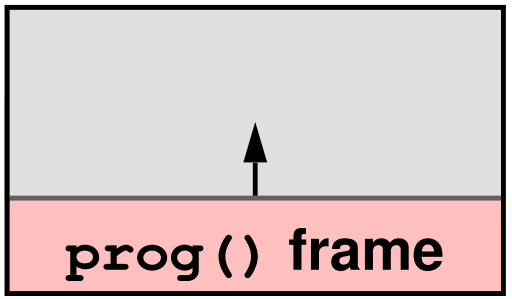
System Calls

- A *trap* is a type of *"software interrupt"*
  - interrupt handler will invoke trap handler

```
prog( ) {
   ...
   write(fd, buffer, size);
   ...
}
```

```
write( ) {
   ...
   trap(write_code);
   ...
}
```



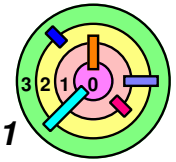
User Stack

User

Kernel

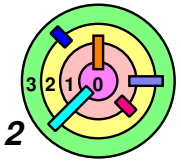
```
intr_handler(intr_code) {
   ...
   if (intr_code == SYSCALL)
     syscall_handler( );
   ...
}

syscall_handler(trap_code) {
   ...
   if (trap_code == write_code)
     write_handler( );
   ...
}
```

