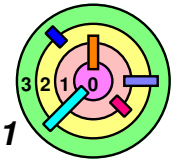
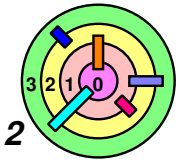
Housekeeping (Lecture 12 - 7/10/2025)

- **PA4 is due at 11:45pm on Tuesday, 7/15/2025**
  - **if you have code from current or a previous semester, *do not look at/copy/share* any code from it**
    - **it's best if you just get rid of it**
  - **if you include files that's not part of the original "`make pa4-submit`" command, the grader will delete them**
  - **PA4 is probably the most difficult assignment**
    - **don't wait too long to start this assignment**
    - **the last part (graph) may be time-consuming**
      - **unless you have found a tool that works, you might want to leave that part until the very end so you don't have to re-do the graphs**
    - **when a thread gives up the CPU volentarily, you get to choose to keep the same priority when it get on the ready list next time (or always use the highest priority next time around)**

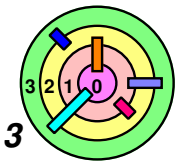


Housekeeping (Lecture 12 - 7/10/2025)

- *Grading guidelines* is the *ONLY* way we will grade and we can only grade on a *standard 32-bit Ubuntu Linux 16.04 inside VirtualBox/UTM* or on AWS Free Tier
  - although not recommended, you can do your development on a different platform
    - you must test your code on a "standard" platform since it's the *only* platform the grader is allowed to grade on
- If you make a submission, read and understand the *ticket* in the *web page* and save the web page as PDF as a record of your submission
  - make sure to *"Verify Your Ticket"* and *"Verify Your Submission"*

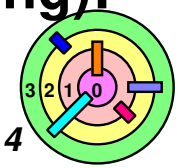


(7.4) Real-Time Scheduling (cont...)

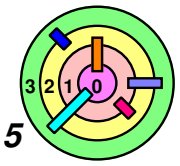


Earliest Deadline First (EDF)

- For scheduling tasks with deadlines, EDF can be proven to be an optimal scheduling policy under certain assumptions
  - but be careful when you specify deadlines
- Ex: task A must be completed 12ms from now and task B must be completed 10ms from now
  - according to EDF, run task B first
  - as it turns out, task A has two parts, it needs 1ms of computation, followed by 10ms of I/O
  - task B only needs to compute
  - if schedule task A first, both tasks can complete before their respective deadlines
  - if schedule task B first, task A will miss the deadline
  - what went wrong?
    - task A's real deadline on the processor is 2ms from now (we are looking at processor scheduling and not I/O scheduling)!

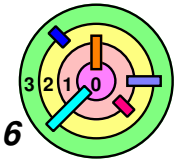


(7.5) Queueing Theory

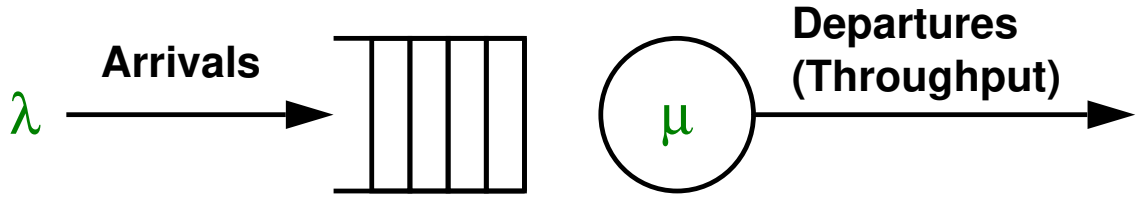


Queueing Theory

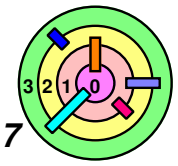
- Can we predict what will hapen to user performance:
  - if a service becomes more popular?
  - if we buy more hardware?
  - if we change the implementation to provide more features?
  - the answer is yes *if* we are allowed to *make certain assumptions* (even though these assumptions may not be realistic)
    - why make unrealistic assumptions if you know they are unrealistic?
      - if you make the right kind of assumptions, it may help you to understand the underlying system so you can make educated guesses



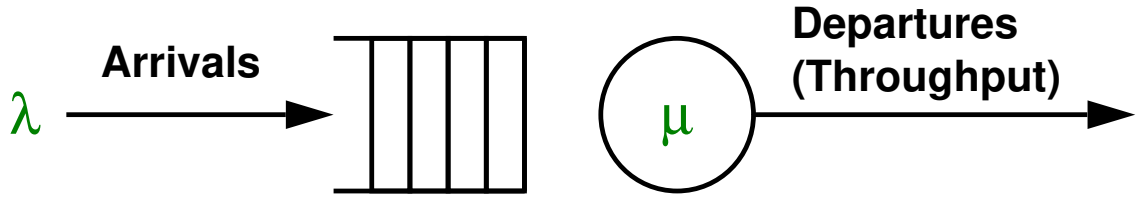
Queueing Model



- **Assumption: average performance in a stable system, where the arrival rate *(λ tasks/sec)* matches the departure rate**
  - **μ is the service rate (tasks/sec)**
    - **λ does not have to equal μ since the server can go idle**



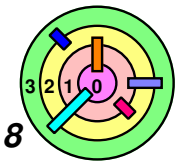
Definitions



- ***Queueing delay (W):*** **wait time**
  - ***Q*** **is the number of tasks queued**
- ***Service time (S):*** **time to service the request**
- ***Response time (R):*** $R = W + S$
- ***Utilization (U):*** **fraction of time the server is busy**

  $U = \lambda / \mu$ if $\lambda < \mu$ (not overloaded)

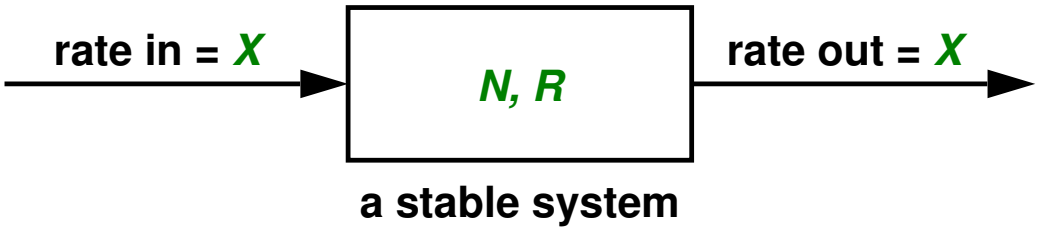
  $= 1$ if $\lambda \geq \mu$ (overloaded)
- ***Throughput (X):*** **rate of task completions,** $X = U \times \mu$
  - **if no overload,** $X = \lambda$
  - **if overload,** $X = \mu$
- ***N:*** **number of tasks in the system,** $N = Q + U$



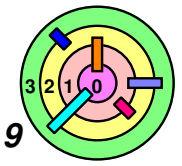
Little's Law

$$N = X \times R$$

- ***N:*** **number of tasks in the system**
  - **applieds to any *stable system*, where arrivals match departures (what goes in must comes out)**

