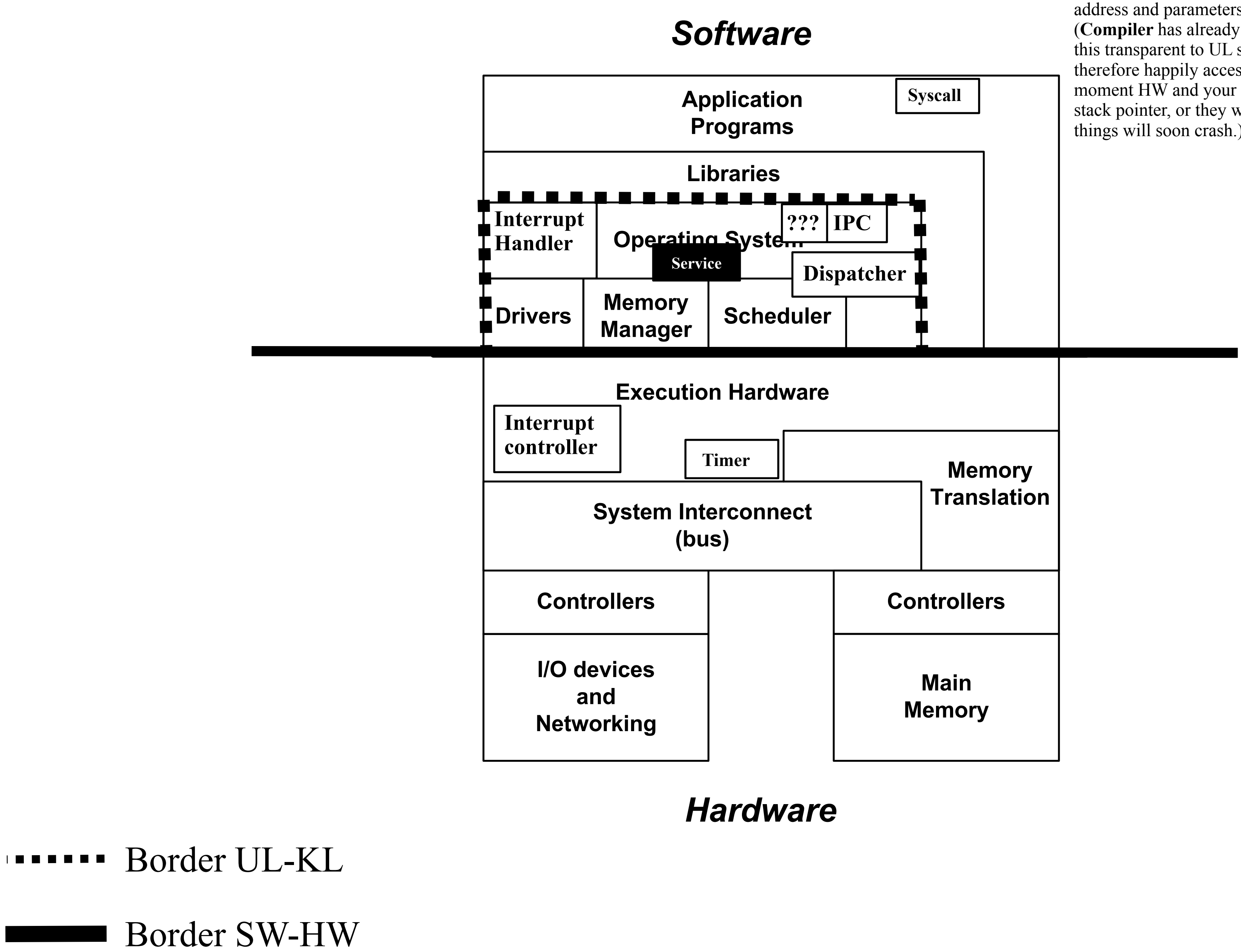


Border UL-KL

Border SW-HW

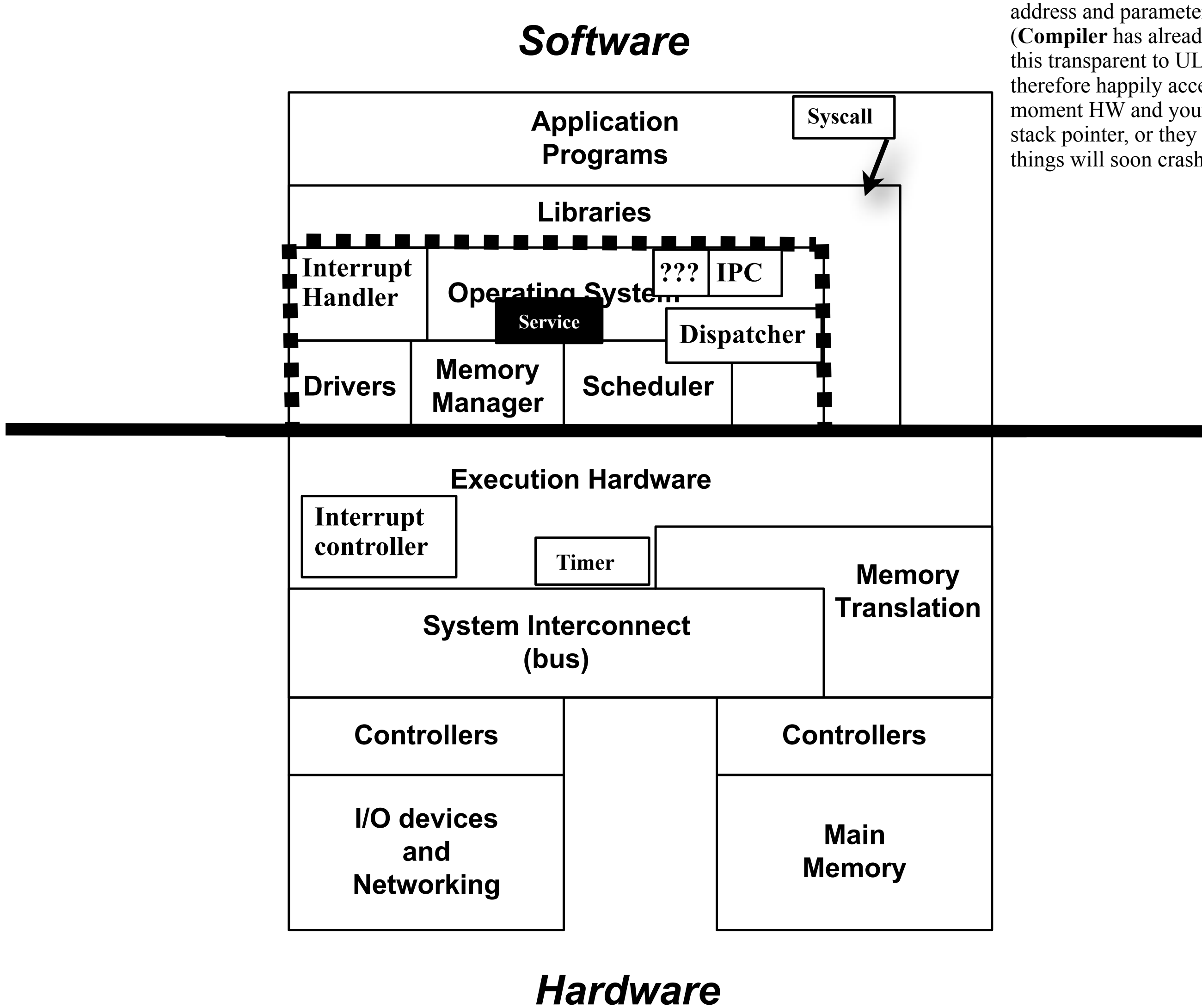
Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)

A call to a library routine is just a normal UL call: the return address and parameters are **pushed** on (user level) **stack**. (**Compiler** has already inserted code to do this and to make this transparent to UL subroutine code (subroutine code can therefore happily access the parameters). However, in a moment HW and your OS must be careful with the stack and stack pointer, or they will probably be lost or overwritten, and things will soon crash.)



Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)

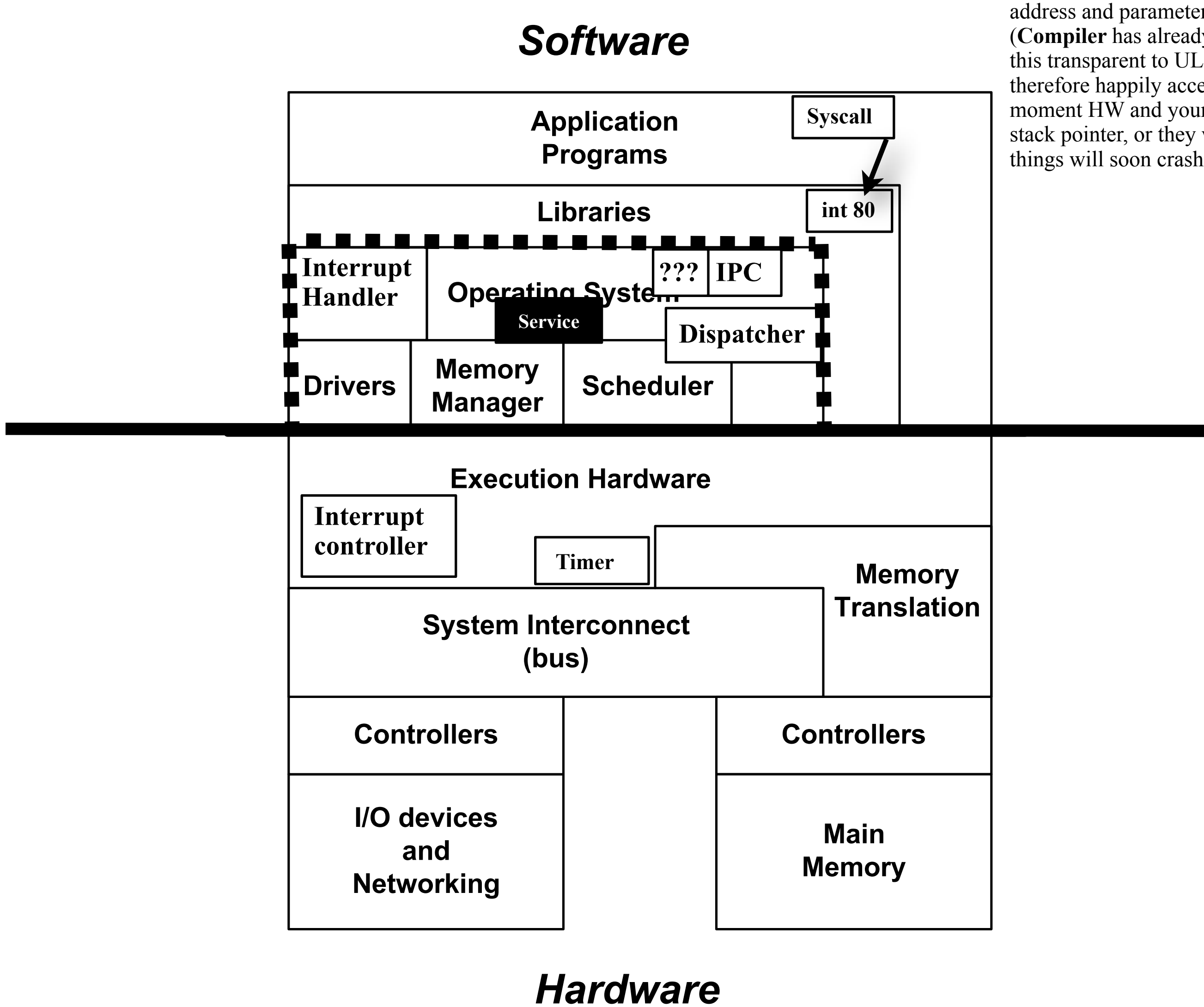
A call to a library routine is just a normal UL call: the return address and parameters are **pushed** on (user level) **stack**. (**Compiler** has already inserted code to do this and to make this transparent to UL subroutine code (subroutine code can therefore happily access the parameters). However, in a moment HW and your OS must be careful with the stack and stack pointer, or they will probably be lost or overwritten, and things will soon crash.)



Border UL-KL

Border SW-HW

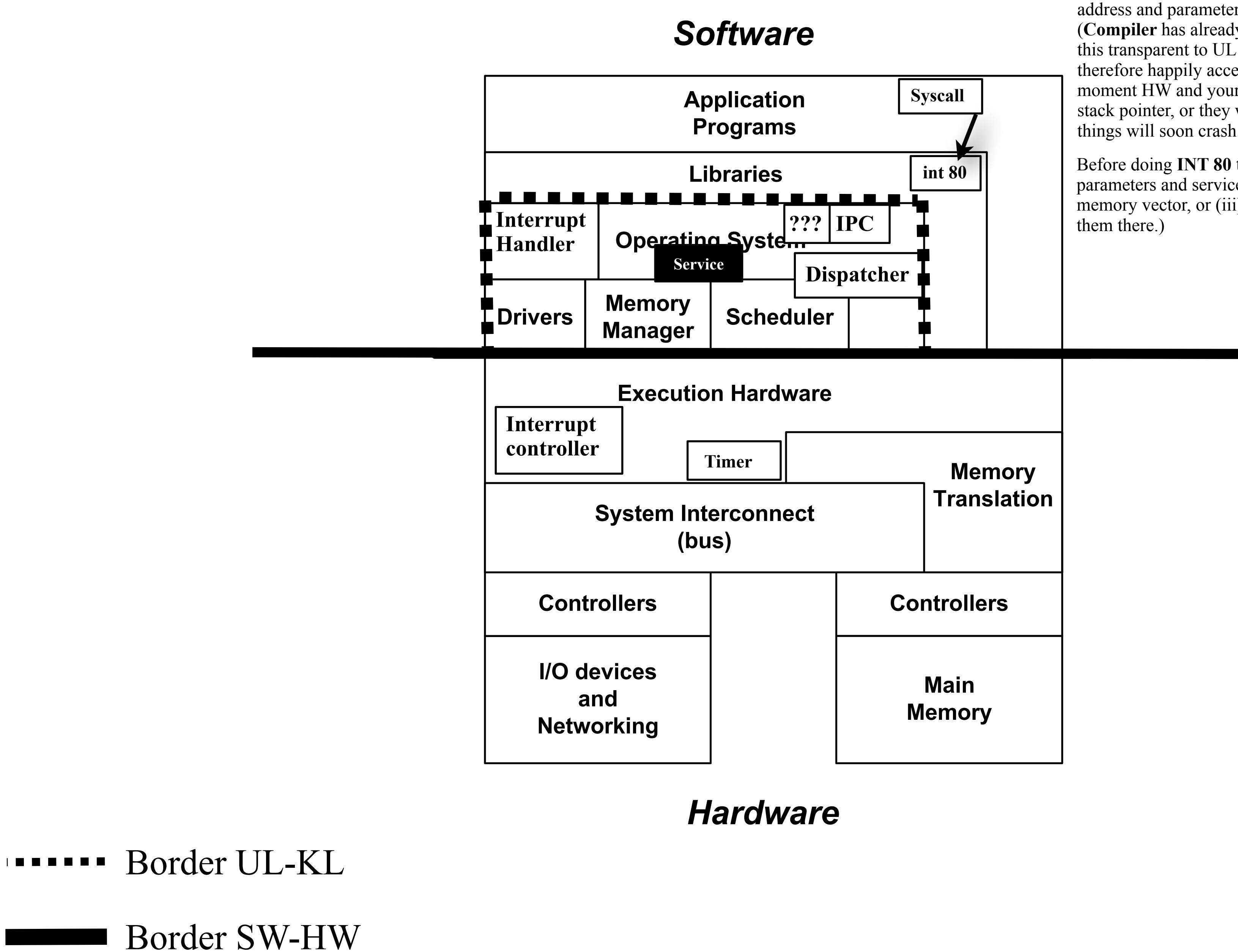
A call to a library routine is just a normal UL call: the return address and parameters are **pushed** on (user level) **stack**. (**Compiler** has already inserted code to do this and to make this transparent to UL subroutine code (subroutine code can therefore happily access the parameters). However, in a moment HW and your OS must be careful with the stack and stack pointer, or they will probably be lost or overwritten, and things will soon crash.)