Kernel Stack

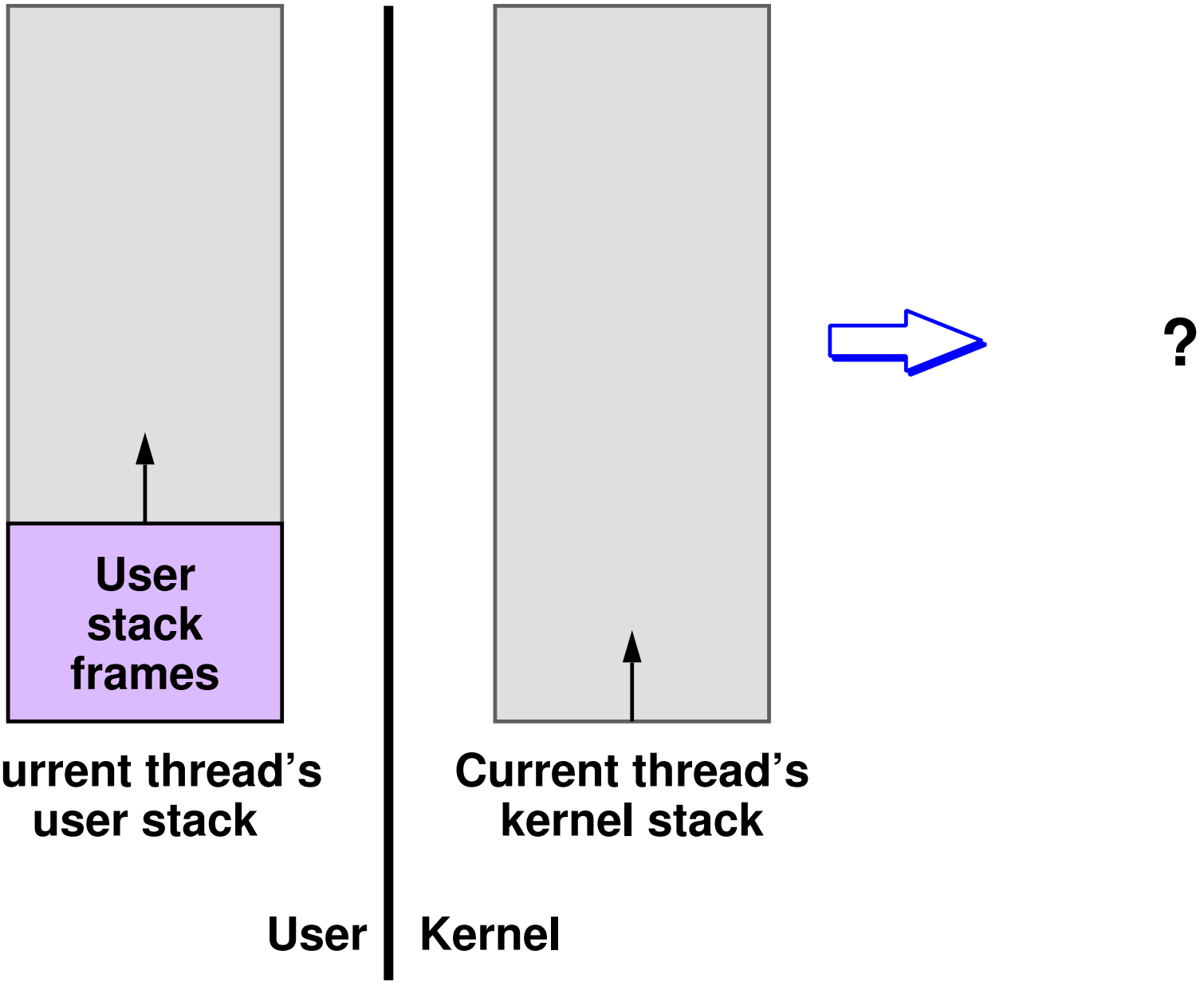


System Calls

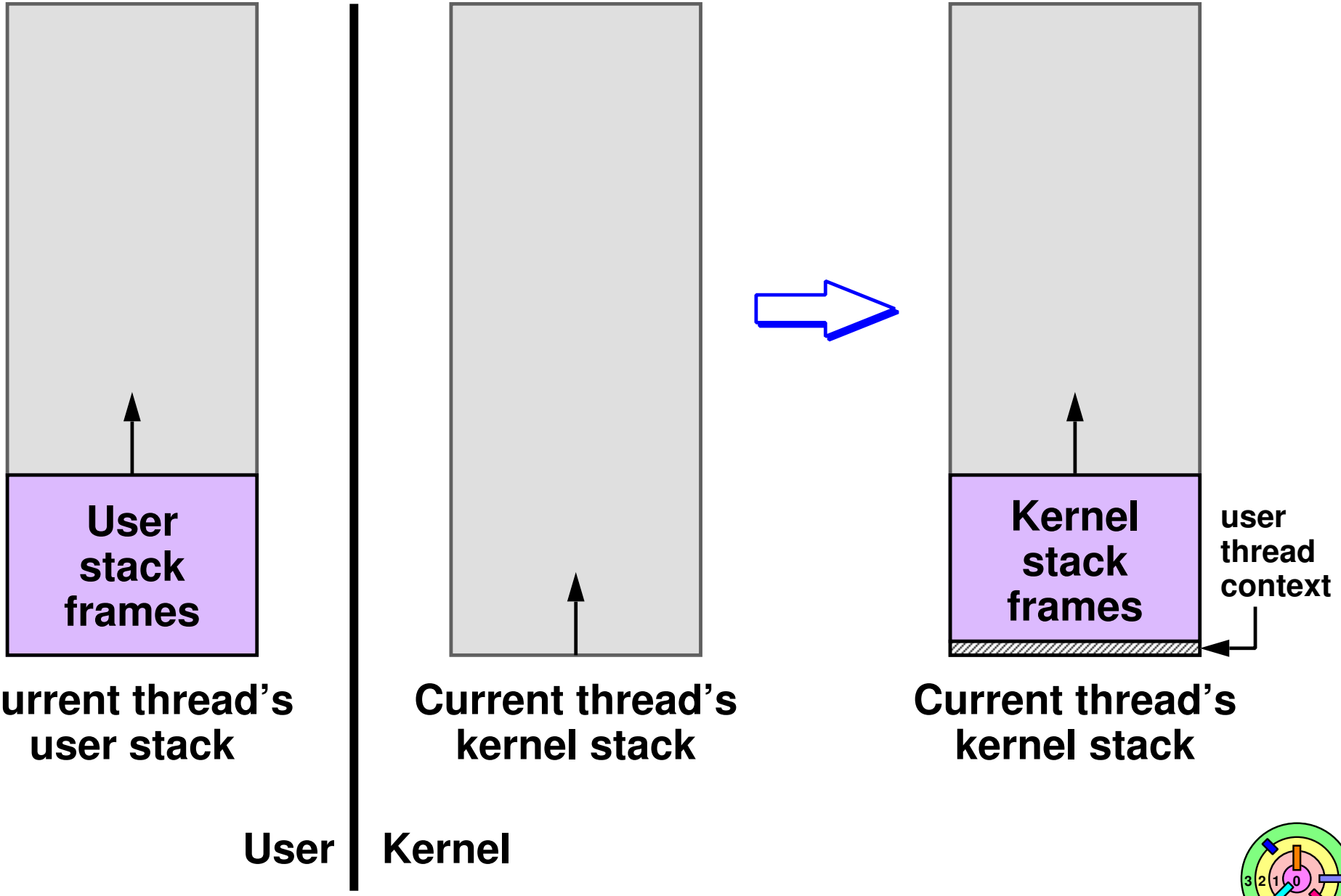
- More details on the *"trap"* machine instruction
  1) Trap into the kernel with all *interrupt disabled* and processor mode set to *privileged mode*
  2) The *Hardware Abstraction Layer (HAL)* save IP and SP in "temporary locations" in kernel space *(e.g., the interrupt stack)*
     - additional registers may be saved
     - *HAL* is *hardware-dependent* (outside the scope of this class)
  3) HAL sets the SP to point to the *kernel stack* designated for the corresponding user thread (information from PCB)
  4) HAL sets IP to *interrupt handler* (written in C)
     - copy user IP and SP from "temporary location" and push them onto kernel stack, then *re-enable interrupt*
  5) On return from the trap handler, disable interrupt and executes a special "return" instruction to *return to user process*
     - `iret` on x86
- Similar sequence happens when you get *hardware interrupt*



System Calls



System Calls



System Calls

- More details on the *"trap"* machine instruction

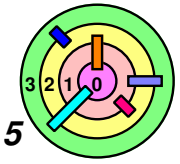
interrupt context

1) Trap into the kernel with all *interrupt disabled* and processor mode set to *privileged mode*
2) The *Hardware Abstraction Layer (HAL)* save IP and SP in "temporary locations" in kernel space *(e.g., the interrupt stack)*
   - additional registers may be saved
   - *HAL* is *hardware-dependent* (outside the scope of this class)
3) HAL sets the SP to point to the *kernel stack* designated for the corresponding user process (information from PCB)
4) HAL sets IP to *interrupt handler* (written in C)
   - copy user IP and SP from "temporary location" and push them onto kernel stack, then *re-enable interrupt*

interrupt context

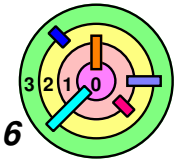
5) On return from the trap handler, disable interrupt and executes a special "return" instruction to *return to user process*
   - `iret` on x86