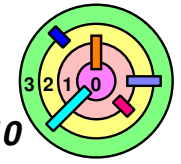


a stable system



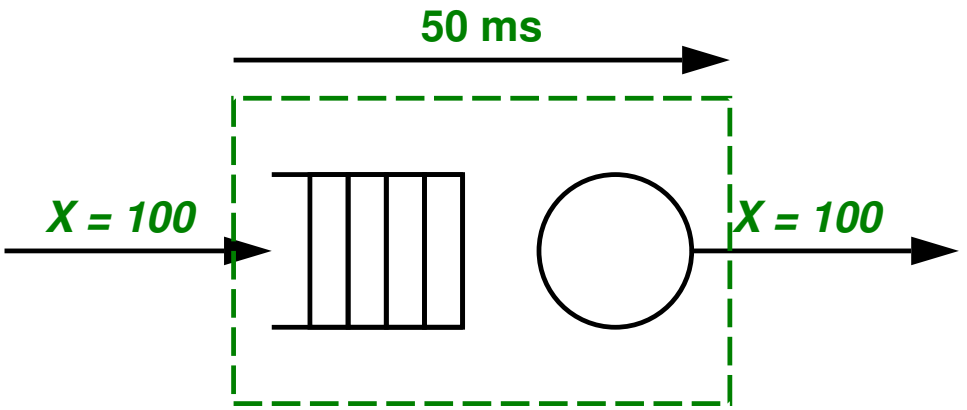
Examples

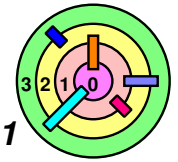
- Suppose a system has throughput *X = 100* tasks/s, average response time *R = 50* ms/task (or μ *= 200* tasks/s)
  - how many tasks are in the system on the average?
  - if the server takes 5 ms/task, what is its utilization?
  - what is the average wait time?
  - what is the average number of queued tasks?



Examples

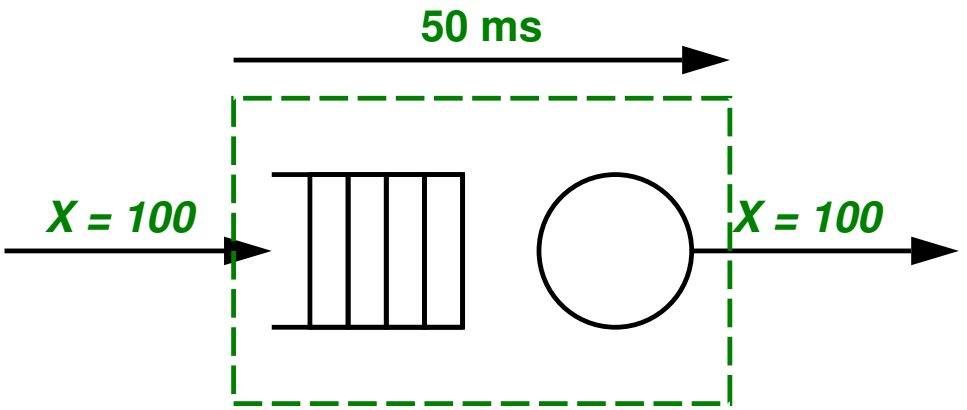
- Suppose a system has throughput *X = 100* tasks/s, average response time *R = 50* ms/task (or μ *= 200* tasks/s)
  - how many tasks are in the system on the average?
  - if the server takes 5 ms/task, what is its utilization?
  - what is the average wait time?
  - what is the average number of queued tasks?





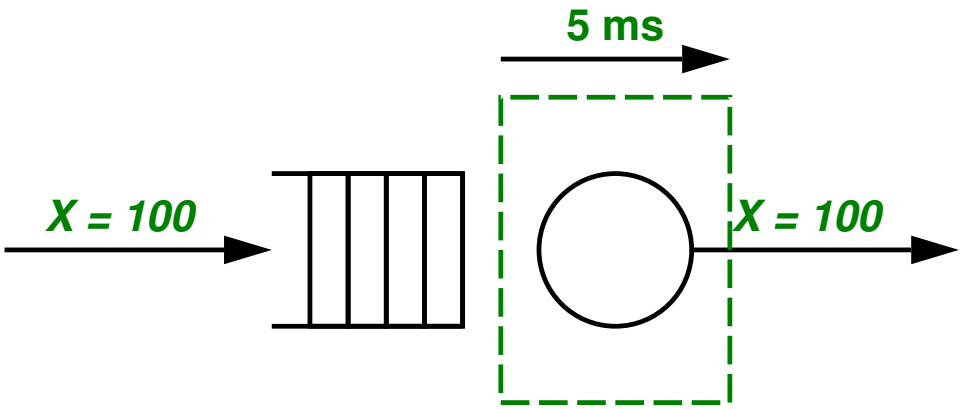
Examples

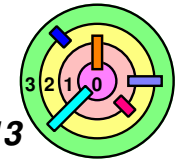
- Suppose a system has throughput *X = 100* tasks/s, average response time *R = 50* ms/task (or μ *= 200* tasks/s)
  - how many tasks are in the system on the average?
    - *N = X* $\times$ *R = 5* tasks
  - if the server takes 5 ms/task, what is its utilization?
  - what is the average wait time?
  - what is the average number of queued tasks?



Examples

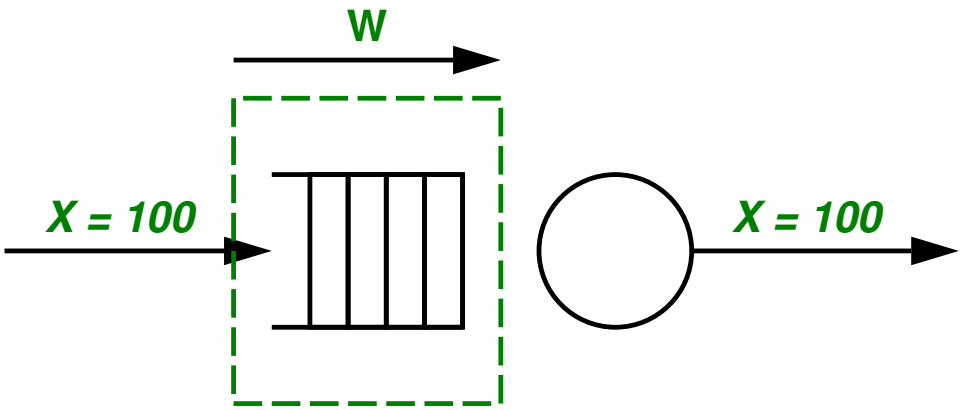
- Suppose a system has throughput *X = 100* tasks/s, average response time *R = 50* ms/task (or μ *= 200* tasks/s)
  - how many tasks are in the system on the average?
    - *N = X* $\times$ *R = 5* tasks
  - if the server takes 5 ms/task, what is its utilization?
    - *U = X* $\times$ *S = 0.5* task = *0.5*
  - what is the average wait time?
  - what is the average number of queued tasks?