▪▪▪▪▪▪ Border UL-KL

▬▬ Border SW-HW

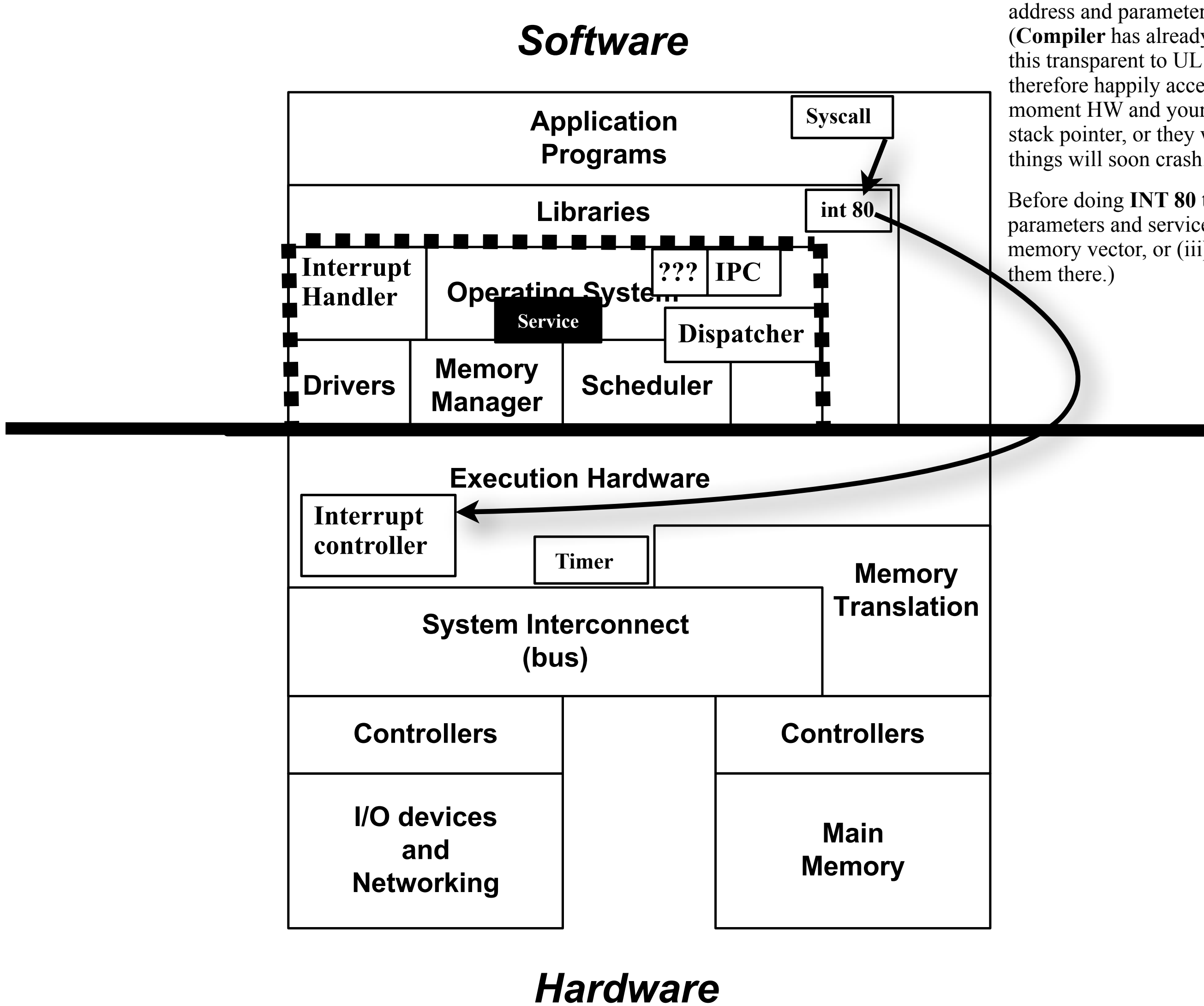
Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)



A call to a library routine is just a normal UL call: the return address and parameters are **pushed** on (user level) **stack**. (**Compiler** has already inserted code to do this and to make this transparent to UL subroutine code (subroutine code can therefore happily access the parameters). However, in a moment HW and your OS must be careful with the stack and stack pointer, or they will probably be lost or overwritten, and things will soon crash.)

Before doing **INT 80** the library routine will take the parameters and service ID and place them in (i) registers, (ii) memory vector, or (iii) stack (so that the Kernel can fetch them there.)

Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)



A call to a library routine is just a normal UL call: the return address and parameters are **pushed** on (user level) **stack**. (**Compiler** has already inserted code to do this and to make this transparent to UL subroutine code (subroutine code can therefore happily access the parameters). However, in a moment HW and your OS must be careful with the stack and stack pointer, or they will probably be lost or overwritten, and things will soon crash.)

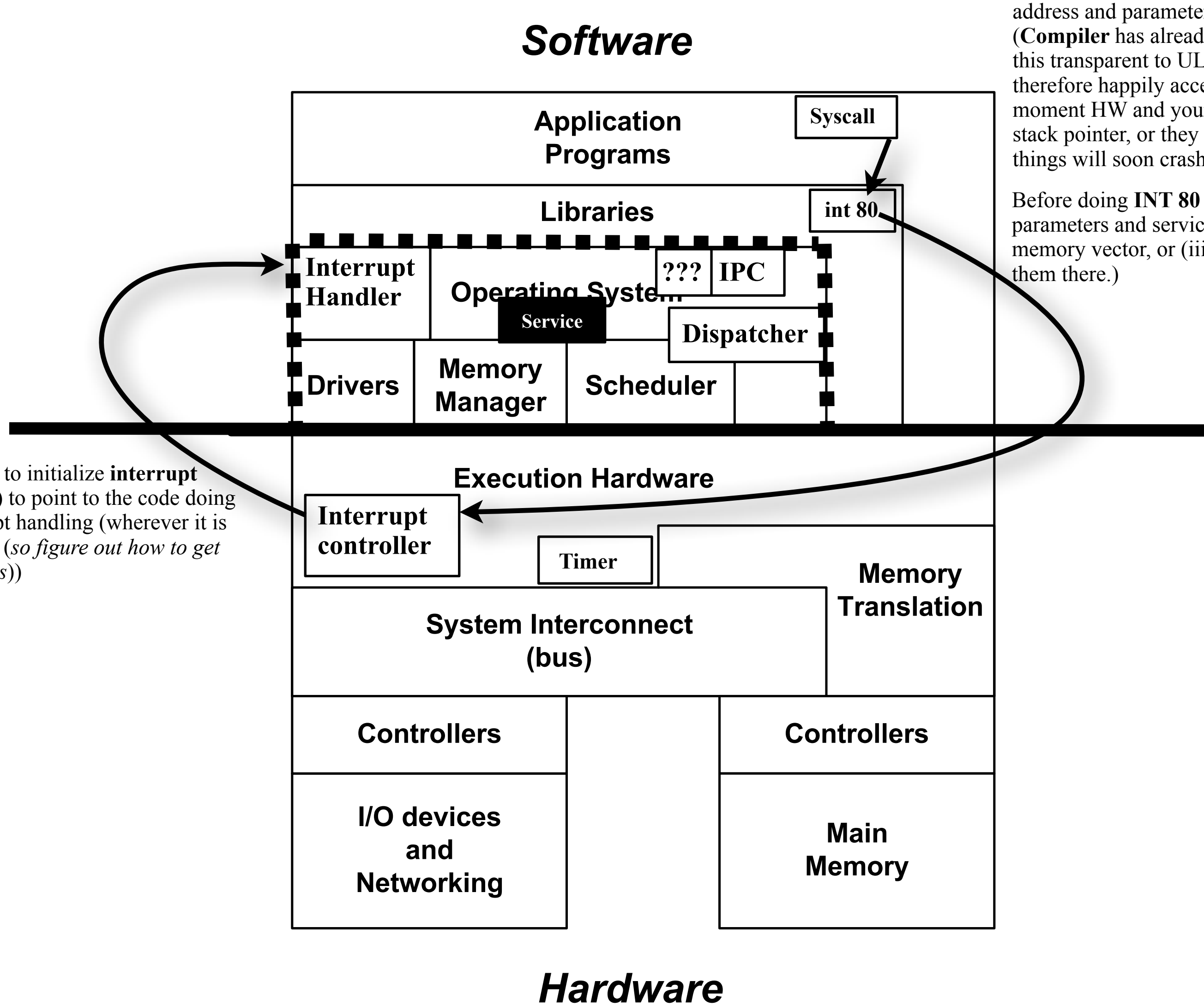
Before doing **INT 80** the library routine will take the parameters and service ID and place them in (i) registers, (ii) memory vector, or (iii) stack (so that the Kernel can fetch them there.)

Border UL-KL

Border SW-HW

A call to a library routine is just a normal UL call: the return address and parameters are **pushed** on (user level) **stack**. (**Compiler** has already inserted code to do this and to make this transparent to UL subroutine code (subroutine code can therefore happily access the parameters). However, in a moment HW and your OS must be careful with the stack and stack pointer, or they will probably be lost or overwritten, and things will soon crash.)

Before doing **INT 80** the library routine will take the parameters and service ID and place them in (i) registers, (ii) memory vector, or (iii) stack (so that the Kernel can fetch them there.)



Remember to initialize **interrupt table (80h)** to point to the code doing the interrupt handling (wherever it is in memory (*so figure out how to get this address*))

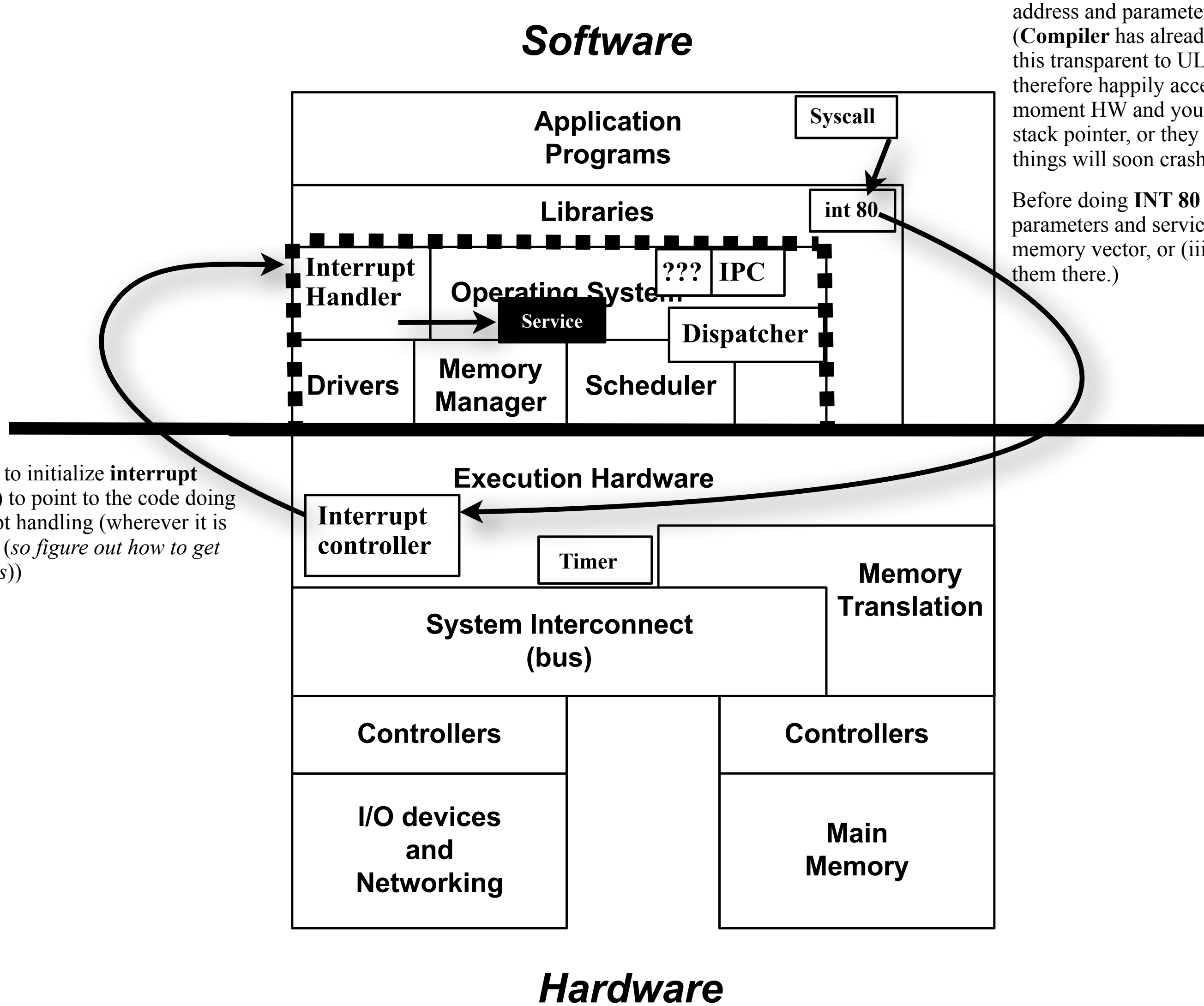
▪▪▪▪▪▪ Border UL-KL

▬ Border SW-HW

Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)

A call to a library routine is just a normal UL call: the return address and parameters are **pushed** on (user level) **stack**. (**Compiler** has already inserted code to do this and to make this transparent to UL subroutine code (subroutine code can therefore happily access the parameters). However, in a moment HW and your OS must be careful with the stack and stack pointer, or they will probably be lost or overwritten, and things will soon crash.)

Before doing **INT 80** the library routine will take the parameters and service ID and place them in (i) registers, (ii) memory vector, or (iii) stack (so that the Kernel can fetch them there.)



Remember to initialize **interrupt table (80h)** to point to the code doing the interrupt handling (wherever it is in memory (*so figure out how to get this address*))

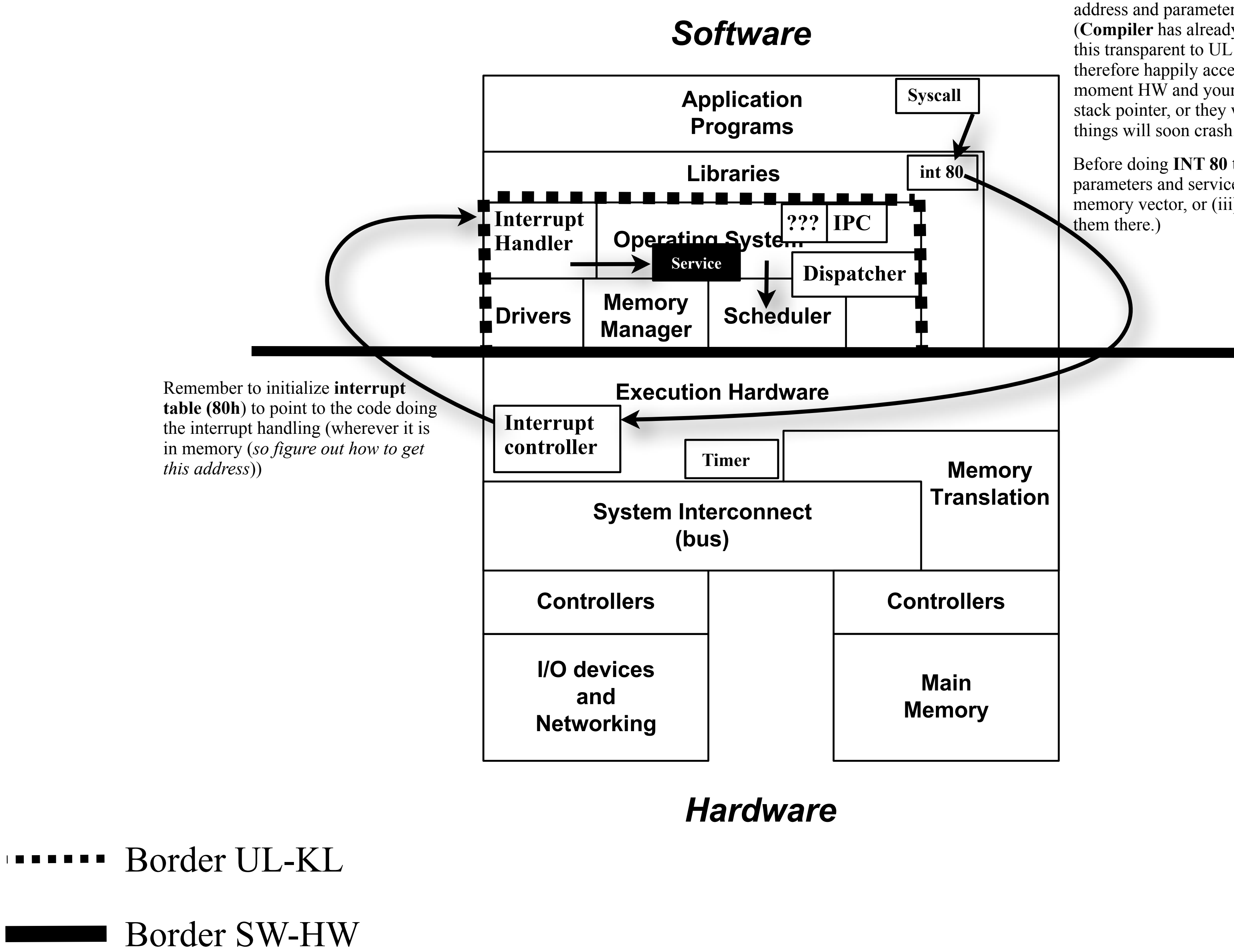
Border UL-KL

Border SW-HW

Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)

A call to a library routine is just a normal UL call: the return address and parameters are **pushed** on (user level) **stack**. (**Compiler** has already inserted code to do this and to make this transparent to UL subroutine code (subroutine code can therefore happily access the parameters). However, in a moment HW and your OS must be careful with the stack and stack pointer, or they will probably be lost or overwritten, and things will soon crash.)