- Similar sequence happens when you get *hardware interrupt*



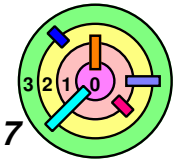
Context Switch

- The big idea here is that in order to perform a context switch, you must *save* your context, *build* new context, then *switch* to it
  - therefore, you must know what constitutes the context
  - then you save all of it
    - what's the *minimum* amount of context to save?
    - context can be stored in several places
      - stack
      - thread control block (e.g., in a system call, the TCB contains pointers to *both* the corresponding user stack frame and the kernel stack frame)
      - etc.
  - when switching back, you must restore the context
- In general, it's difficult to make a "clean" context switch
  - when you switch from context A to context B
    - there may be time you are in the context of *both* A and B
    - there may be time you are in *neither* contexts



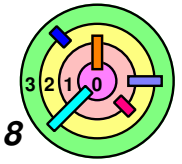
3.1 Context Switching

- Procedures
- Threads & Coroutines
- Systems Calls
- *Interrupts*



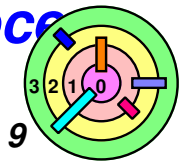
Interrupts

- Do not confuse *interrupts* with *signals* (even though the terminologies related to them are similar)
  - *signals* are *generated by the kernel*
    - they are delivered to the *user process*
    - *signal* ≠ *software interrupt*
  - *interrupts* are *generated by the hardware*
    - they are delivered to the *kernel*
      - they are delivered to HAL and then the kernel
- When an *interrupt* occurs, the processor puts aside the current context and switch to an *interrupt context*
  - the current context can be a *thread (user or kernel) context* or *another interrupt context*
    - need to be able to *mask/block individual interrupts* (similar to *signal* masking/blocking)
      - separate from enabling/disabling interrupt
  - when the interrupt handler finishes, the processor generally resumes the context that was interrupted



Interrupting A User Thread

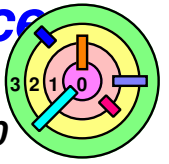
- If interrupt occurs when a *user thread* is executing in the CPU
  1) *Disable interrupt* and set processor mode to *privileged mode*
  2) The *Hardware Abstraction Layer (HAL)* save IP and SP in "temporary locations" in kernel space *(e.g., the interrupt stack)*
     - additional registers may be saved
     - *HAL* is *hardware-dependent* (outside the scope of this class)
  3) HAL sets the SP to point to the *kernel stack* designated for the corresponding user thread (information from PCB)
     - umm... which kernel stack?
  4) HAL sets IP to *interrupt handler* (written partly in C)
     - copy user IP and SP from "temporary location" and push them onto kernel stack, then *re-enable interrupt (with some interrupts blocked/masked)*
     - stay in *interrupt context*
  5) On return from the interrupt handler, disable interrupt and executes a special "return" instruction to *return to user proce*
     - `iret` on x86



Interrupting A User Thread

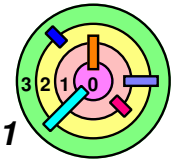
- If interrupt occurs when a *user thread* is executing in the CPU
  1) *Disable interrupt* and set processor mode to *privileged mode*
  2) The *Hardware Abstraction Layer (HAL)* save IP and SP in "temporary locations" in kernel space *(e.g., the interrupt stack)*
     - additional registers may be saved
     - *HAL* is *hardware-dependent* (outside the scope of this class)
  3) HAL sets the SP to point to the *kernel stack* designated for the corresponding user thread (information from PCB)
     - umm... which kernel stack?
  4) HAL sets IP to *interrupt handler* (written partly in C)
     - copy user IP and SP from "temporary location" and push them onto kernel stack, then *re-enable interrupt (with some interrupts blocked/masked)*
     - stay in *interrupt context*
  5) On return from the interrupt handler, disable interrupt and executes a special "return" instruction to *return to user proce*
     - `iret` on x86

ain
fference
ith
yscall



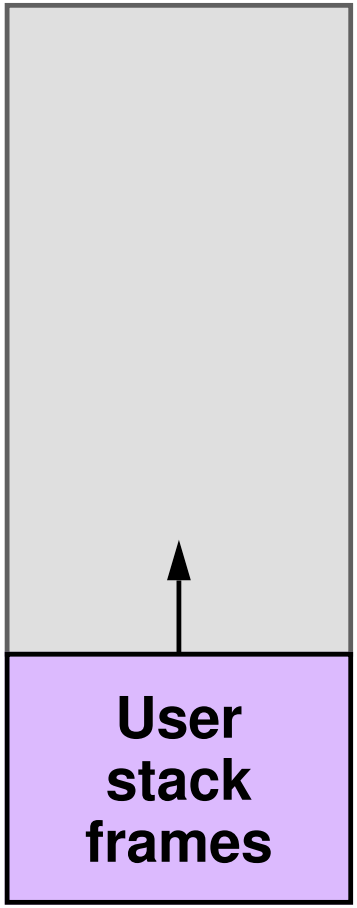
Interrupts

- *Interrupt service routine* is executed in an *interrupt context*
  - no thread context here
- Interrupt service routine is written in C (mixed with assembly code)
  - to execute C code, you need a stack
  - which stack should it use?
  - there are several possibilities
    1) allocate a new stack each time an interrupt occurs
       - too slow
    2) have one stack shared by all interrupt handlers
       - not often done
    3) interrupt handler could *borrow the kernel stack* from the thread it is interrupting
       - most common