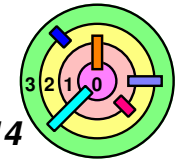




Examples

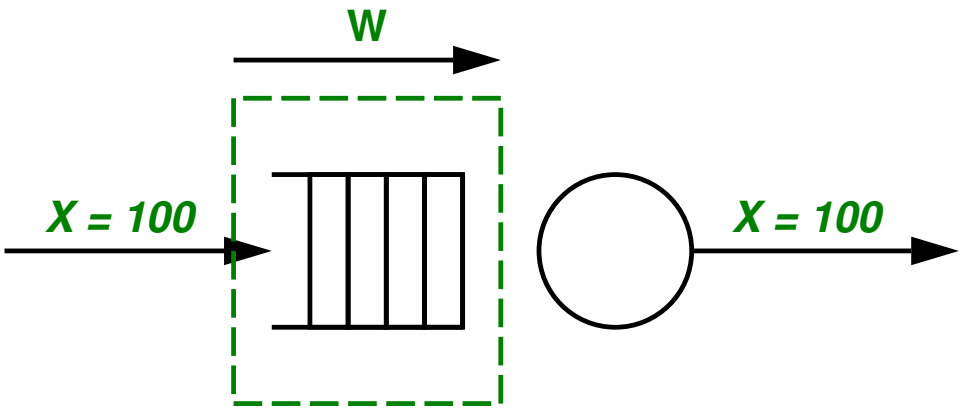
- Suppose a system has throughput *X = 100* tasks/s, average response time *R = 50* ms/task (or μ *= 200* tasks/s)
  - how many tasks are in the system on the average?
    - *N = X* $\times$ *R = 5* tasks
  - if the server takes 5 ms/task, what is its utilization?
    - *U = X* $\times$ *S = 0.5* task = *0.5*
  - what is the average wait time?
    - *W = R - S = 45* ms/task
  - what is the average number of queued tasks?

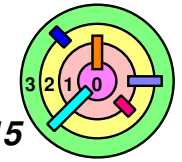




Examples

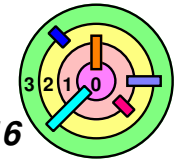
- Suppose a system has throughput *X = 100* tasks/s, average response time *R = 50* ms/task (or μ *= 200* tasks/s)
  - how many tasks are in the system on the average?
    - *N = X* $\times$ *R = 5* tasks
  - if the server takes 5 ms/task, what is its utilization?
    - *U = X* $\times$ *S = 0.5* task = *0.5*
  - what is the average wait time?
    - *W = R - S = 45* ms/task
  - what is the average number of queued tasks?
    - *Q = X* $\times$ *W = 4.5* tasks





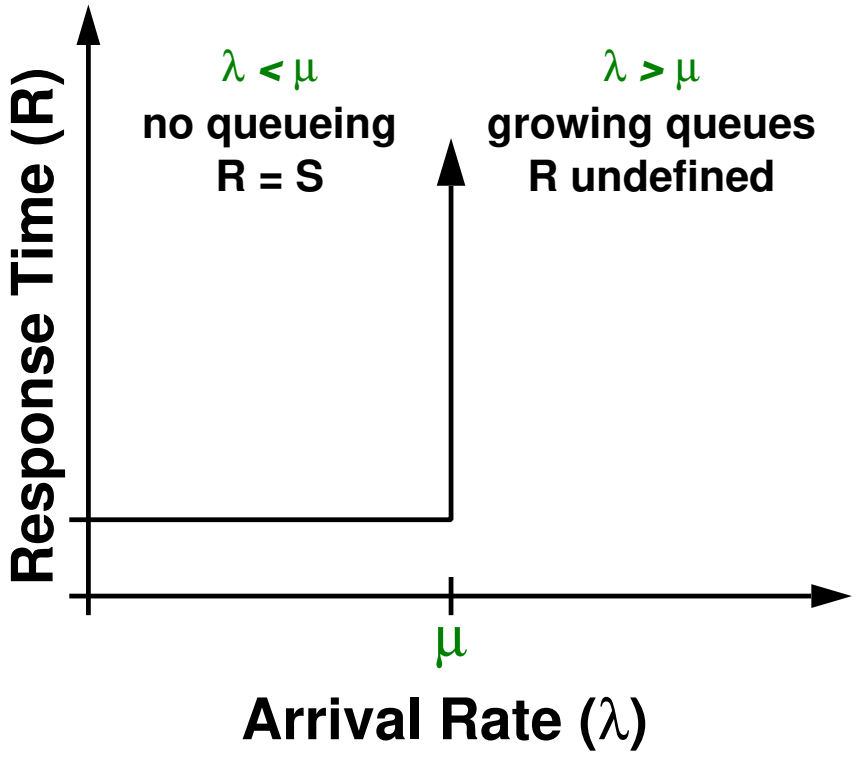
Utilization

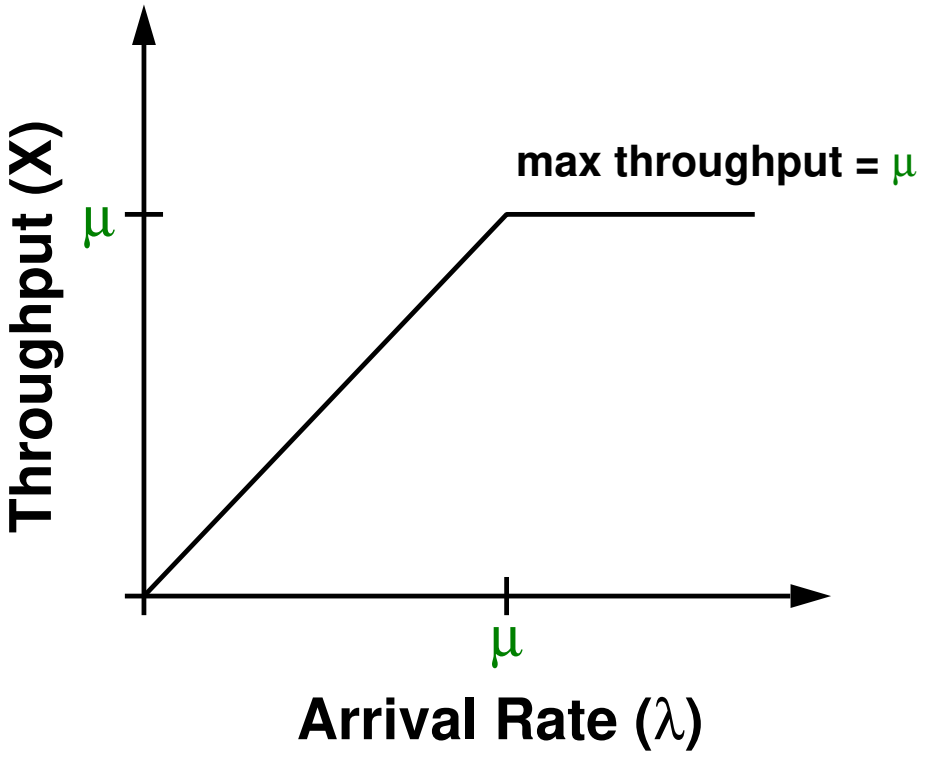
- In general, we want to maximize utilization (so that resources are not idling)
  - low utilization typically implies that resouces are wasted
  - but high utilization typically implies higher queueing delay and higher response times
  - operating at high utilization also increases the risk of overload
  - the above claims are not so obvious because we are used to thinking about a deterministic system (i.e., a conveyor belt) and not a ***stochastic*** system (where queueing theory is needed)
- 50 years ago, computers are very expensive, people can wait for computers
  - now that computers are much cheaper, it's okay to make the computer wait