Before doing **INT 80** the library routine will take the parameters and service ID and place them in (i) registers, (ii) memory vector, or (iii) stack (so that the Kernel can fetch them there.)

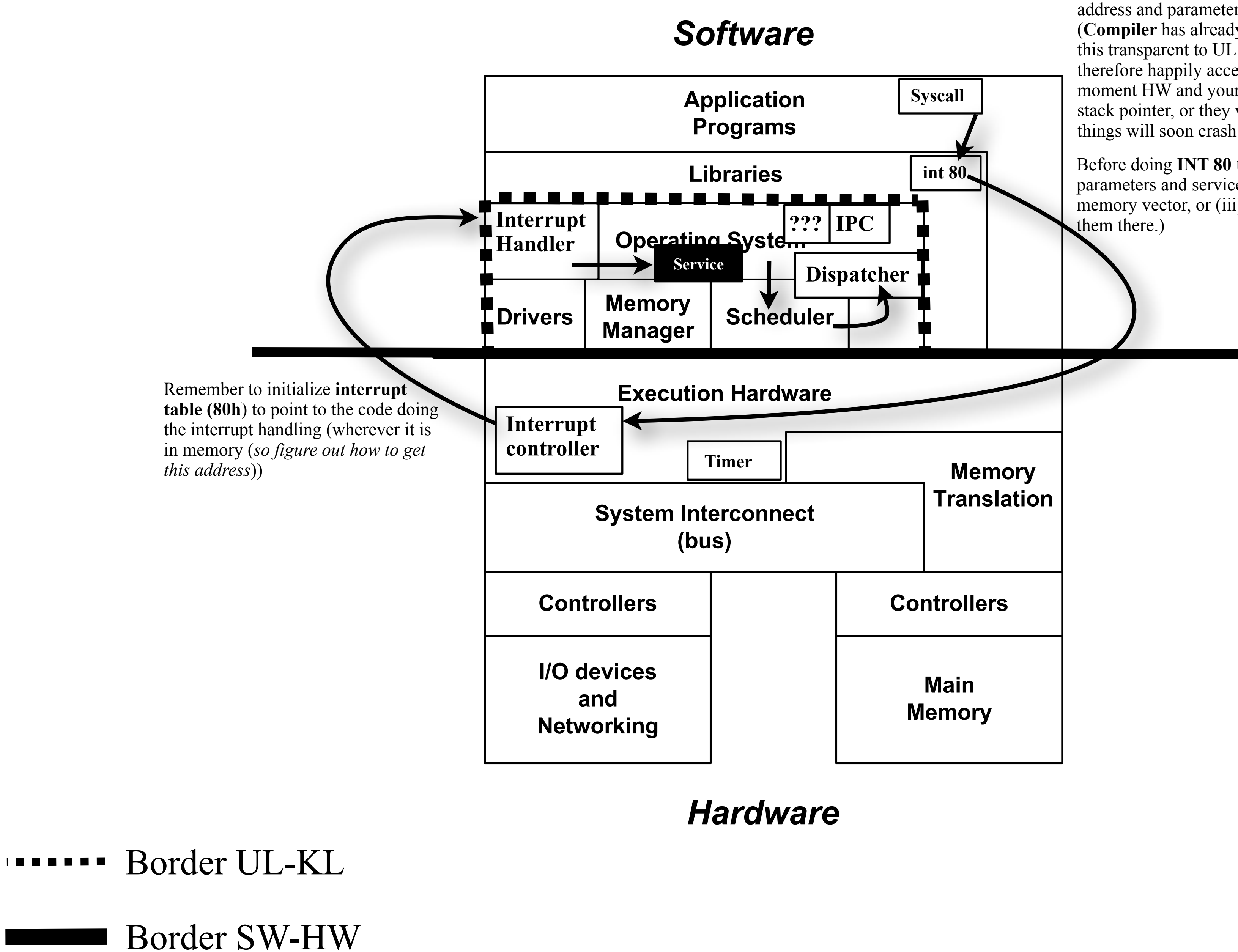


Remember to initialize **interrupt table (80h)** to point to the code doing the interrupt handling (wherever it is in memory (*so figure out how to get this address*))

Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)

A call to a library routine is just a normal UL call: the return address and parameters are **pushed** on (user level) **stack**. (**Compiler** has already inserted code to do this and to make this transparent to UL subroutine code (subroutine code can therefore happily access the parameters). However, in a moment HW and your OS must be careful with the stack and stack pointer, or they will probably be lost or overwritten, and things will soon crash.)

Before doing **INT 80** the library routine will take the parameters and service ID and place them in (i) registers, (ii) memory vector, or (iii) stack (so that the Kernel can fetch them there.)

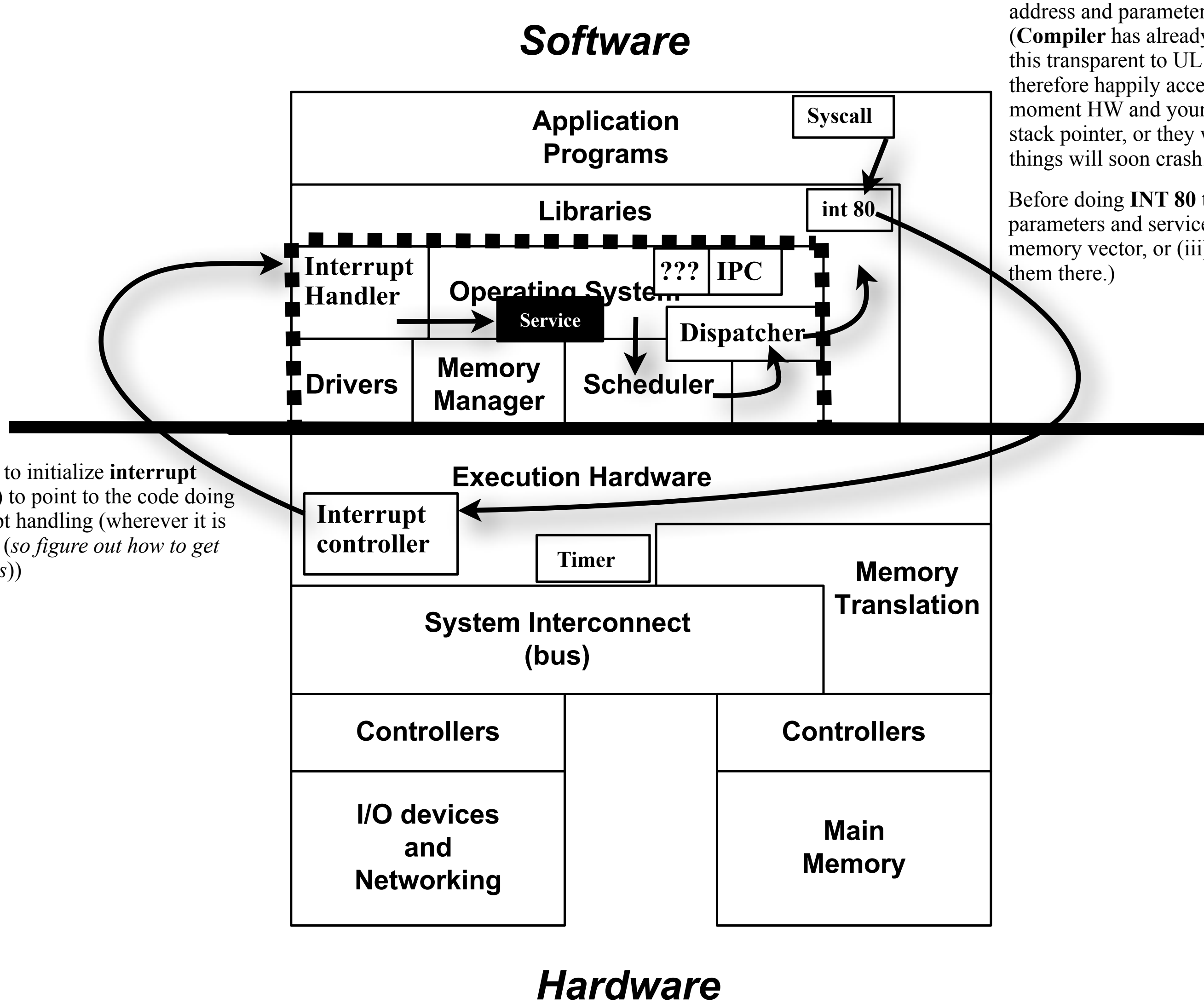


Remember to initialize **interrupt table (80h)** to point to the code doing the interrupt handling (wherever it is in memory (*so figure out how to get this address*))

Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)

A call to a library routine is just a normal UL call: the return address and parameters are **pushed** on (user level) **stack**. (**Compiler** has already inserted code to do this and to make this transparent to UL subroutine code (subroutine code can therefore happily access the parameters). However, in a moment HW and your OS must be careful with the stack and stack pointer, or they will probably be lost or overwritten, and things will soon crash.)

Before doing **INT 80** the library routine will take the parameters and service ID and place them in (i) registers, (ii) memory vector, or (iii) stack (so that the Kernel can fetch them there.)



Remember to initialize **interrupt table (80h)** to point to the code doing the interrupt handling (wherever it is in memory (*so figure out how to get this address*))

Border UL-KL

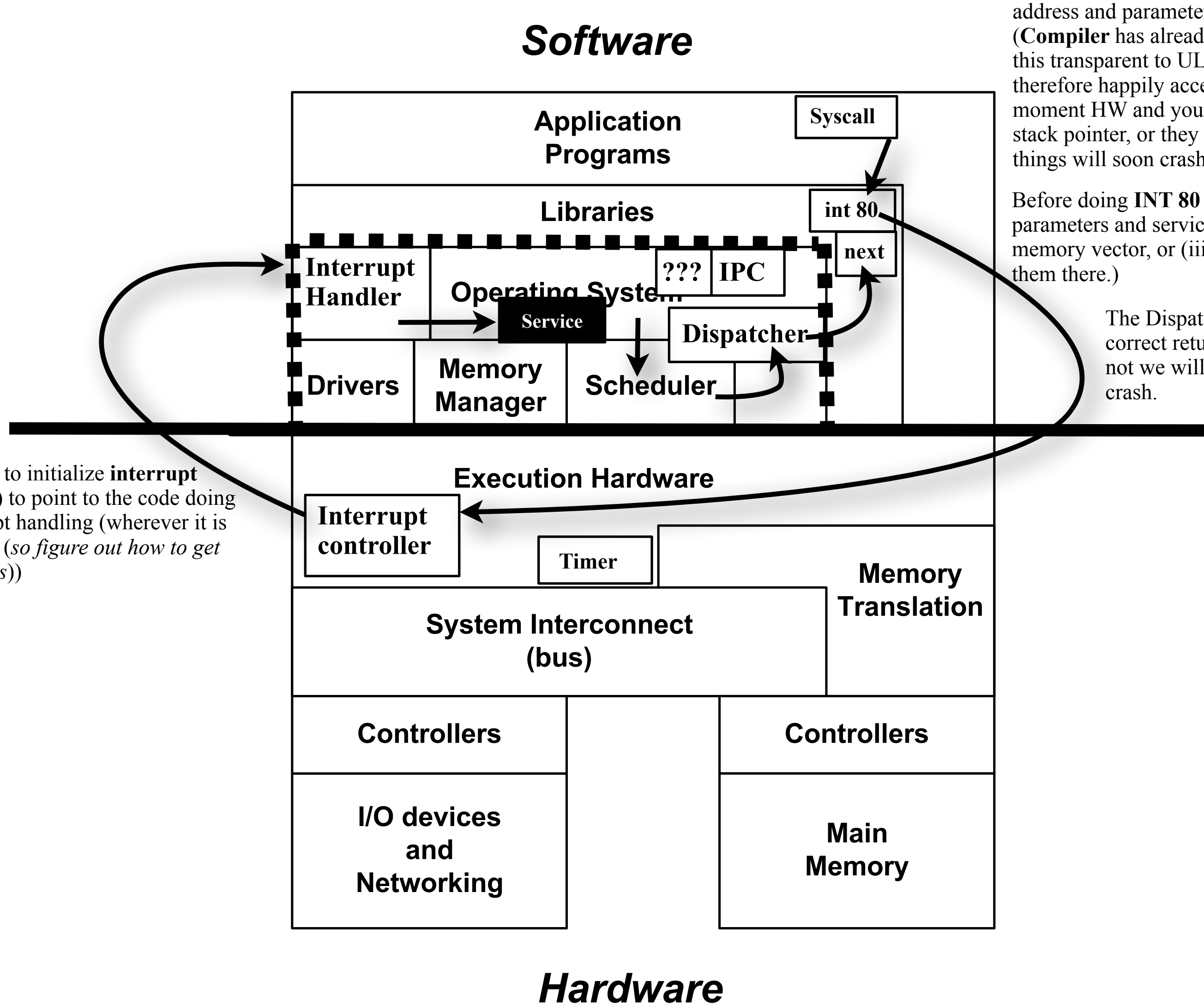
Border SW-HW

Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)

A call to a library routine is just a normal UL call: the return address and parameters are **pushed** on (user level) **stack**. (**Compiler** has already inserted code to do this and to make this transparent to UL subroutine code (subroutine code can therefore happily access the parameters). However, in a moment HW and your OS must be careful with the stack and stack pointer, or they will probably be lost or overwritten, and things will soon crash.)

Before doing **INT 80** the library routine will take the parameters and service ID and place them in (i) registers, (ii) memory vector, or (iii) stack (so that the Kernel can fetch them there.)

The Dispatcher will assume that the stack has the correct return address when the **iret** is issued. If not we will end up somewhere bad and probably crash.