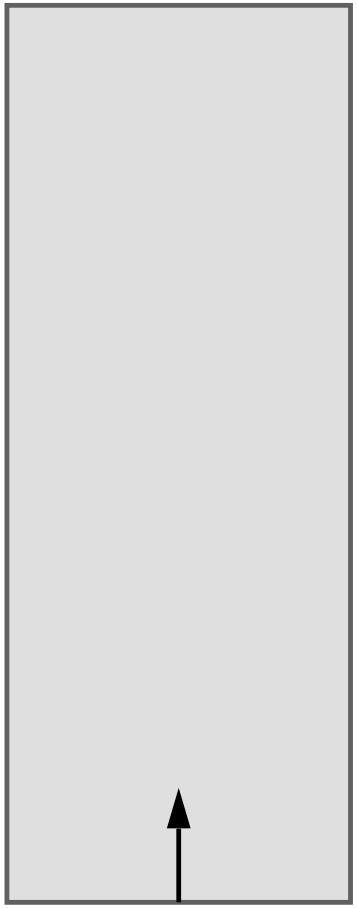


Interrupting User Thread

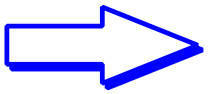
- similar to system call
  - except that you *stay in interrupt context* with some interrupts masked

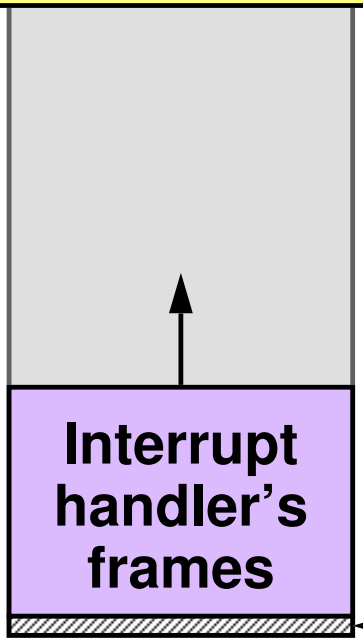


Current thread's user stack



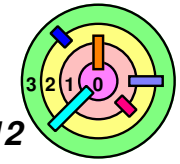
Current thread's kernel stack



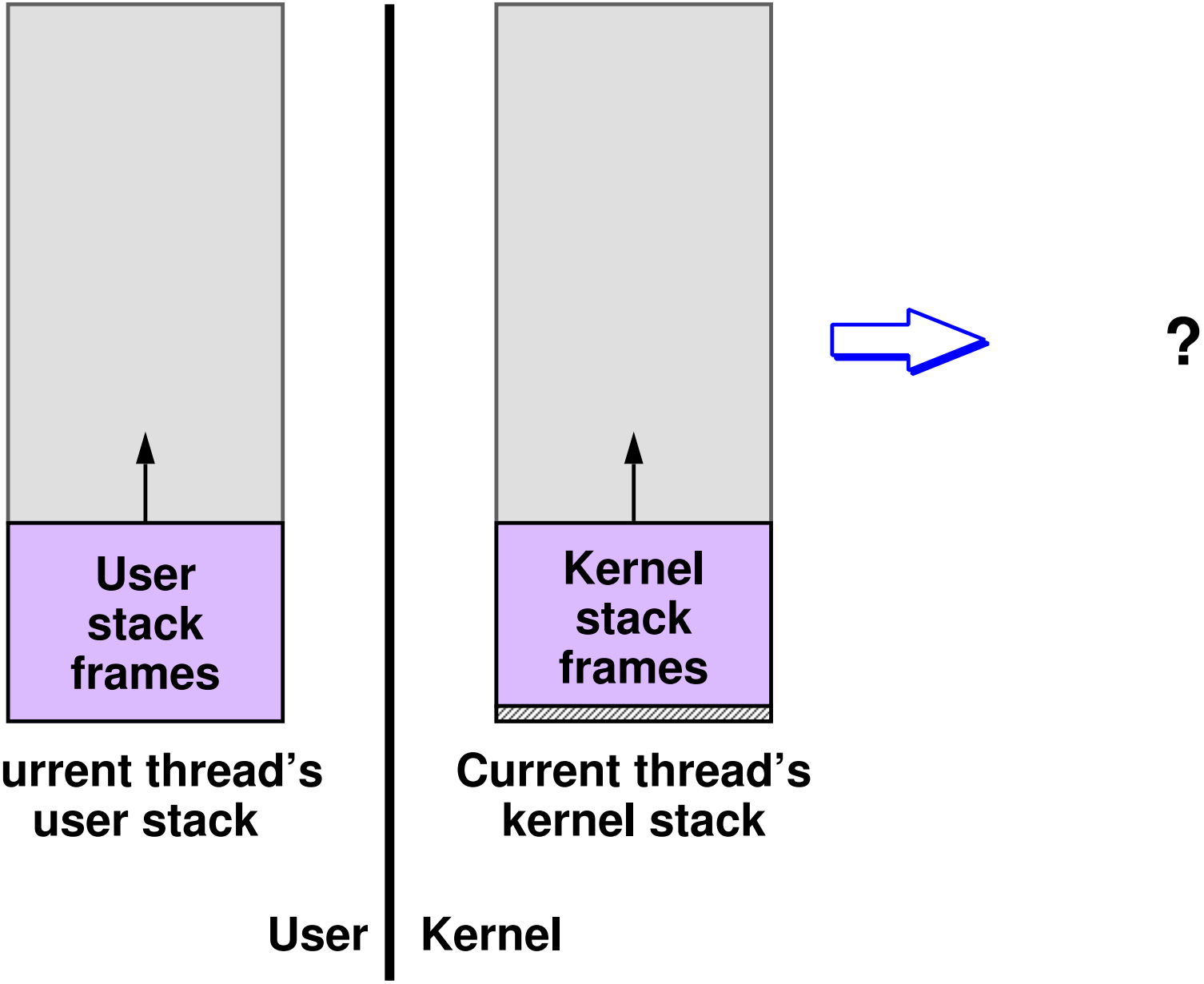


Current thread's kernel stack

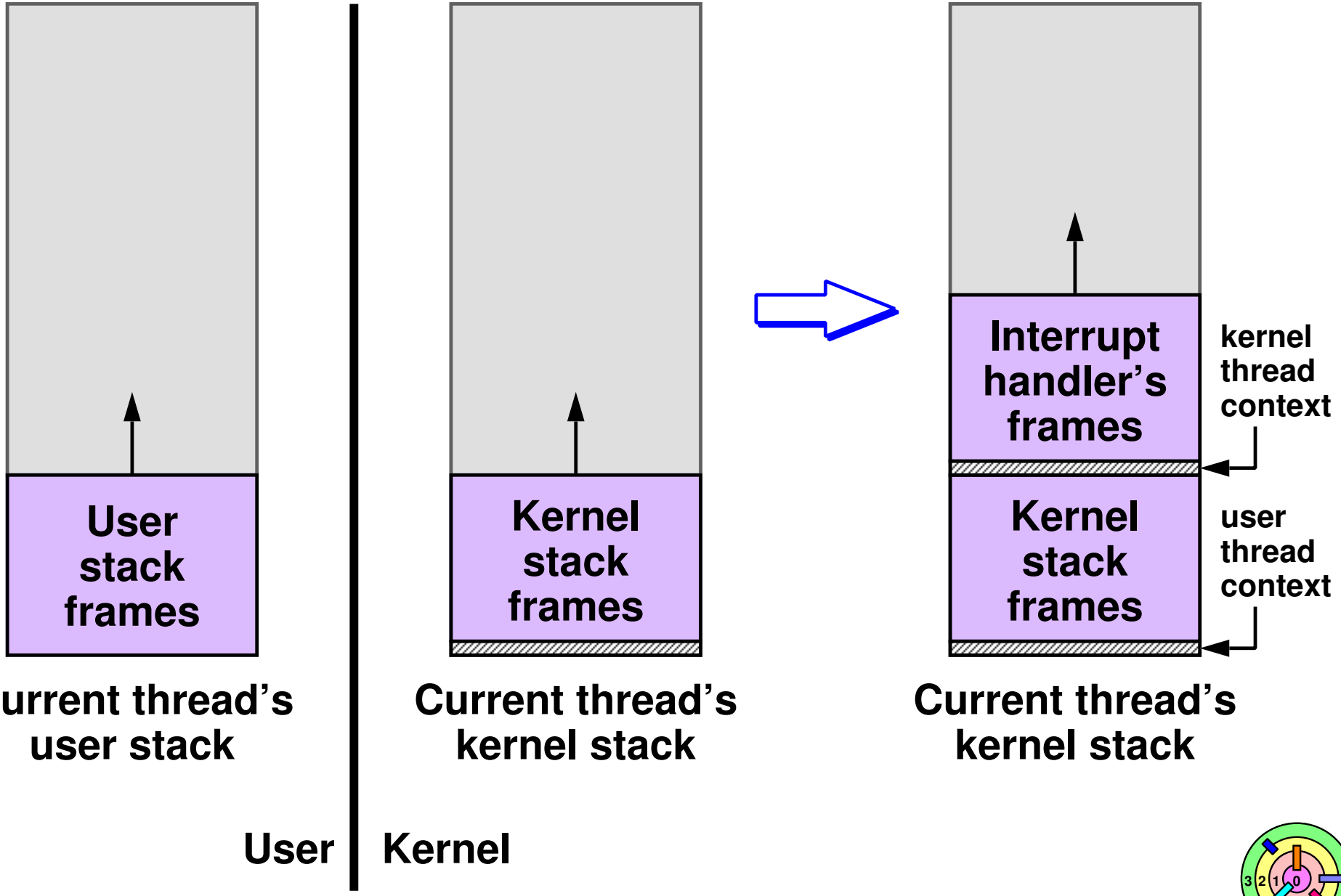
User | Kernel



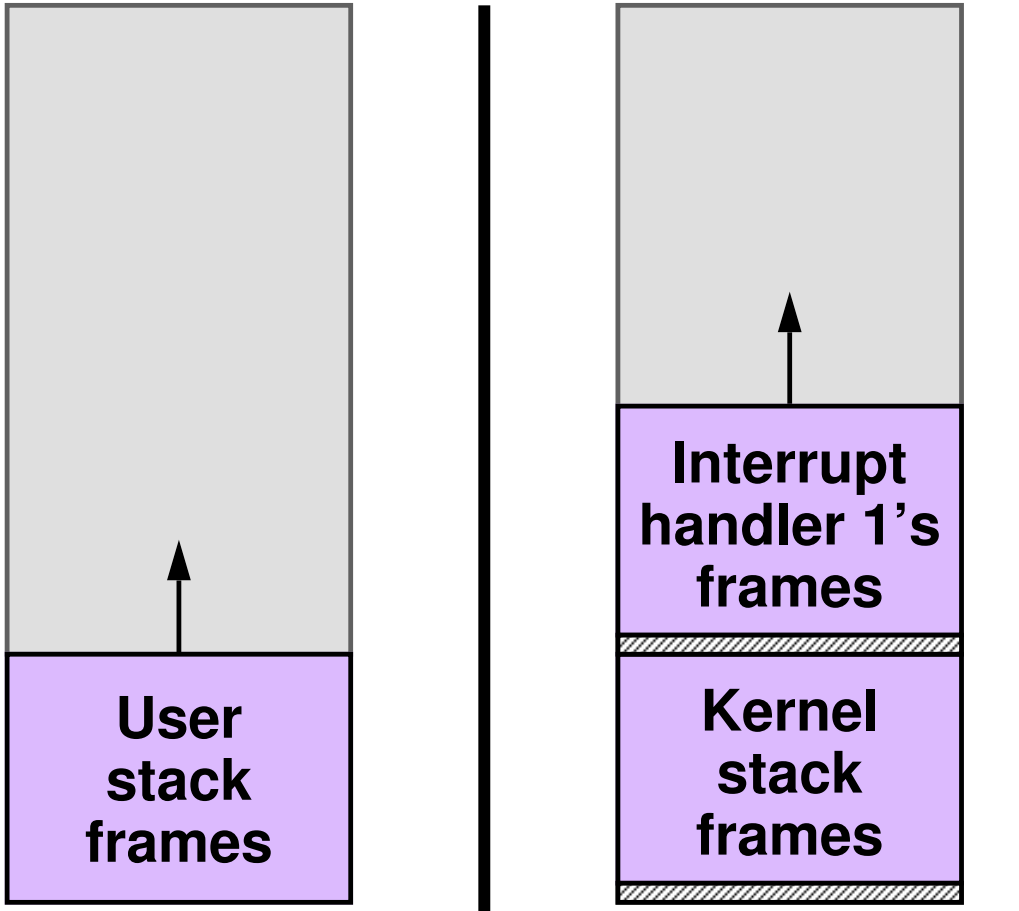
Interrupting Kernel Thread



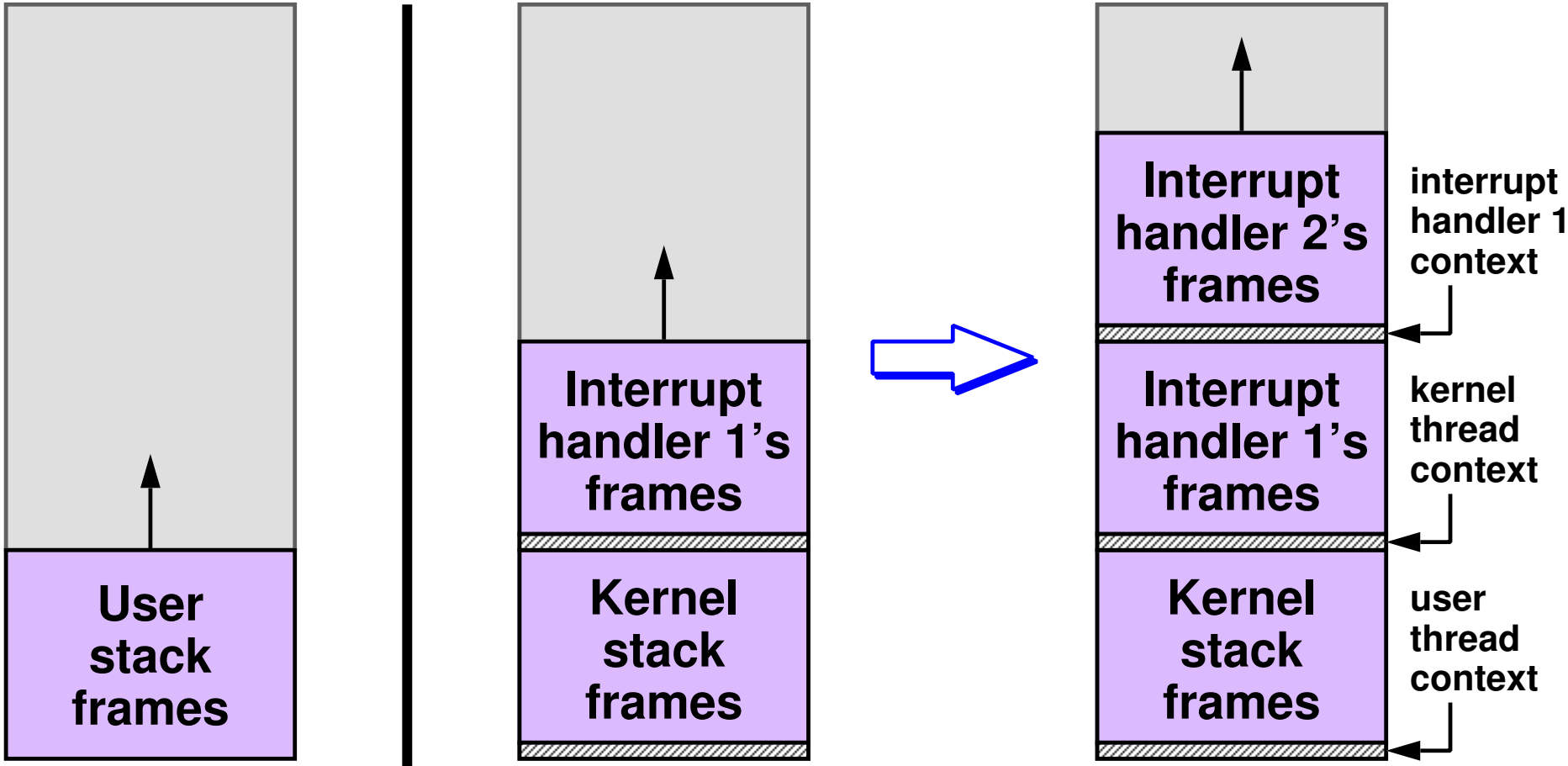
Interrupting Kernel Thread



Interrupting Another Interrupt Service Routine

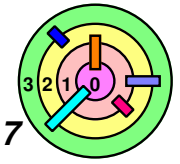


Interrupting Another Interrupt Service Routine



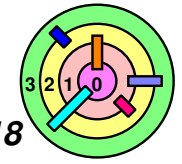
Interrupts

- For approaches (2) and (3), there is *no way* to suspend one interrupt handler and *resume* the execution of another
  - since there is only *one stack* for all the interrupt handlers
    - the global variable `CurrentThread` does not change!
  - therefore, the handler of the most recent interrupt must *run to completion*