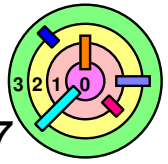
Response Time As A Function Of Arrival Rate

- Assuming $\lambda < \mu$, best response time scenario: ***evenly spaced arrivals***
  (fixed-sized tasks that arrive equally spaced from on another)
  - result is that there is no waiting/queueing (i.e., conveyor belt)



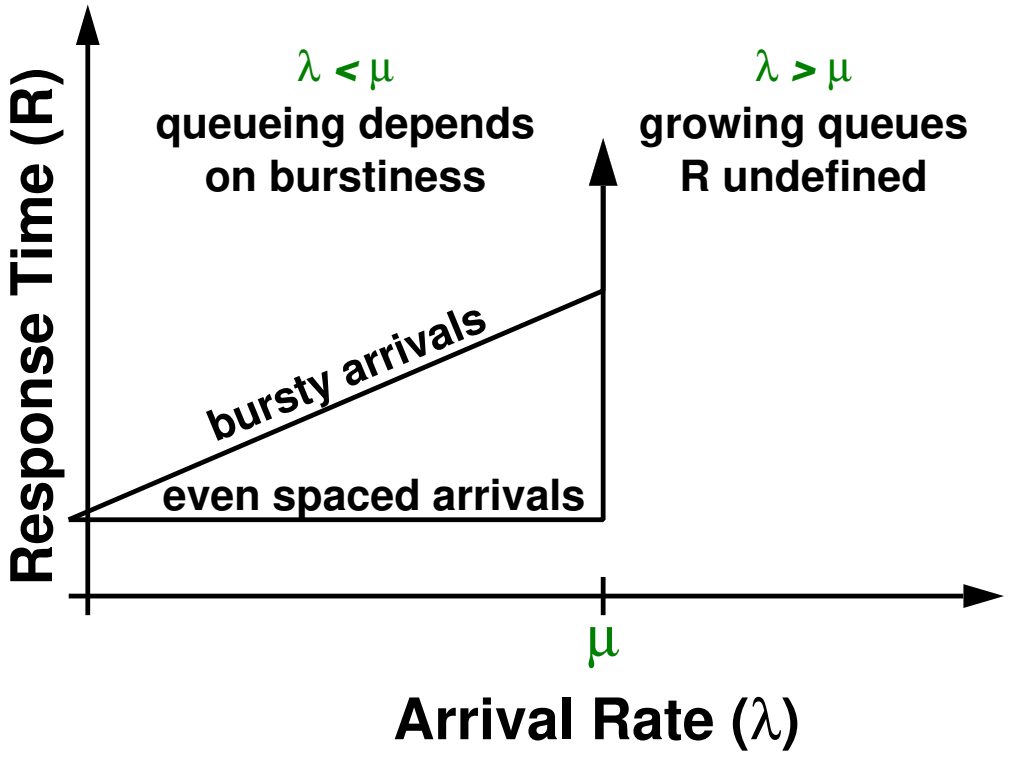


- Can R really go to infinity?
  - you will run out of memory and new arrivals will be dropped
    - analysis is a lot more complicated



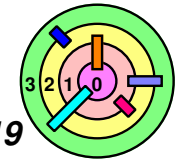
Response Time As A Function Of Arrival Rate

- Assuming $\lambda < \mu$, worst response time scenario: ***burst arrivals***
  (a group of tasks arrive at the same time)
  - one task in a group can be served right away, others must wait



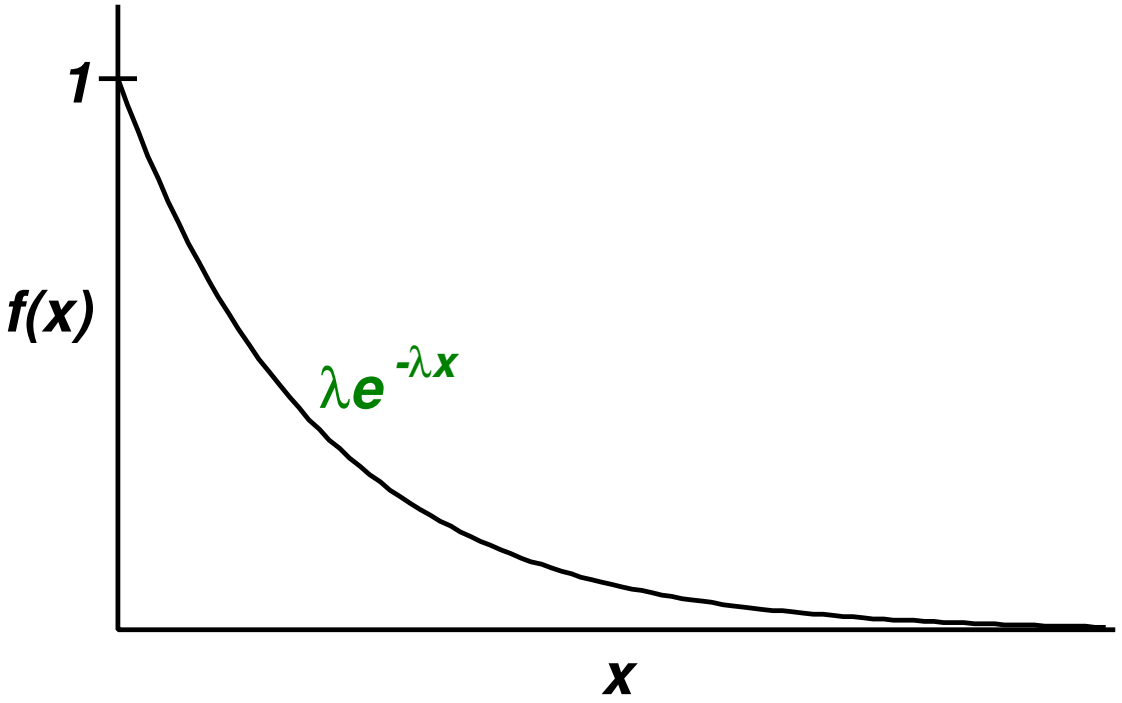
Random Arrivals And Random Task Sizes

- Most systems are somewhere in between best and worst cases
  - how do you represent an "average" system?
  - you would like to assume that *arrivals* are *independent* of each other and *task sizes* are *random*
- A useful model for understanding queueing behavior is to use an *exponential distribution* to describe the time between tasks arriving *(inter-arrival time)* and the time it takes to service each task
  - the exponential distribution provides a stunningly simple *approximate* description of real-life queueing system
    - we are not claiming that all real systems always obey the exponential model; in fact, most do not
    - however, the model is often accurate enough to provide insight on system behavior; it's also easy to understsand the circumstances under which it is inaccurate
- *"All models are wrong, but some models are useful"* is a well-known scientific precept

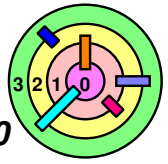


Exponential Distribution

- In math, an exponential distribution of a continuous random variable with a mean of $1/\lambda$ has the probability density function: $f(x) = \lambda e^{-\lambda x}$
  - a useful property of an exponential distribution is that it is *memoryless*
    - no matter how long you have waited, your expected wait time stays constant