Remember to initialize **interrupt table (80h)** to point to the code doing the interrupt handling (wherever it is in memory (*so figure out how to get this address*))

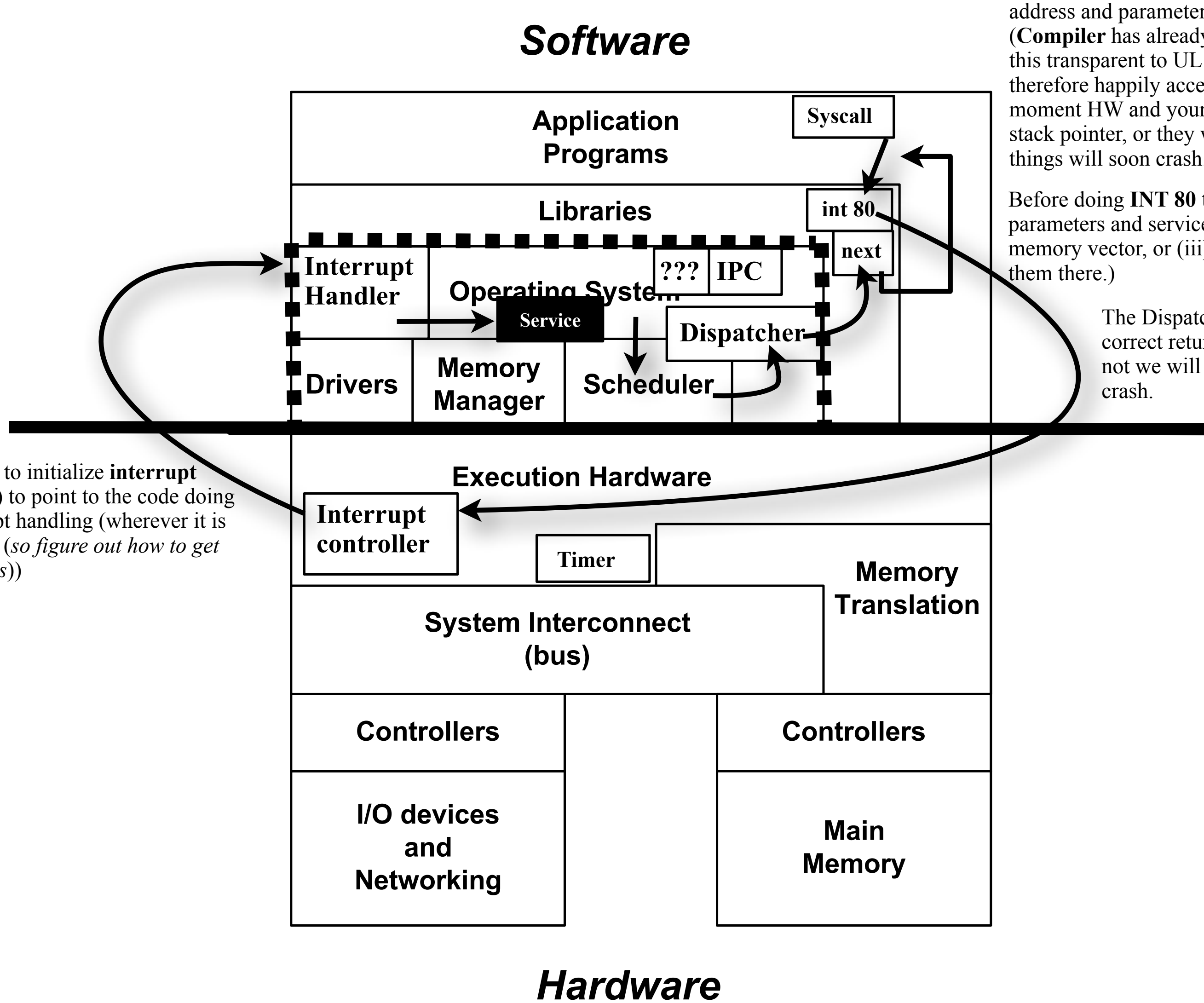
Border UL-KL

Border SW-HW

A call to a library routine is just a normal UL call: the return address and parameters are **pushed** on (user level) **stack**. (**Compiler** has already inserted code to do this and to make this transparent to UL subroutine code (subroutine code can therefore happily access the parameters). However, in a moment HW and your OS must be careful with the stack and stack pointer, or they will probably be lost or overwritten, and things will soon crash.)

Before doing **INT 80** the library routine will take the parameters and service ID and place them in (i) registers, (ii) memory vector, or (iii) stack (so that the Kernel can fetch them there.)

The Dispatcher will assume that the stack has the correct return address when the **iret** is issued. If not we will end up somewhere bad and probably crash.



Remember to initialize **interrupt table (80h)** to point to the code doing the interrupt handling (wherever it is in memory (*so figure out how to get this address*))

Border UL-KL

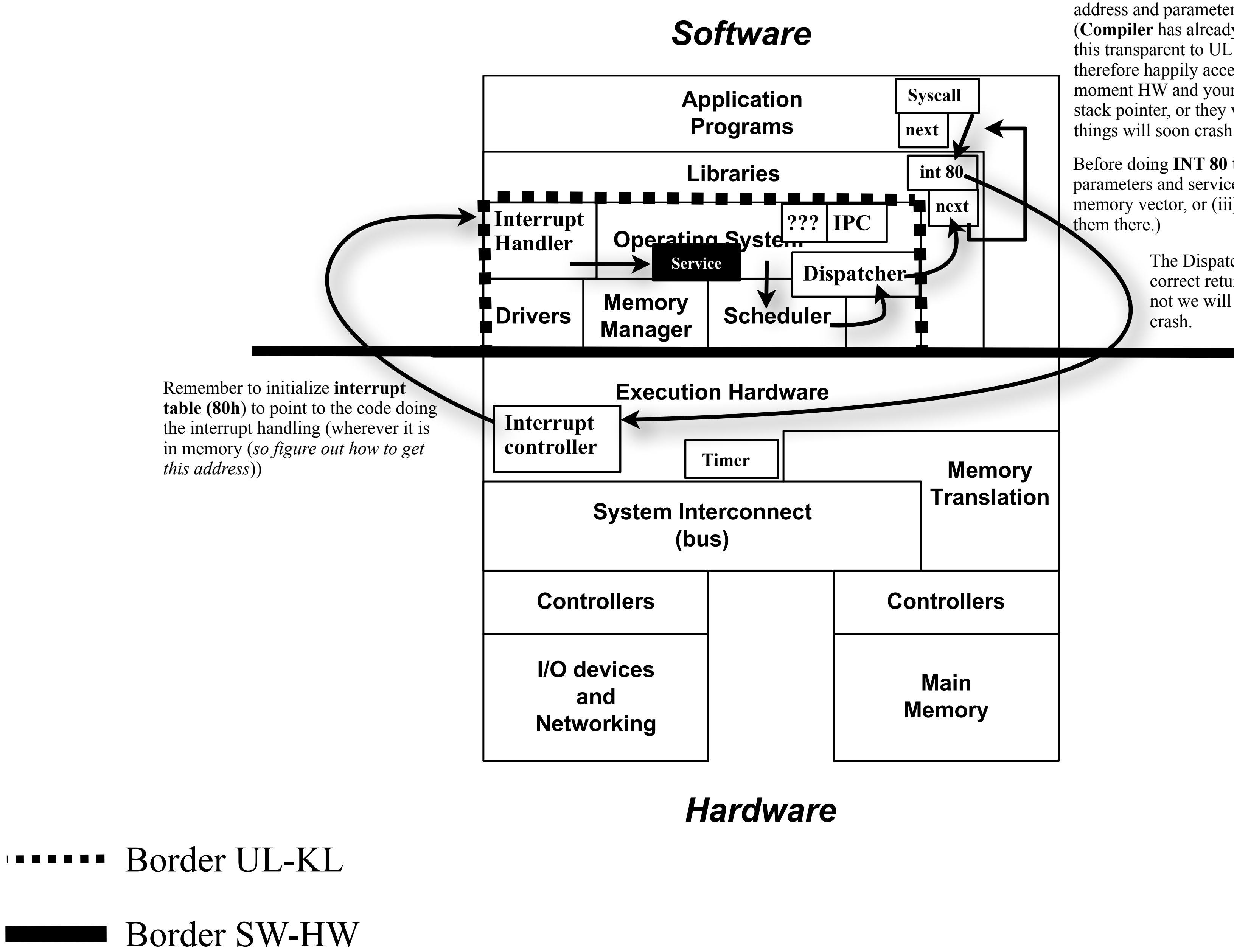
Border SW-HW

Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)

A call to a library routine is just a normal UL call: the return address and parameters are **pushed** on (user level) **stack**. (**Compiler** has already inserted code to do this and to make this transparent to UL subroutine code (subroutine code can therefore happily access the parameters). However, in a moment HW and your OS must be careful with the stack and stack pointer, or they will probably be lost or overwritten, and things will soon crash.)

Before doing **INT 80** the library routine will take the parameters and service ID and place them in (i) registers, (ii) memory vector, or (iii) stack (so that the Kernel can fetch them there.)

The Dispatcher will assume that the stack has the correct return address when the **iret** is issued. If not we will end up somewhere bad and probably crash.



Remember to initialize **interrupt table (80h)** to point to the code doing the interrupt handling (wherever it is in memory (*so figure out how to get this address*))

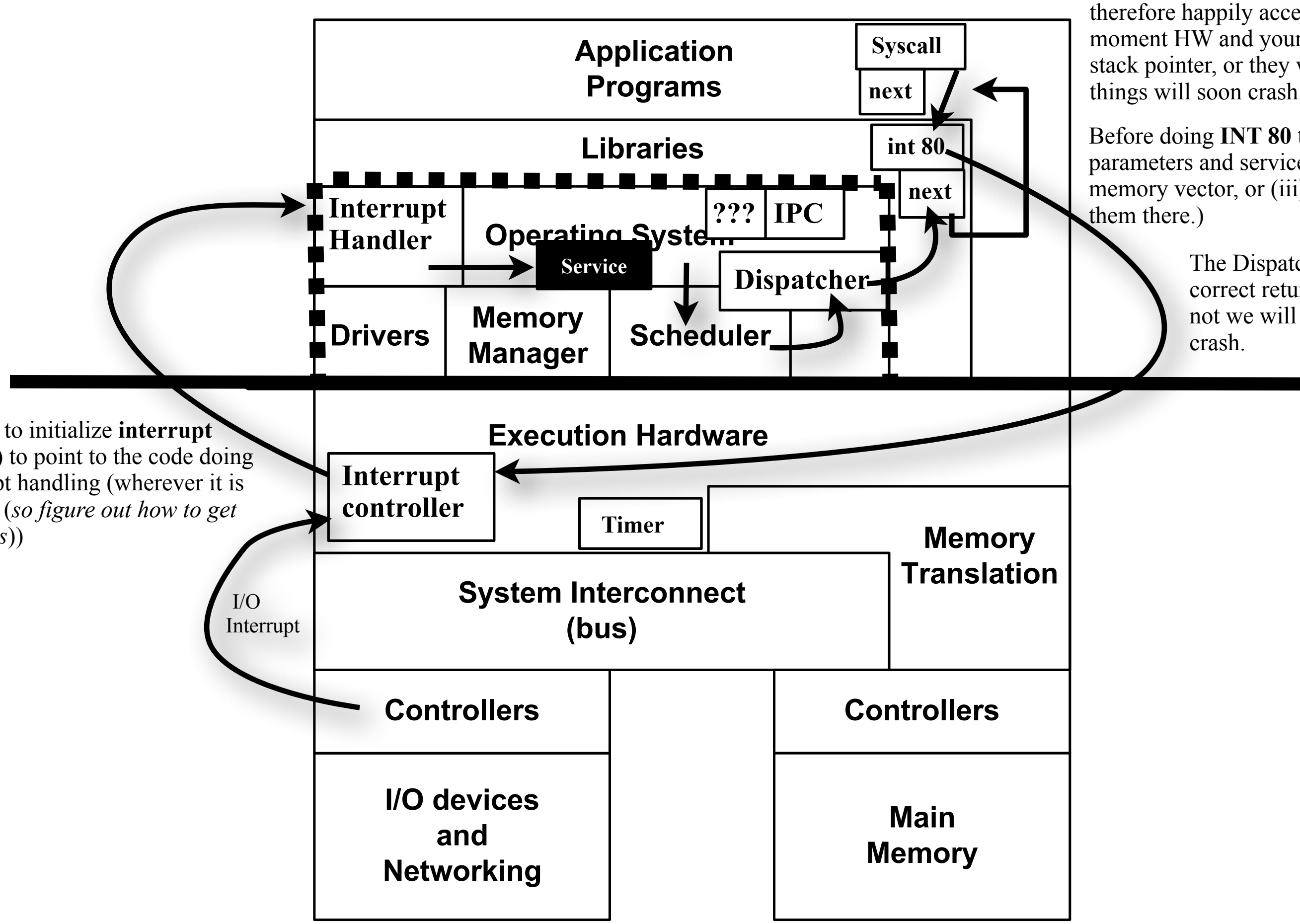
Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)

A call to a library routine is just a normal UL call: the return address and parameters are **pushed** on (user level) **stack**. (**Compiler** has already inserted code to do this and to make this transparent to UL subroutine code (subroutine code can therefore happily access the parameters). However, in a moment HW and your OS must be careful with the stack and stack pointer, or they will probably be lost or overwritten, and things will soon crash.)

Before doing **INT 80** the library routine will take the parameters and service ID and place them in (i) registers, (ii) memory vector, or (iii) stack (so that the Kernel can fetch them there.)

The Dispatcher will assume that the stack has the correct return address when the **iret** is issued. If not we will end up somewhere bad and probably crash.

Software



Remember to initialize **interrupt table (80h)** to point to the code doing the interrupt handling (wherever it is in memory (*so figure out how to get this address*))

Hardware

▪▪▪▪▪▪ Border UL-KL

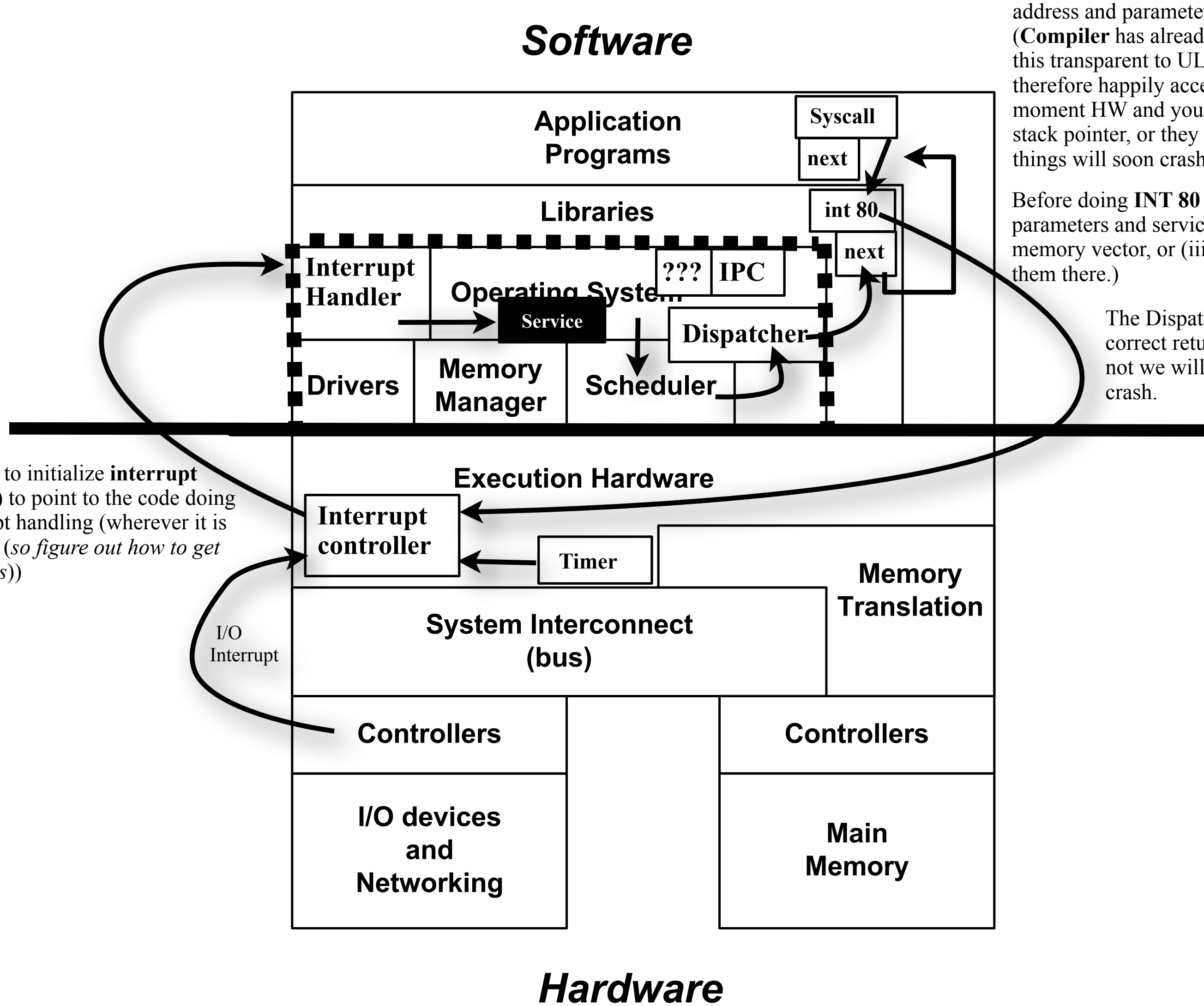
▬ Border SW-HW

Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)

A call to a library routine is just a normal UL call: the return address and parameters are **pushed** on (user level) **stack**. (**Compiler** has already inserted code to do this and to make this transparent to UL subroutine code (subroutine code can therefore happily access the parameters). However, in a moment HW and your OS must be careful with the stack and stack pointer, or they will probably be lost or overwritten, and things will soon crash.)

Before doing **INT 80** the library routine will take the parameters and service ID and place them in (i) registers, (ii) memory vector, or (iii) stack (so that the Kernel can fetch them there.)

The Dispatcher will assume that the stack has the correct return address when the **iret** is issued. If not we will end up somewhere bad and probably crash.