    - when it's done, the stack frame is removed, and the next-most-recent interrupt now must run to completion
  - this is a *big deal!*
    - once you have interrupt handlers running, a normal thread (no matter how important it is) cannot run until *all* interrupt handlers complete
      - this is why an *interrupt service routine* should *do as little as possible* (and figure out a way to do the rest later)
    - if we have approach (1), then we won't have this problem
      - but it's so slow that it's unacceptable



Interrupts

- What if an interrupt service routine takes too long to run?
  - interrupt handler places a description of the work that must be done on a queue of some sort, then arranges for it to be done in some other context at a later time
    - still need to do something in the interrupt handler
      1) *unblock a kernel thread* that's sleeping in the corresponding I/O queue
      2) *start the next I/O opertion* on the same device
  - this approach is used in many systems, including Windows and Linux
    - will discuss further in Ch 5

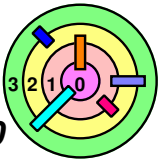


Interrupt Mask

- The CPU can have interrupt *disabled*
  - if any interrupt occurs while interrupt is disabled, the interrupt indication remains *pending*
  - once interrupt is *enabled*, a pending interrupt is *delivered* and the CPU is interrupted to execute a corresponding *interrupt service routine*
  - disable/enable *all* interrupts
- When interrupt is *enabled*, *individual* interrupt can be *masked*, i.e., temporarily *blocked*
  - similar to signal masking/blocking in user space programs
    - although you cannot "disable" all signals in user space
  - if an interrupt occurs while it is masked, the interrupt indication remains *pending*
  - once that interrupt is unmasked/unblocked, the interrupt is *delivered* and the CPU is interrupted to execute a corresponding *interrupt service routine (ISR)*

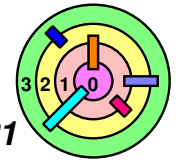
Interrupt Mask

- How interrupts are masked/blocked is architecture-dependent
  - common approaches
    1) hardware register implements a *bit vector / mask*
       - if a particular bit is set, the corresponding interrupt class is enable (or disabled)
       - the kernel masks interrupts by setting bits in the register
       - when an interrupt does occur, the corresponding mask bit is set in the register (block other interrupts of the same class)
       - cleared when the handler returns
    2) *hierarchical interrupt levels* (more common)

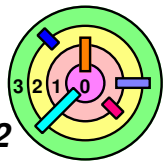


Interrupt Mask

- How interrupts are masked/blocked is architecture-dependent
  - common approaches
    1) hardware register implements a *bit vector / mask*
    2) *hierarchical interrupt levels* (more common)
       - the processor masks interrupts by setting an *Interrupt Priority Level (IPL)* in a hardware register
       - all interrupts with the *current or lower levels* are masked
       - the kernel masks a class of interrupts by setting the IPL to a particular value
       - when an interrupt does occur, the current IPL is set to that of the level the interrupt belongs
       - restores to previous value on handler return
  - even if (1) is implement in the hardware, abstraction (in HAL) can be built to make it look as if (2) is implemented
    - easier for kernel programmers to work with interrupts
    - for this class, we assume that (2) is used