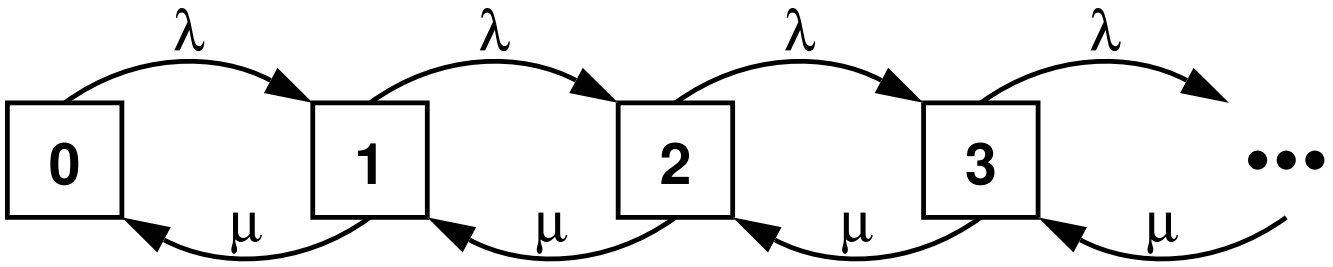


- A Gaussian/normal distribution is more like our best case scenario
  - most cases looks like the average case
- A *heavy-tailed distribution* is closer to our worst case scenario
  - the longer you have waited for an event, the longer you are likely to still need to wait

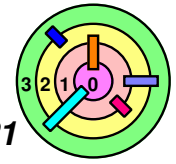


Exponential Distribution

- With a ***memoryless*** distribution, the behavior of queueing systems become simple to understand
  - our queueing system becomes a ***finite state machine***

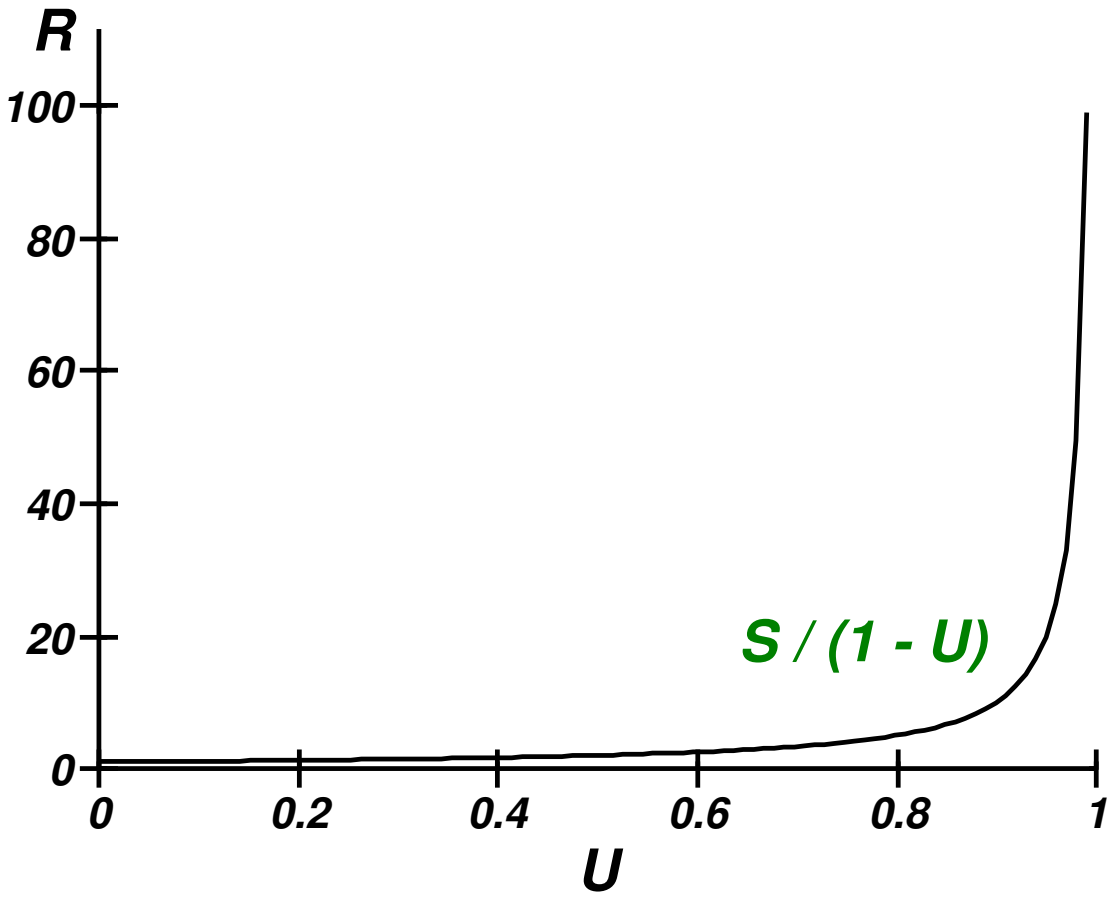


  - the name of a state is the number of tasks at the system
  - start in state 0, leaving state 0 at the rate of λ
  - in state 1, leaving to go to state 2 at the rate of λ and simultaneously leaving to go to state 0 at the rate of μ
  - and so on
  - using this model, we can calculate the ***average response time*** (assuming $\lambda < \mu$) is: $R = S / (1 - U) = 1 / (\mu - \lambda)$
    - $R \rightarrow$ infinity as λ approaches μ *(or, as* $U \rightarrow 1$*)*



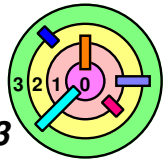
Average Response Time vs. Utilization

- When the utilization is low, very little queueing delay and response time is close to service time
- As utilization increases, queueing and response time also increases and the relationship is highly non-linear
  - at high utilization, small increase in utilization can drastically increase queueing delay and response time



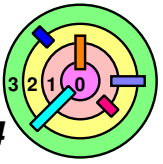
Examples

- Suppose a queueing system (with exponentially distributed arrivals and task sizes) is 20% utilized
  - if the utilization is increased by 5% (i.e., to 25% utilization), how much does the response time increase?
    - *R = S / (1 - U) = S / (1 - 0.2) = 1.25 S*
    - *R' = S / (1 - U') = S / (1 - 0.25) = 1.33 S*
    - *ΔR = (R' - R) = (1.33 - 1.25) S = 0.08 S*
      - small increase in response time
- Suppose a queueing system (with exponentially distributed arrivals and task sizes) is 90% utilized
  - if the utilization is increased by 5% (i.e., to 95% utilization), how much does the response time increase?
    - *R = S / (1 - U) = S / (1 - 0.9) = 10 S*
    - *R' = S / (1 - U') = S / (1 - 0.95) = 20 S*
    - *ΔR = (R' - R) = (20 - 10) S = 10 S*
      - response time doubles



Variance

- For a queueing system with exponentially distributed arrivals and task sizes, variance in response time $Var(R) = S / [(1 - U)^2]$
- What if arrivals are less bursty than exponential?
  - variance would decrease (goes to zero in the extreme case)
- What if arrivals are more bursty than exponential?
  - variance would increase (goes to infinity in the extreme case)