Remember to initialize **interrupt table (80h)** to point to the code doing the interrupt handling (wherever it is in memory (*so figure out how to get this address*))

Border UL-KL

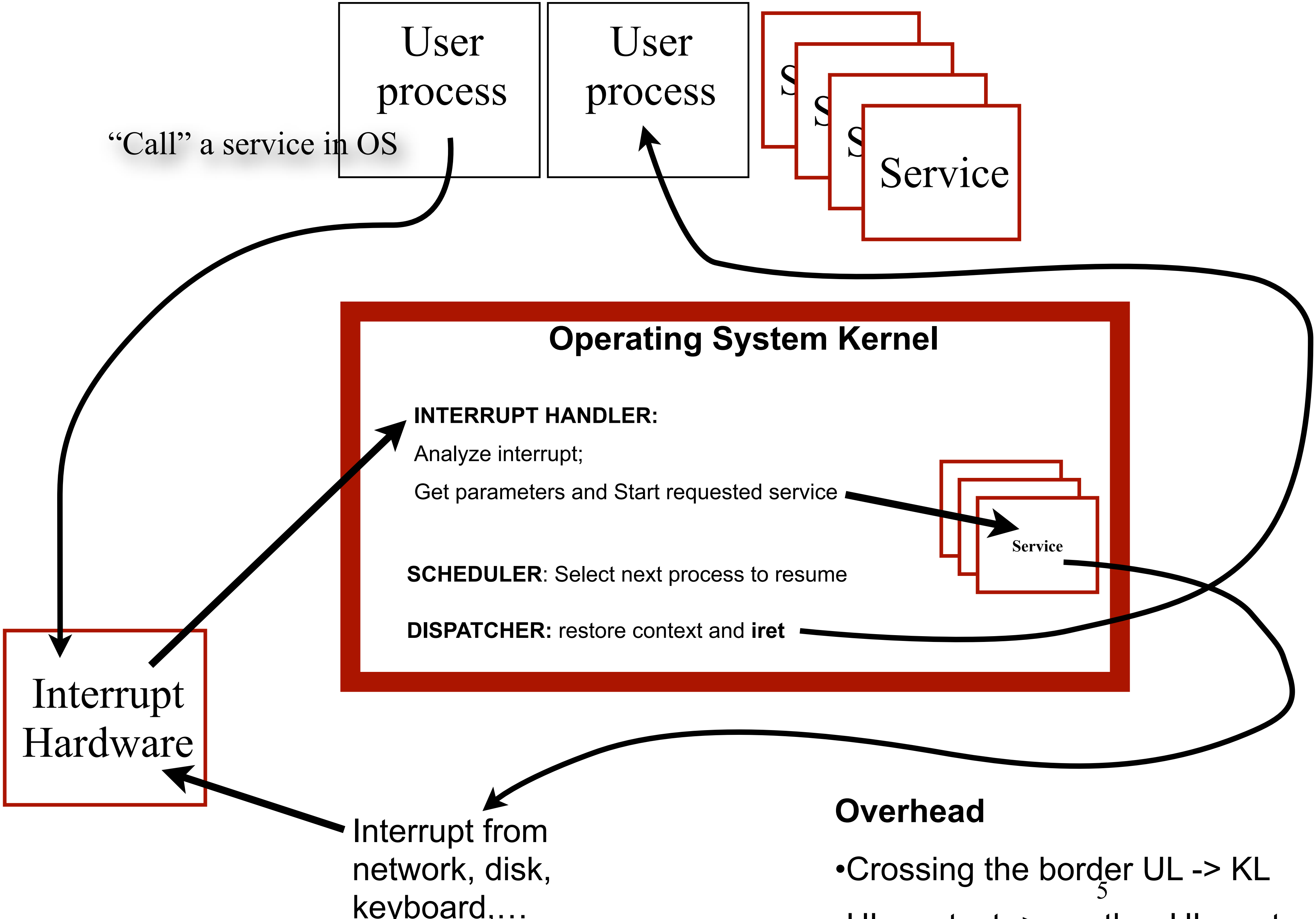
Border SW-HW

The Architecture of an OS

- Layered
- Monolithic
- Micro kernel and Client/Server
- Virtual Machine, (Library, Exokernel)
- Hybrids

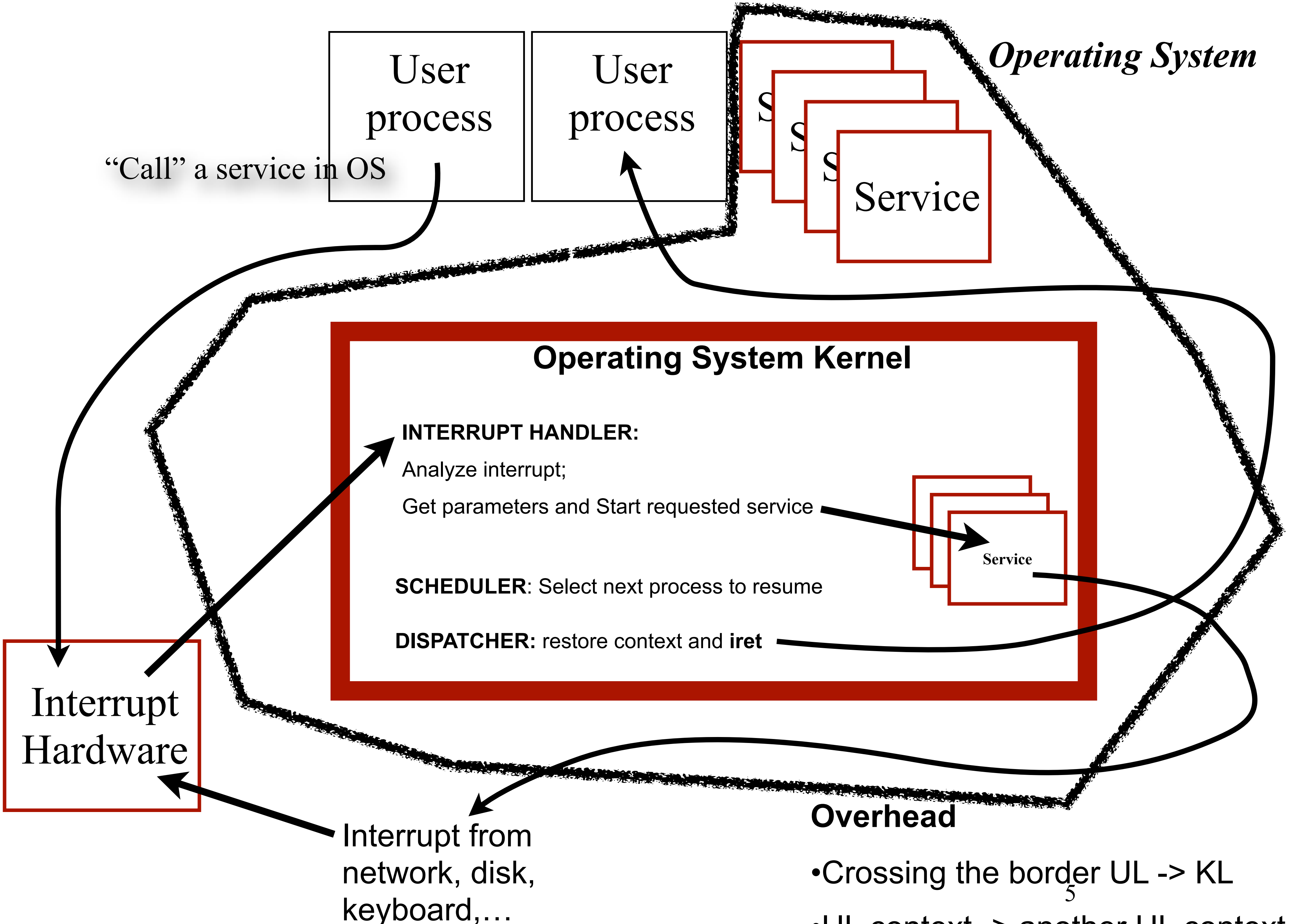
Goals of the architecture

- OS as Resource Manager
- OS as Virtual Machine (abstractions)
- Architecture, Design, Implementation, & Tuning result in OS being:
  - Protective, interactively fast, throughput fast, energy efficient, flexible, secure, small (easier to do protection, security, performance, less bugs)



Overhead

- Crossing the border UL -> KL
- UL context -> another UL context

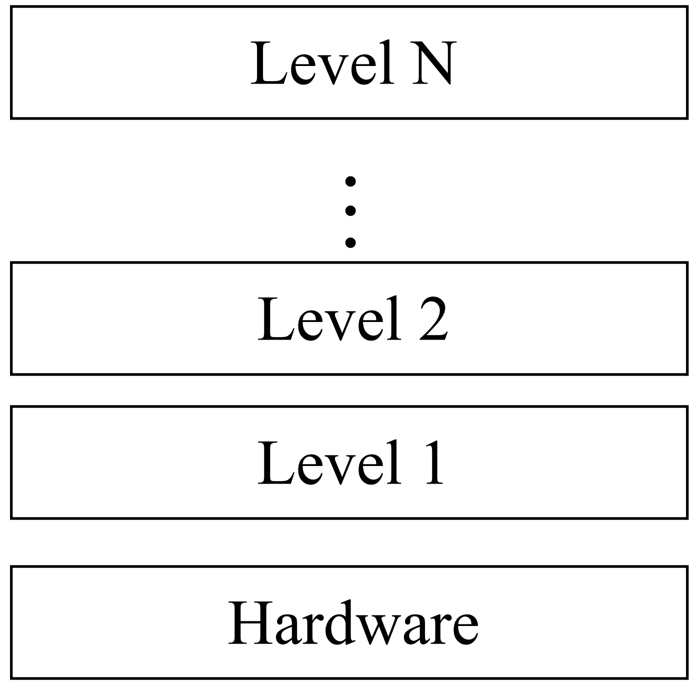


Overhead

- Crossing the border UL -> KL
- UL context -> another UL context

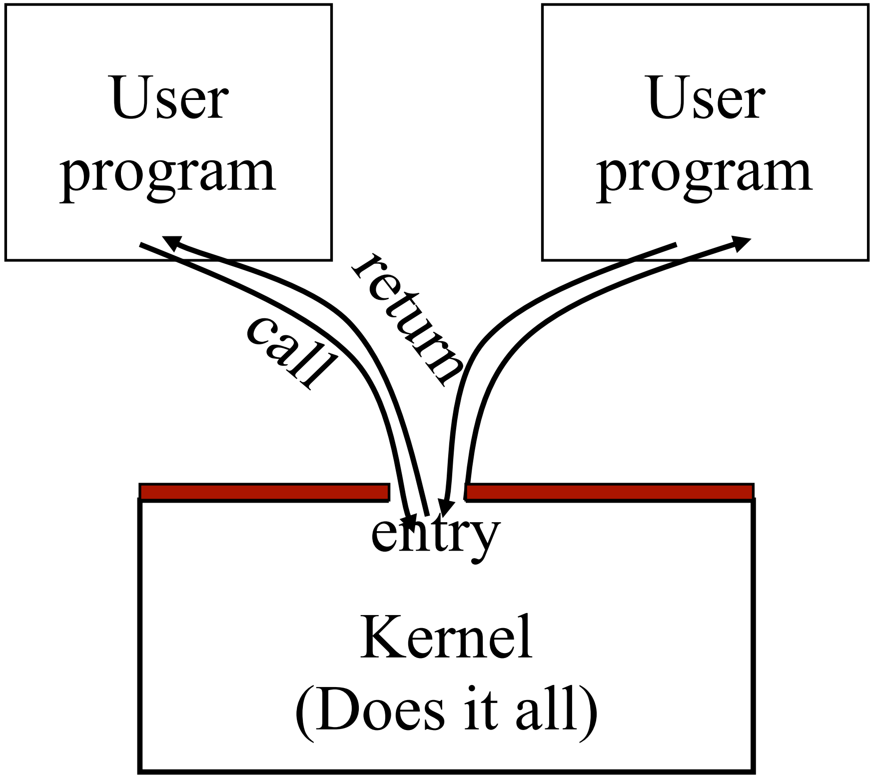
Layered Structure

- Hiding information at each layer
- Develop a layer at a time
- Examples
  - THE (6 layers, semaphores, Dijkstra 1968)
  - MS-DOS (4 layers)
- Pros
  - Separation of concerns
  - Elegance
- Cons
  - Protection boundary crossings
  - Performance



Monolithic

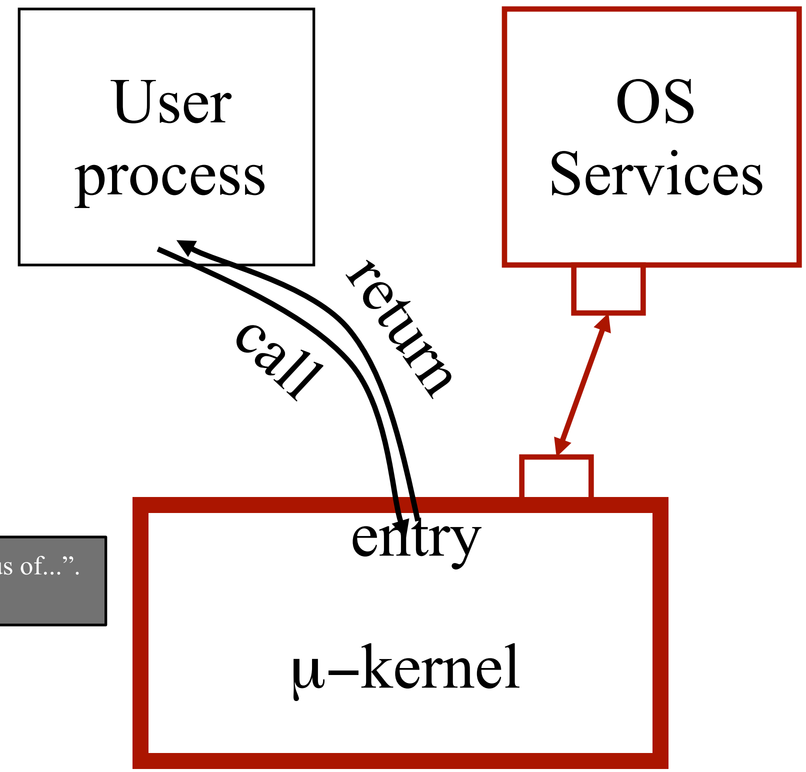
- All kernel routines are together
- A system call interface
- Examples (of fat kernels):
  - Classic Unix (Linux, BSD Unix, ...)
  - Windows NT (hybrid)
  - Mach (as a fat kernel)
  - OS X (fat kernel, but...)
- Pro
  - Performance
  - Shared kernel space
- Cons
  - Stability
  - Flexibility



Microkernel

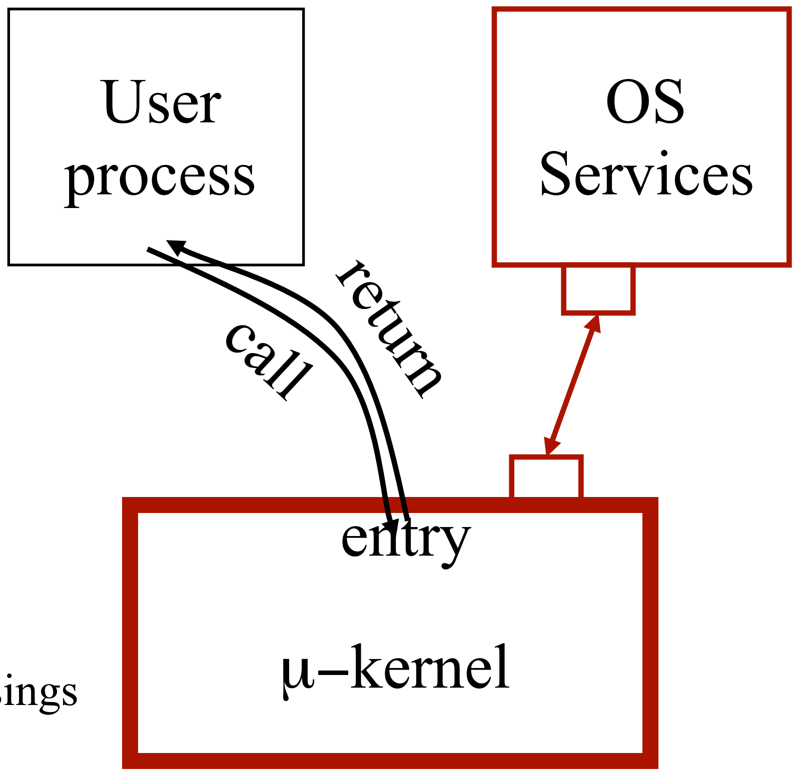
- Micro-kernel is "small"
  - process abstraction, address space, interrupts
- Services are implemented as user level processes
- Micro-kernel get services on behalf of users by messaging with the service processes
- Example: L4, **Nucleus**, Taos, Mach (as a micro kernel), OS-X (*not, but uses some technologies from Mach making it different from BSD and Linux*)

Brinch-Hansen: "The Nucleus of...". Recommended read.