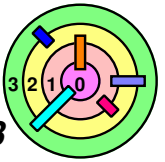


3.2 Input/Output Architectures



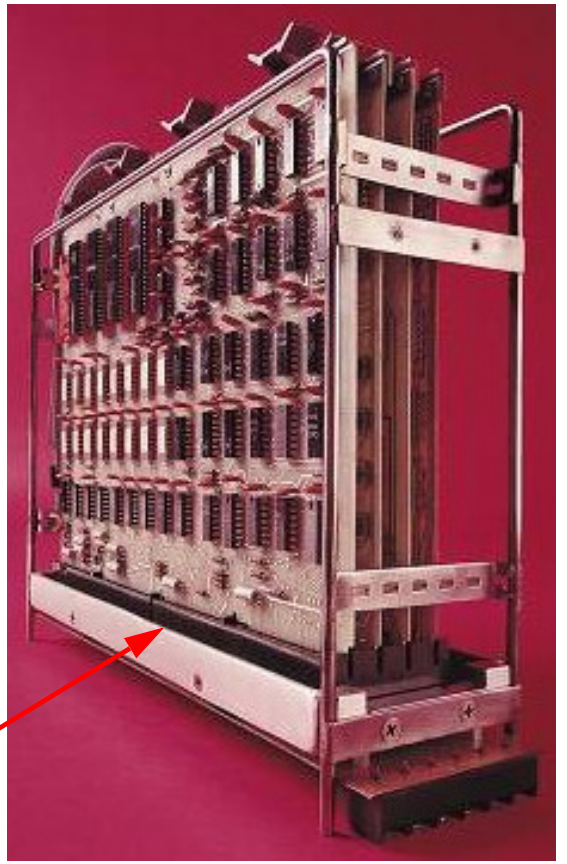
Input/Output

- Architectural concerns
  - memory-mapped I/O
    - programmed I/O (PIO)
    - direct memory access (DMA)
  - I/O processors (channels)
- Software concerns
  - device drivers
  - concurrency of I/O and computation



What Does A Computer Look Like?

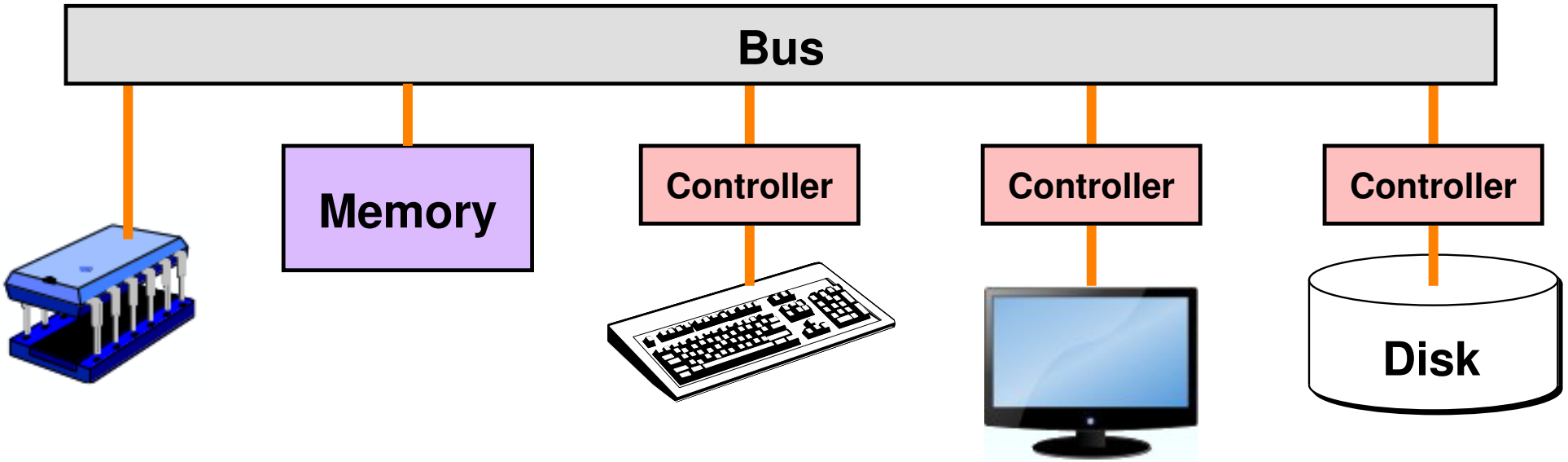
- LSI-11
  - processor for PDP-11
- Boards are connected over a "bus"
  - on the "backplane"
  - various standards for PDP-11
    - Unibus, Q-Bus, etc.



connect to backplane bus

http://hampage.hu/pdp-11/lsi11.html

Simple I/O Architecture



- memory-mapped I/O
  - all controllers listen on the bus to determine if a request is for itself or not
  - I/O controllers "process" the bus request
    - and respond to relatively few addresses
    - if no one responds, you get a "bus error"
  - memory is not really a "device"