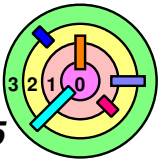


Secondary Bottlenecks

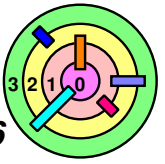
- If a thread visits multiple resources (e.g., processor, disk, network), its total response time is just the sum of the individual response times

$$R = \sum_i S_i / (1 - U_i)$$

  - the bottleneck resource is the resource with the largest response time / delay
  - if you improve the available resource at the bottleneck (e.g., add another processor or buy a faster processor), the bottleneck can move to another resource (i.e., the secondary bottleneck)

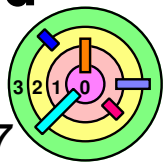


(7.6) Overload Management



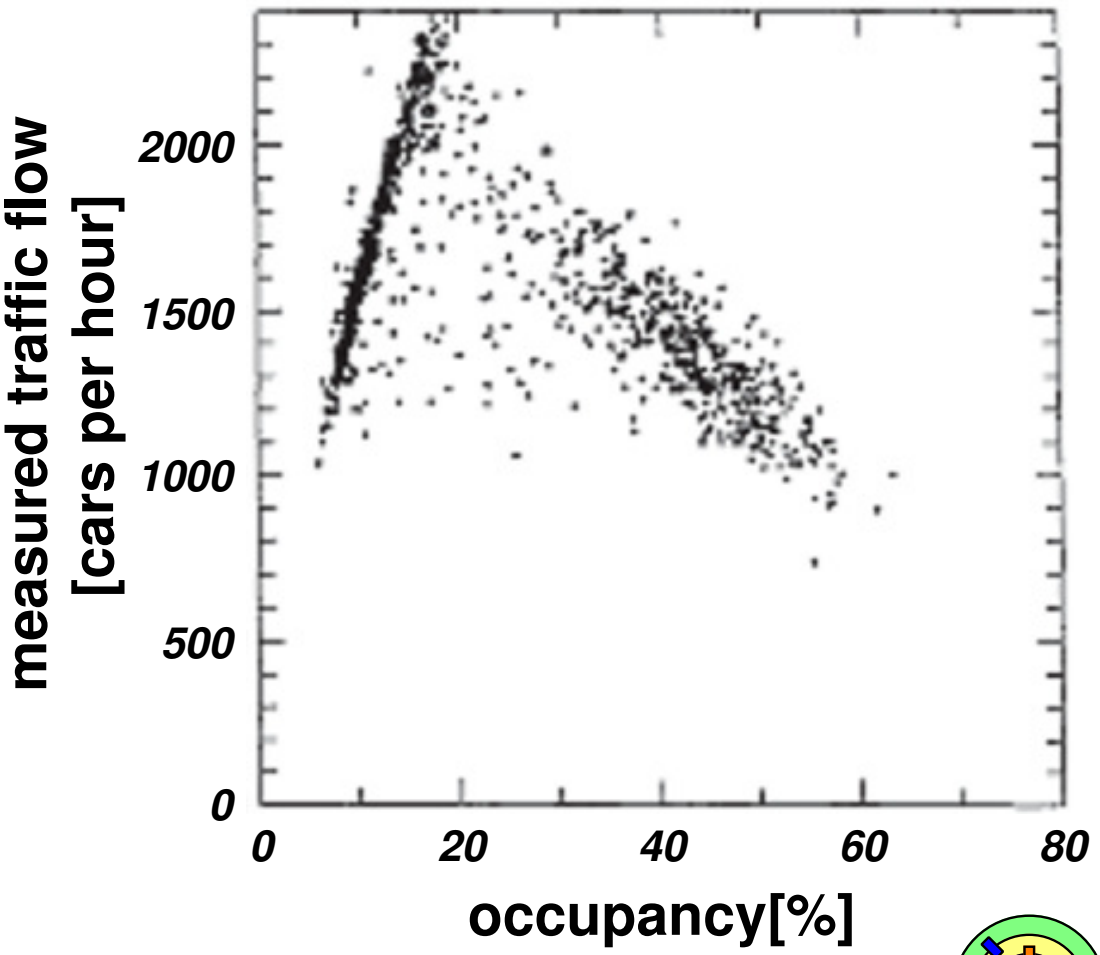
Overload Management

- What if arrivals occur faster than service can handle them
  - if do nothing, response time will become infinite
  - there are basically two ways to deal with this
    - either turn arrivals away or slow everyone down (i.e., degrade service)
- Turn users away?
  - but which ones?
    - ART would be smaller if you turn away users that have the highest service demand (although they can get very upset)
- Degrade service?
  - compute result with fewer resources
  - Ex: CNN static front page on 9/11, Amazon quick response
- Restaurant analogy
  - solution: once you are in the restaurant, you should get good service; if you cannot get in, you wait (or leave)



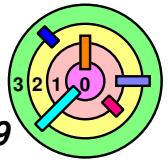
Increased Load ⇒ Increased Overhead

- In a real system, when load increases, overhead also increases
  - e.g., if you use a list to manage requests, as the list gets longer, it takes more time to access the list
  - e.g., highway traffic
    - at low load, increase load = more throughput
    - when congestion starts, throughput would decrease
      - keep increasing load can lead to traffic jam and throughput can becomes really low
    - solution: *admission control* to *limit system load*

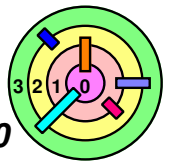


Increased Load ⇒ Increased Overhead

- In a real system, when load increases, overhead also increases
  - e.g., time slicing and caching
    - when web server load is low, hit rate on web caches are high and tasks uses cache efficiently
    - as more tasks are added to the system, there are more time slices and fewer web cache hits
  - e.g., Internet traffic
    - when load is low, all packets go through quickly
    - as load increases, some packets are dropped which causes senders to retransmit the dropped packets and effectively increases the load even more
      - positive feedback