Microkernel Pros et Cons

- Pros
  - Easier to
    - extend or customize
    - Port to a new platform
  - Fault isolation
  - Smaller kernel => easier to tune/optimize
- Cons
  - Performance
    - Naive case: Many protection boundary crossings
      - How many?
  - Harder to let system services share resources
    - Why?



“Truths” on Micro Kernel Flexibility and Performance

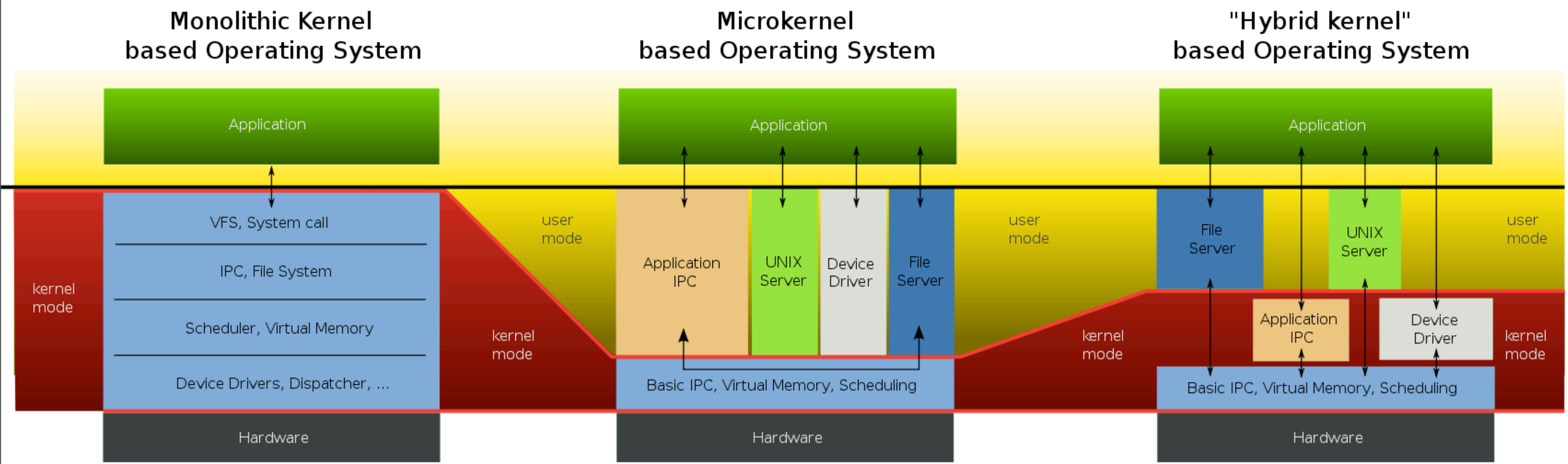
- A micro kernel restricts application level flexibility.
- Switching overhead kernel-user mode is inherently expensive.
- Switching address-spaces is costly.
- IPC is expensive.
- Micro kernel architectures lead to memory system degradation.
- Kernel should be portable (on top of a small hardware-dependent layer).

NO: Can be <50 cycles

NO: 6-20 microsec round-trip, 53-500 cycles/IPC one way

The answer is: **Not necessarily so**

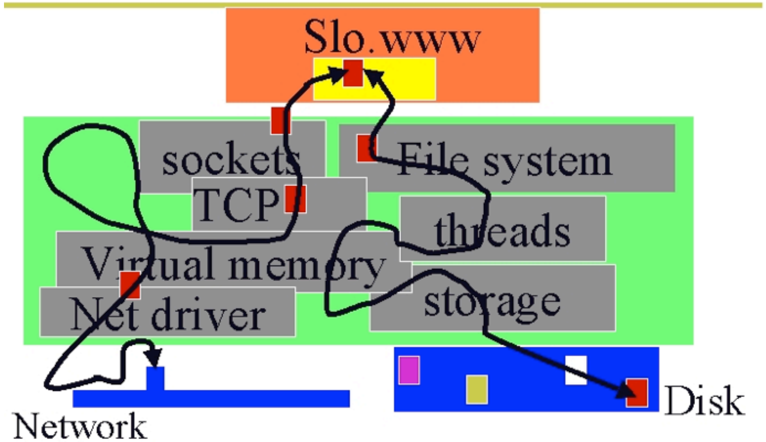
Taken from J. Liedtke, SOSP 15 paper: ”On micro kernel construction”



- http://en.wikipedia.org/wiki/File:OS-structure2.svg

Exokernels

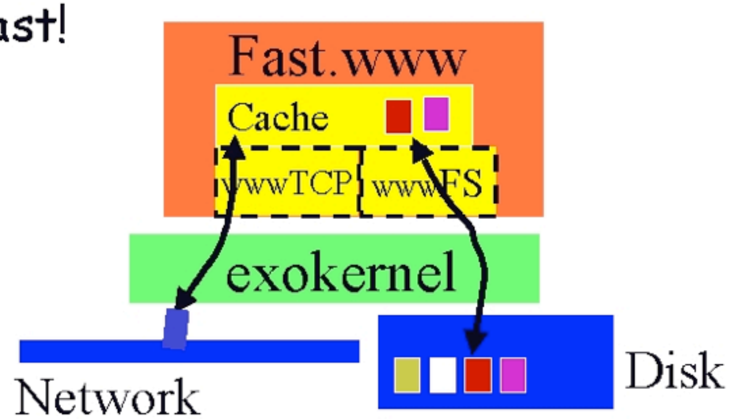
Traditional OS structure



- slow and can't fix it!

Exokernel: application control

- Application software can override OS
- Fast!

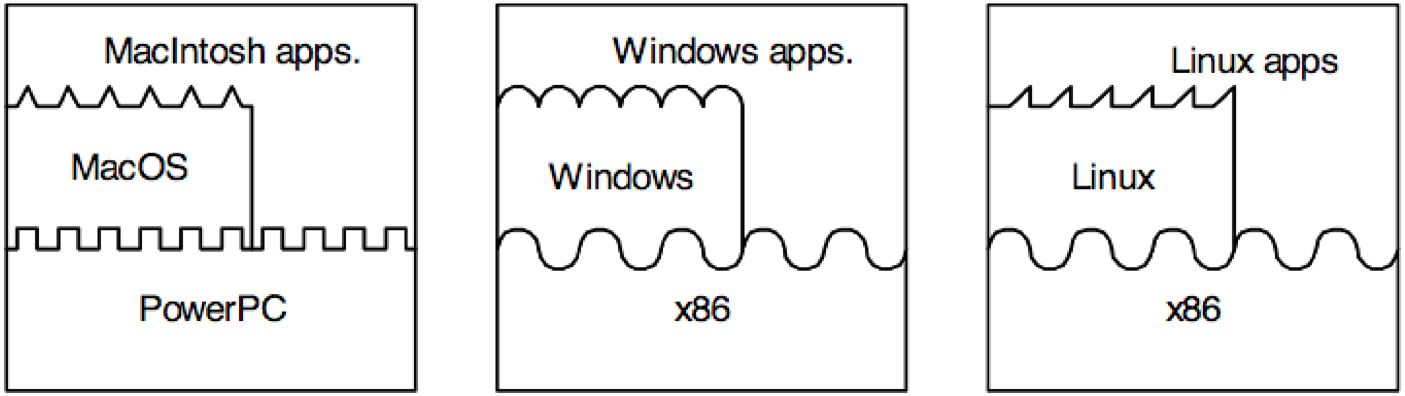


http://pdos.csail.mit.edu/exo/exo-slides/sld003.htm

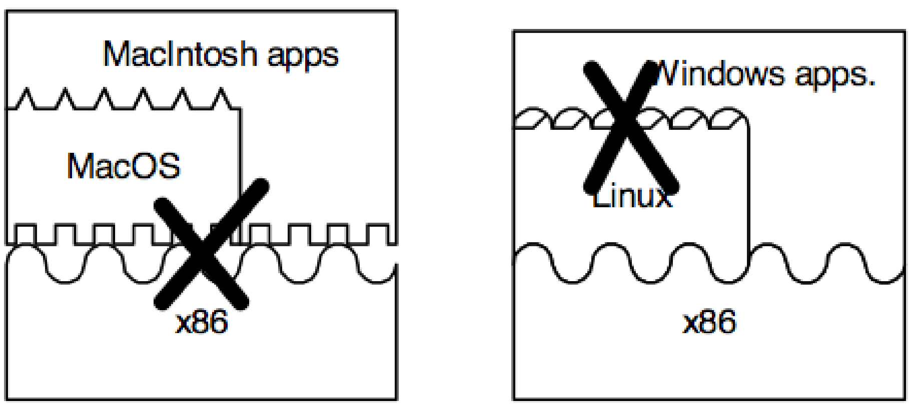
http://pdos.csail.mit.edu/exo/exo-slides/sld004.htm

Taken from Smith and Nairs book: Virtual Machine Architectures, Implementations, and Applications

Life is Hard?



(a)



Well, can be done today
after Apple's switch to Intel

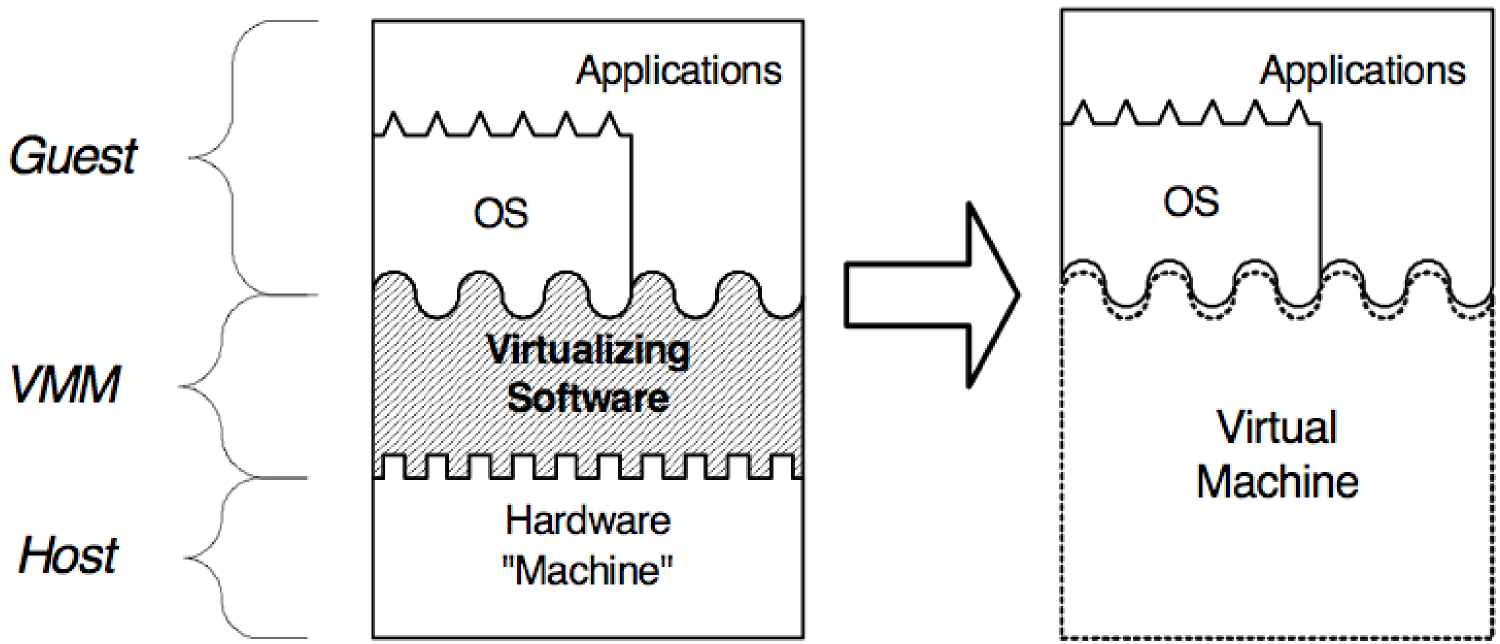
(b)

Virtual Machines to the Rescue

"A running program is often referred to as a virtual machine - a machine that doesn't exist as a matter of actual physical reality. The virtual machine idea is itself one of the most elegant in the history of technology and is a crucial step in the evolution of ideas about software. To come up with it, scientists and technologists had to recognize that a computer running a program isn't merely a washer doing laundry. A washer is a washer whatever clothes you put inside, but when you put a new program in a computer, it becomes a new machine.... The virtual machine: A way of understanding software that frees us to think of software design as machine design."

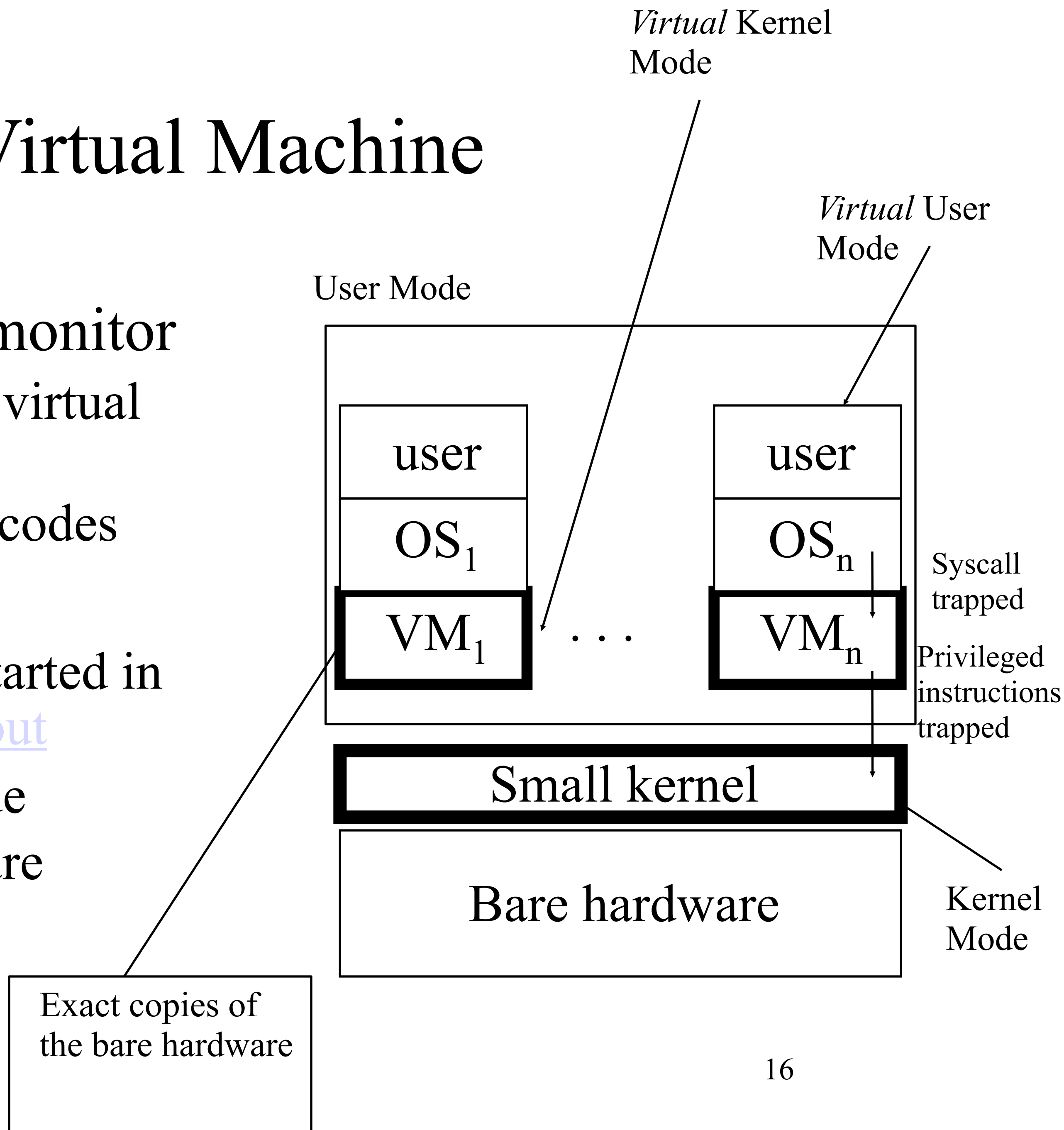
From David Gelernter's "Truth, Beauty, and the Virtual Machine," Discover Magazine, September 1997, p. 72.