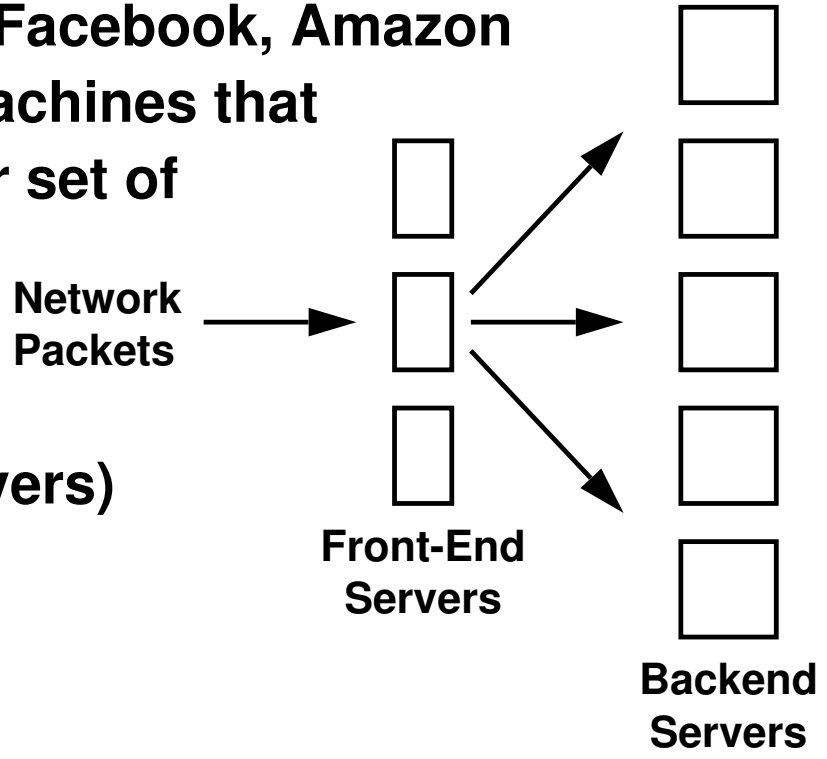


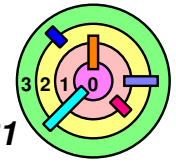
(7.7) Case Study: Servers in a Data Center



Architecture For Providing Web Services

- Many web services, such as Google, Facebook, Amazon are organized as a set of front-end machines that redirect incoming requests to a larger set of backend machines
  - isolates from the architecture of the backend systems (can easily add or remove backend servers)

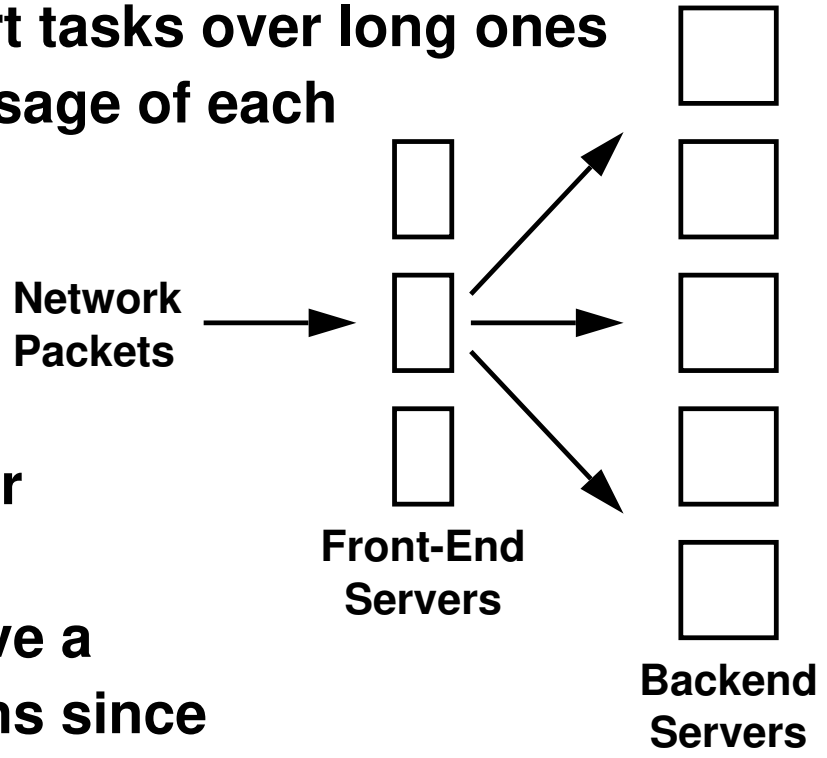


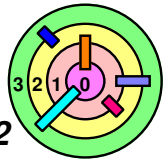


Architecture For Providing Web Services

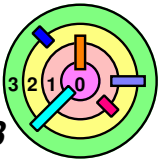
To provide good response time to clients:

- when a client first connects to the service, the front-end node assigns each customer to a backend server to ***balance load***
- additional request from the same client would be assigned to the same backend server (affinity scheduling)
- a backend server would favor short tasks over long ones by keeping track of the resource usage of each client
- must monitor arrival rates of clients and resource usage of each client so that backend servers can be added before server utilization gets too high
- need to predict future load and have a backup plan for overload conditions since it can take a long time to bring new servers online





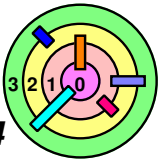
Extra Slides



Ch 8: Address Translation

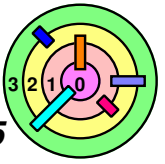
Bill Cheng

http://merlot.usc.edu/william/usc/

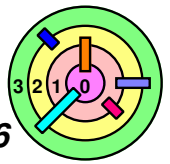


Main Points

- Address translation concept
  - how do we convert a virtual address to a physical address?
- Flexible address translation
  - base and bound
  - segmentation
  - paging
  - multilevel translation
- Efficient address translation
  - translation lookside buffers
  - virtually and physically addressed caches