What if we could do this

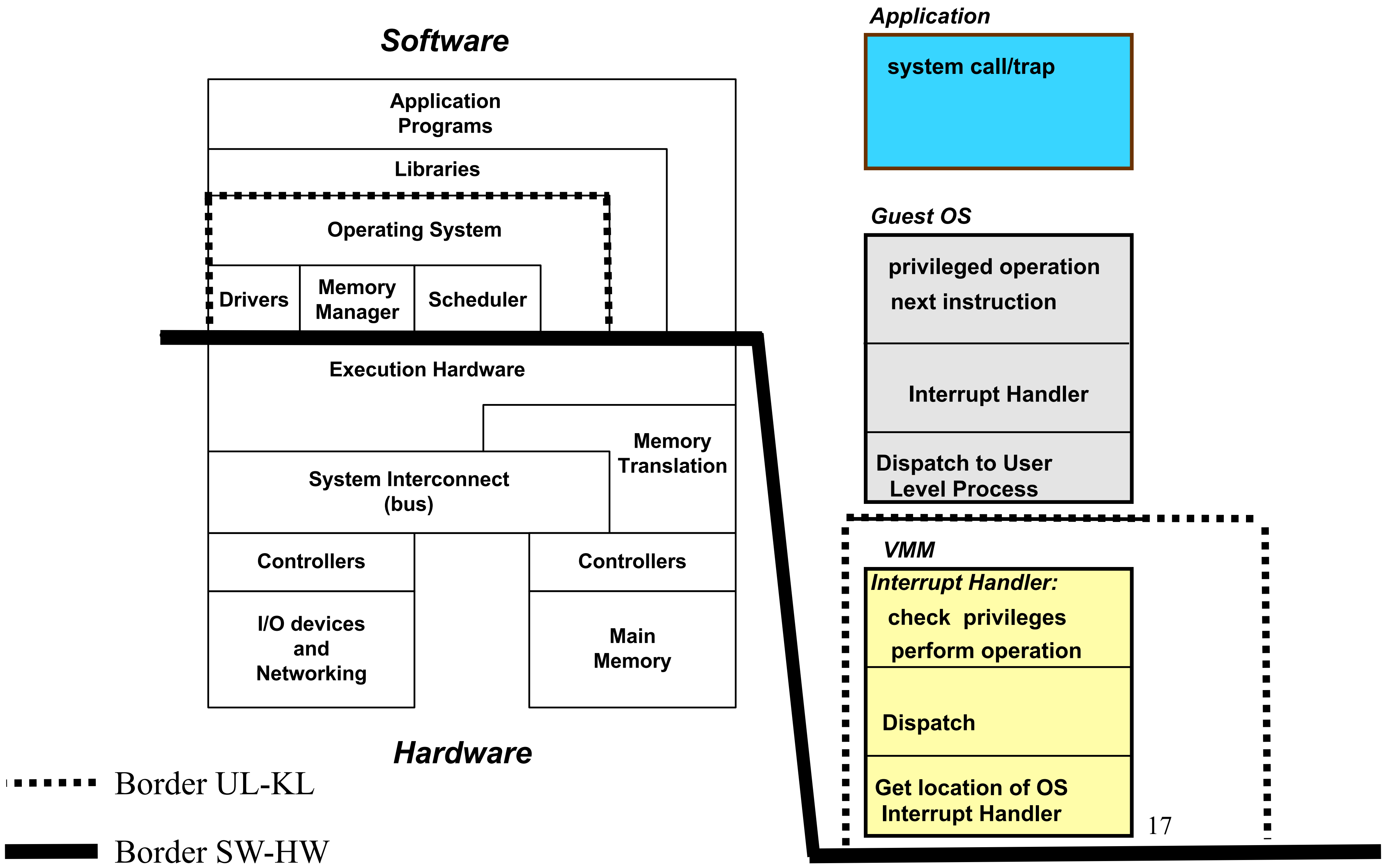


Virtual Machine

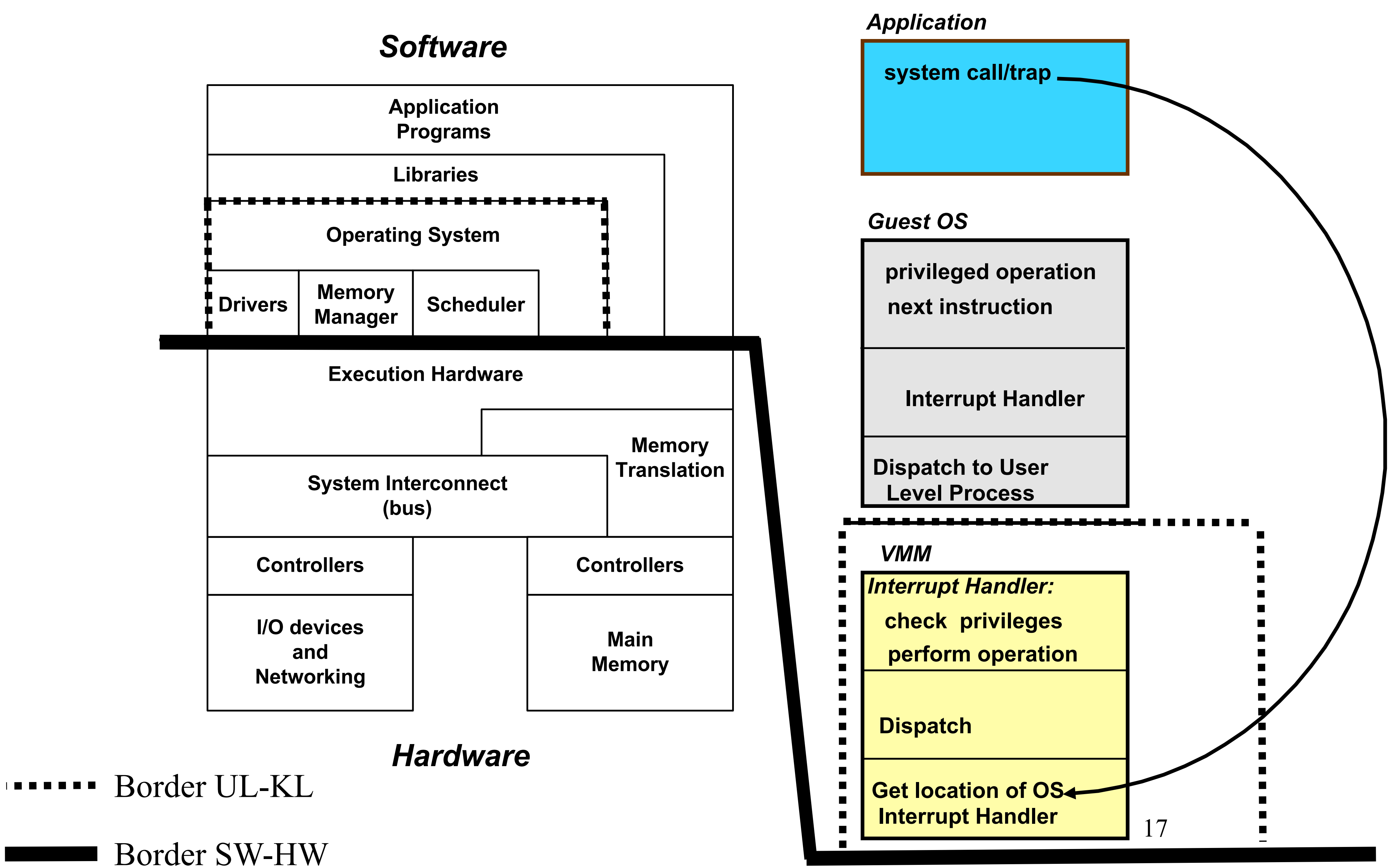
- Virtual machine monitor
  - provide multiple virtual “real” hardware
  - run different OS codes
- Example
  - IBM VM/370: Started in the 70’s. Check out
  - virtual 8086 mode
  - Java VM, VMware
  - Xen



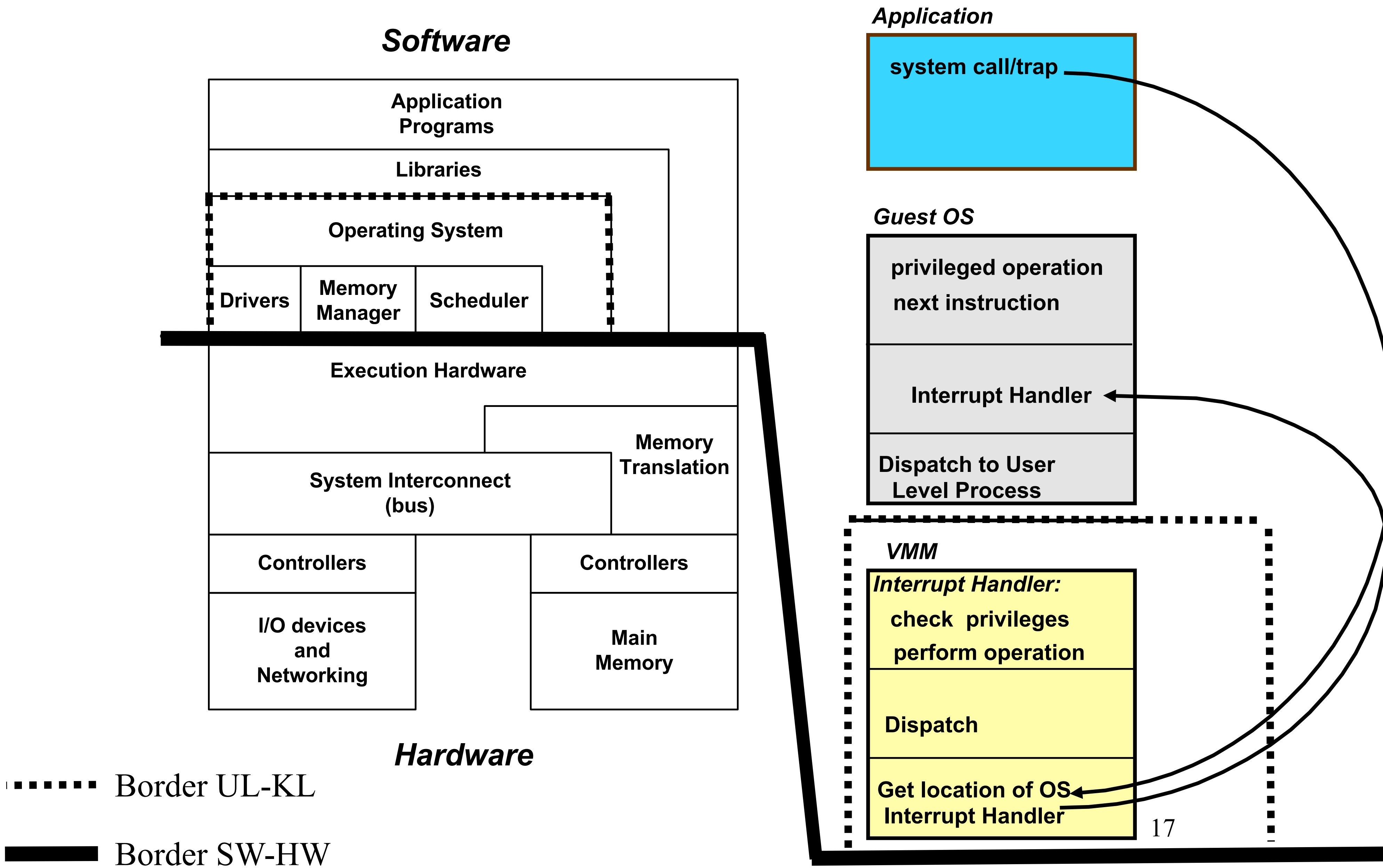
Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)



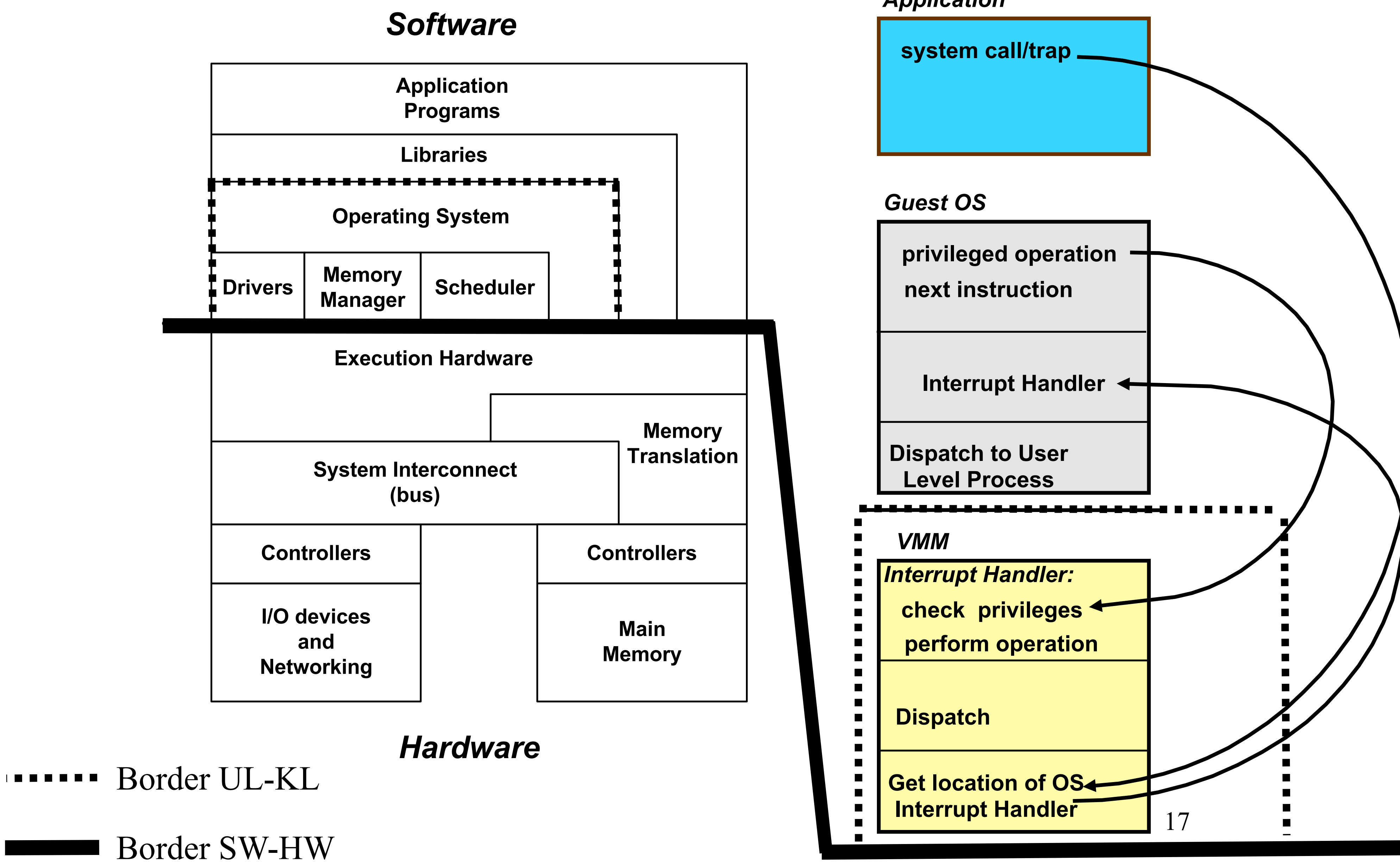
Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)



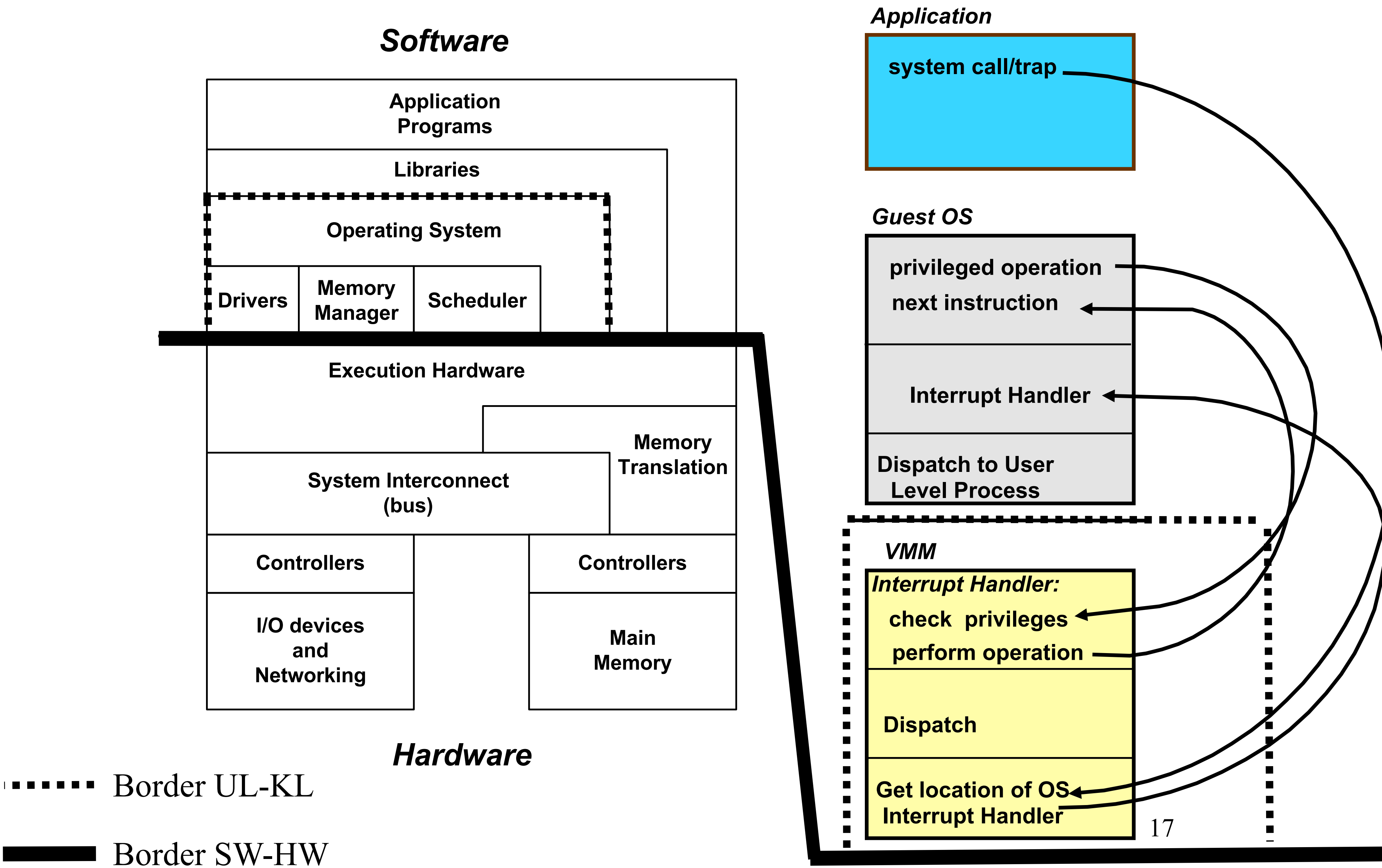
Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)



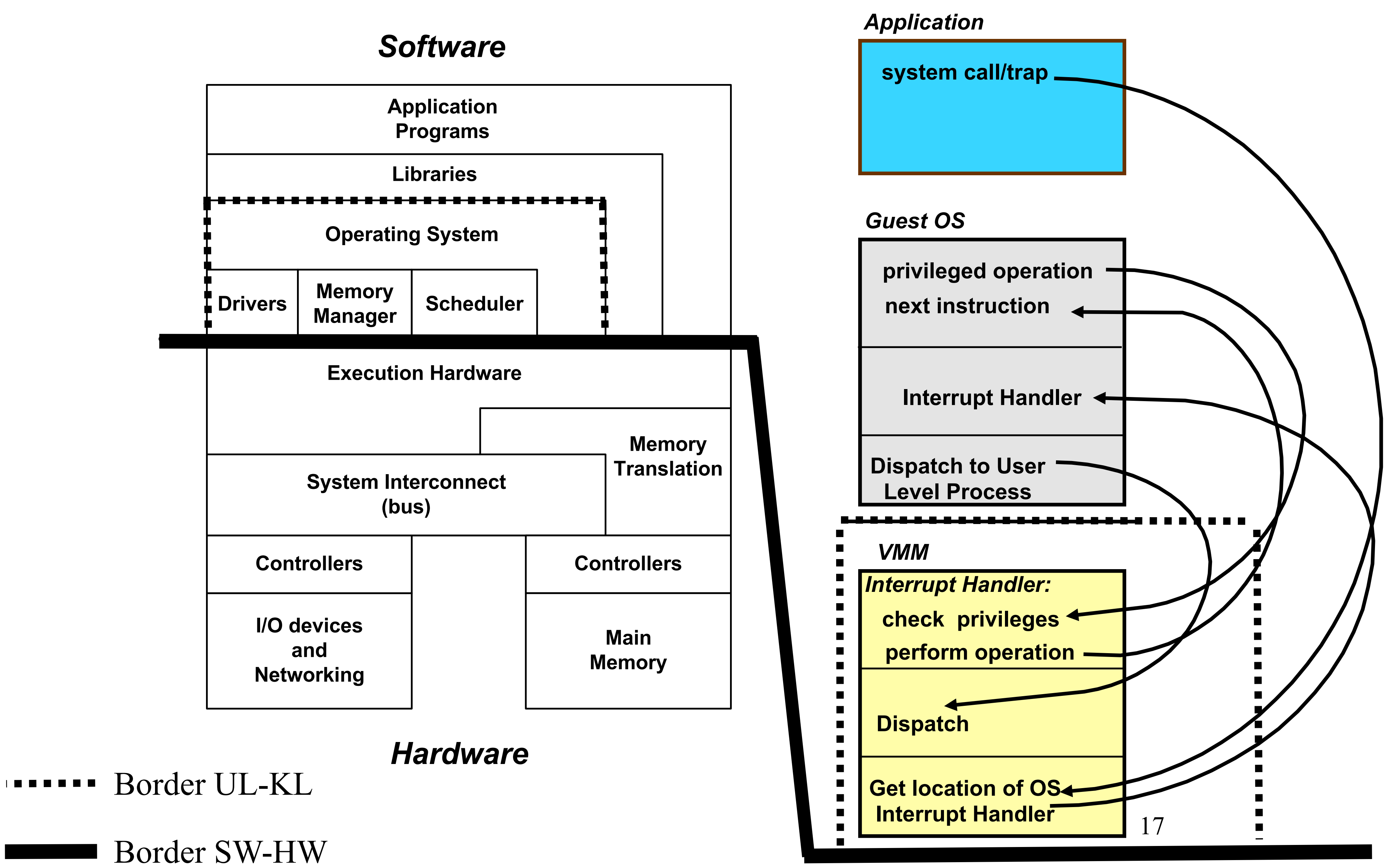
Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)



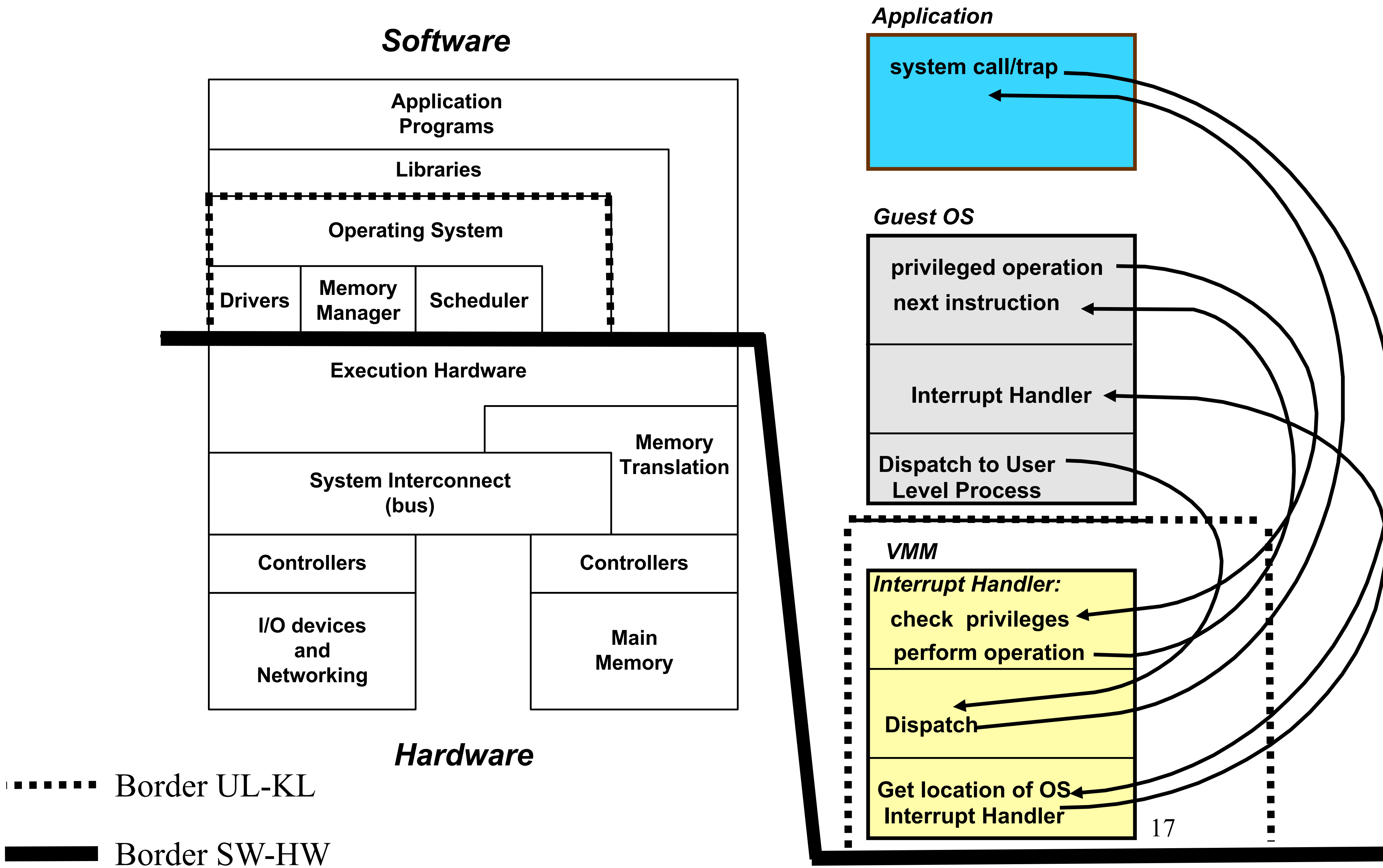
Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)



Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)



Adapted from J.E. Smith, 2006: Virtual Machine: Supporting Changing technology and New Applications (talk, U. of Wisconsin)



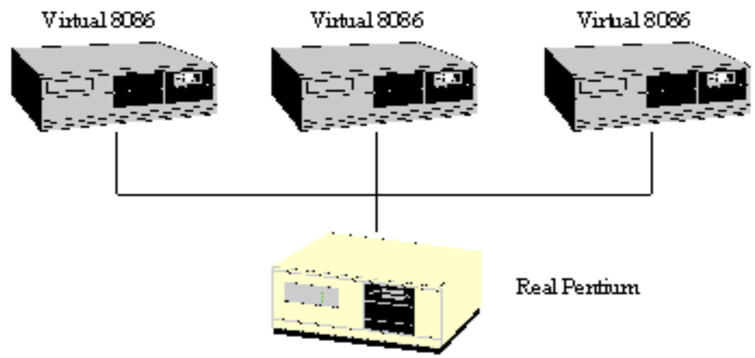
Old Virtual Machine Systems

- CMSCambridge Monitor System or Conversational Monitor System. Single User Interactive OS developed in conjunction with the Virtual Machine Control Program CP-40 at IBM Cambridge Laboratories. Later adapted for CP-67 and VM/370. Late 1960s [Meyer & Seawright 1970].
- CPControl Program. A component of VM/370 for the IBM/370. CP is the kernel which implements the virtual machine. Early 1970s.
- CP-40Virtual machine control program for a modified IBM 360/40. See also CMS. Mid 1960s [Goldberg 1974].
- CP-67Virtual machine control program for the IBM 360/67. Successor to CP-40. See also CMS. Late 1960s [Meyer & Seawright 1970].
- HITAC 8400 OSA Virtual machine system for the Hitac 8400 (RCA Spectra 70/45). Late 1960s [Goldberg 1974].
- IBM 360/30 OSVirtual machine for the IBM 360/30. Late 1960s [Goldberg 74].M44/44XVirtual machine system for modified IBM 7044. An early exploration of virtual machine ideas. Mid 1960s [Goldberg 1974, Belady et al 1981].
- Newcastle Recursive VMVirtual Machine system developed on a Burroughs 1700. Early 1970s [Goldberg 1974].
- PDP-10Virtual machine system for the PDP-10. Early 1970s [Goldberg 1974].
- UCLA VMVirtual machine system developed at UCLA for modified PDP-11/45 for data security studies. Early 1970s [Goldberg 1974].
- UMMPSVirtual machine system for the IBM 360/67. Early 1970s [Goldberg 1974].
- VM/370Virtual machine system for IBM 370. Successor to CP-67. See also CMS. First Release 1972 [IBMSJ 1979, Creasy 1981].
- VM/PCA version of VM/370 for the PC/370. Early 1980s [Daney & Foth 1984].
- VOSVirtual machine OS running on the Michigan Terminal System. Early 1970s [Srodowa & Bates 1973].

Figure 1. IBM System/360 Model 40 Data Processing System

Virtual 8086

A NEW OLD IDEA: PENTIUM VIRTUAL 8086 MODE



- Virtual 8086 mode on the Pentium makes it possible to run old 16-bit DOS applications on a virtual machine

Java VM

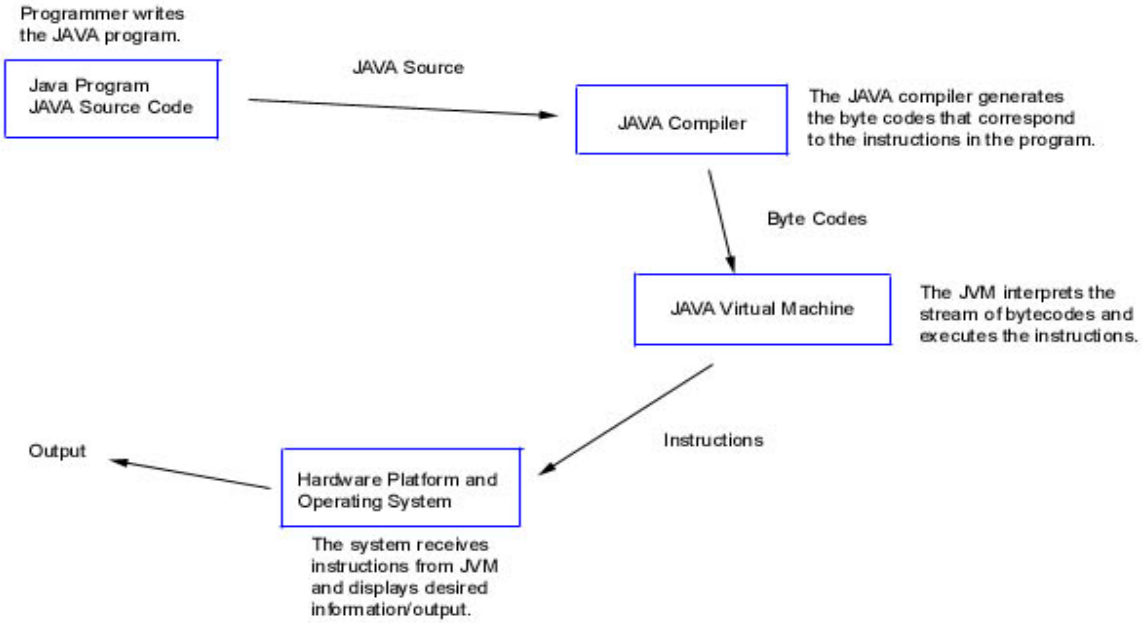


Figure 1.1: Diagram of Java Program Execution

Virtual Machine Hardware Support

- What is the minimal support?
  - 2 modes
  - Exception and interrupt trapping
- Can virtual machine be protected without such support?
  - Yes, emulation instead of executing on real machine

Pro et Contra

Monolithic	Layered	VM	C/S	Micro kernel
•Performance	•Clean, less bugs •Clear division of labour	•Many virtual computers with different OS'es •Test of new OS while production work continues •All in all: flexibility	•Clear division of labour	•More flexible •Small means less bugs +manageable •Distributed systems •Failure isolation of services at Kernel Level
•More unstructured	•Performance issues?	•Performance issues? •Complexity issues?	•Performance issues?	•Flexibility issues? •Performance issues?

Some Links

- Virtual machine
  - http://whatis.techtarget.com/definition/0,,sid9_gci213305,00.html
- Exokernel
  - http://pdos.lcs.mit.edu/exo/
- THE
  - http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD196.PDF
- L4
  - http://os.inf.tu-dresden.de/L4/
- VM
  - http://www.vm.ibm.com/