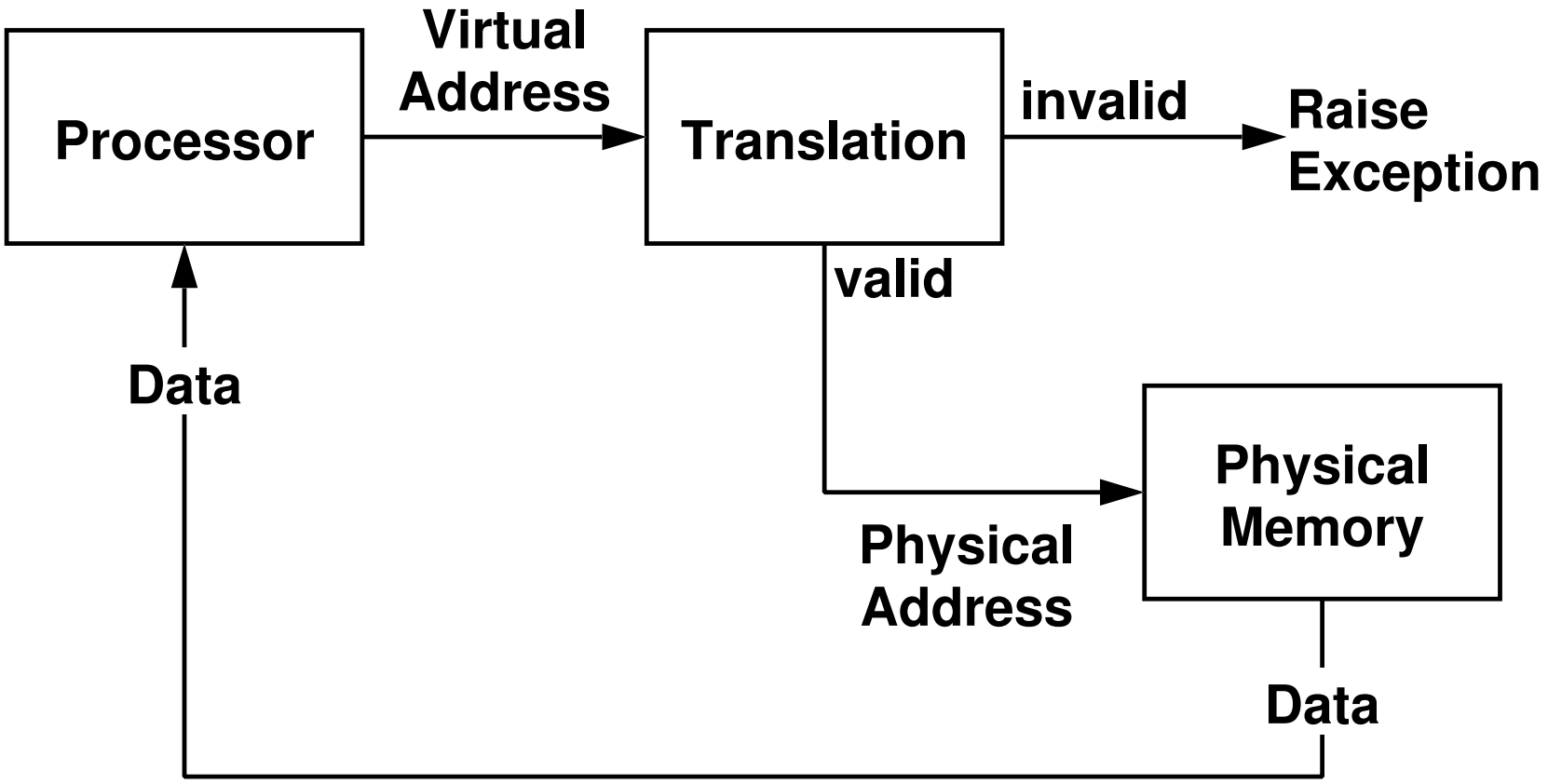


(8.1) Address Translation Concept



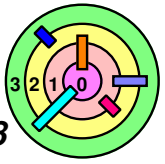
Address Translation Concept

- Don't use physical addresses directly
  - use virtual address instead and translate it to a physical address in hardware before accessing physical memory (RAM)

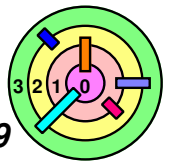


Address Translation Goals

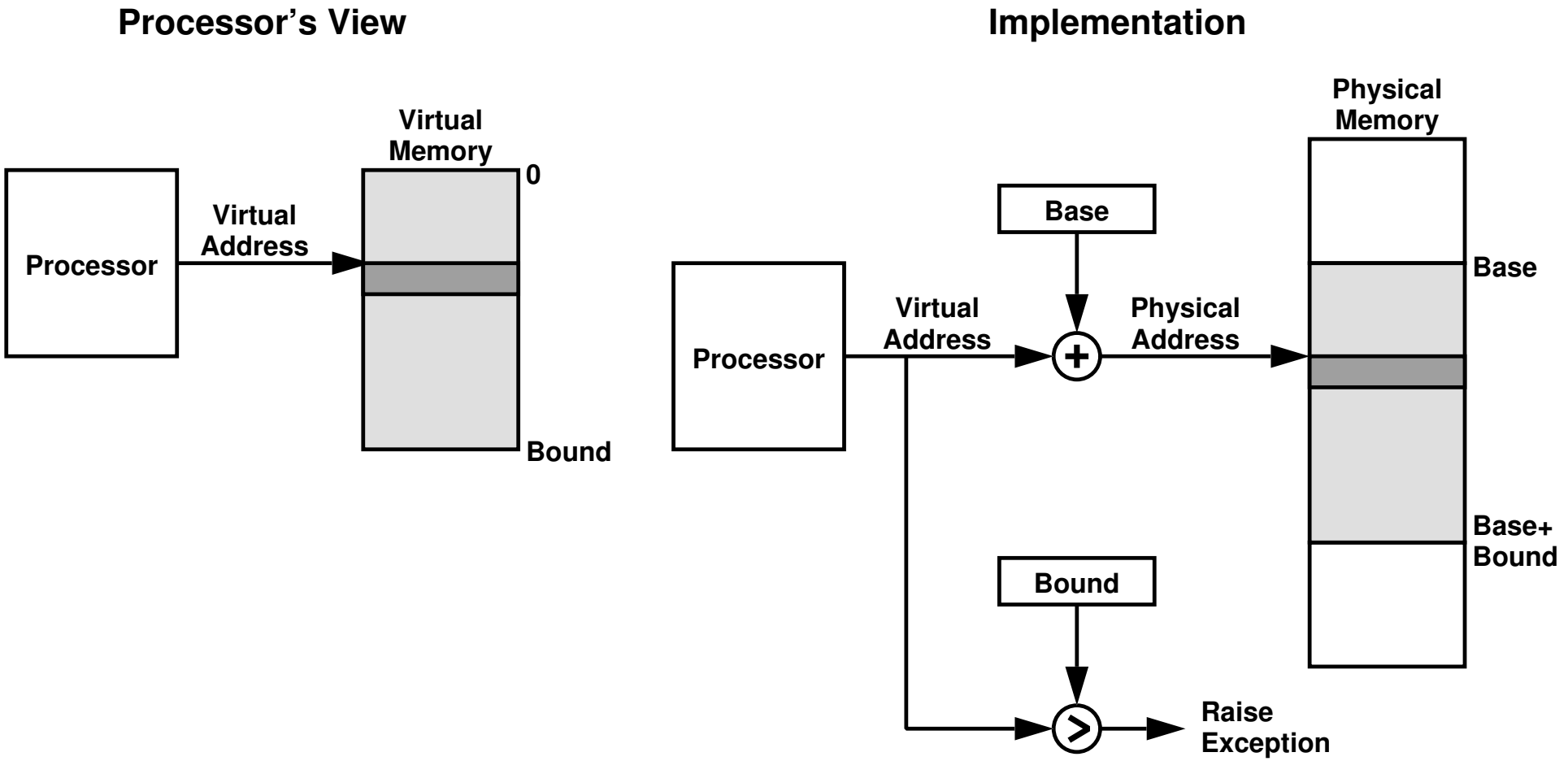
- ***Memory protection:*** protect processes from accessing each other's memory and protect kernel from all processes
- ***Memory sharing:*** share selected regions of memory
  - shared libraries, shared files, shared data structures
- ***Flexible memory placement:*** in physical memory / RAM
- ***Sparse addresses:*** do not waste RAM
  - allows multiple dynamic regions of memory to grow and shrink
- ***Runtime lookup efficiency:*** fast and compact hardware
- ***Compact translation table:*** minimal space overhead
- ***Portability:*** make it easy to map kernel data structures to specific capabilities of various address translation architectures



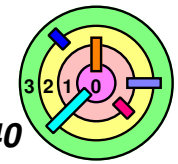
(8.2) Towards Flexible Address Translation



Base & Bounds Registers

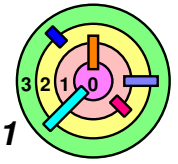


- Usable virtual addresses are from 0 to Bound-1
- Base and bound registers are part of the context of the process



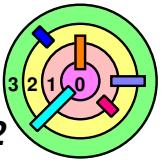
Base & Bounds Registers

- Pros
  - simple
  - fast: 2 registers, adder, comparator
  - can relocate in physical memory with small change in PCB
- Cons
  - cannot keep program from accidentally overwriting its own code
  - cannot share code/data with other processes
  - cannot grow stack/heap as needed

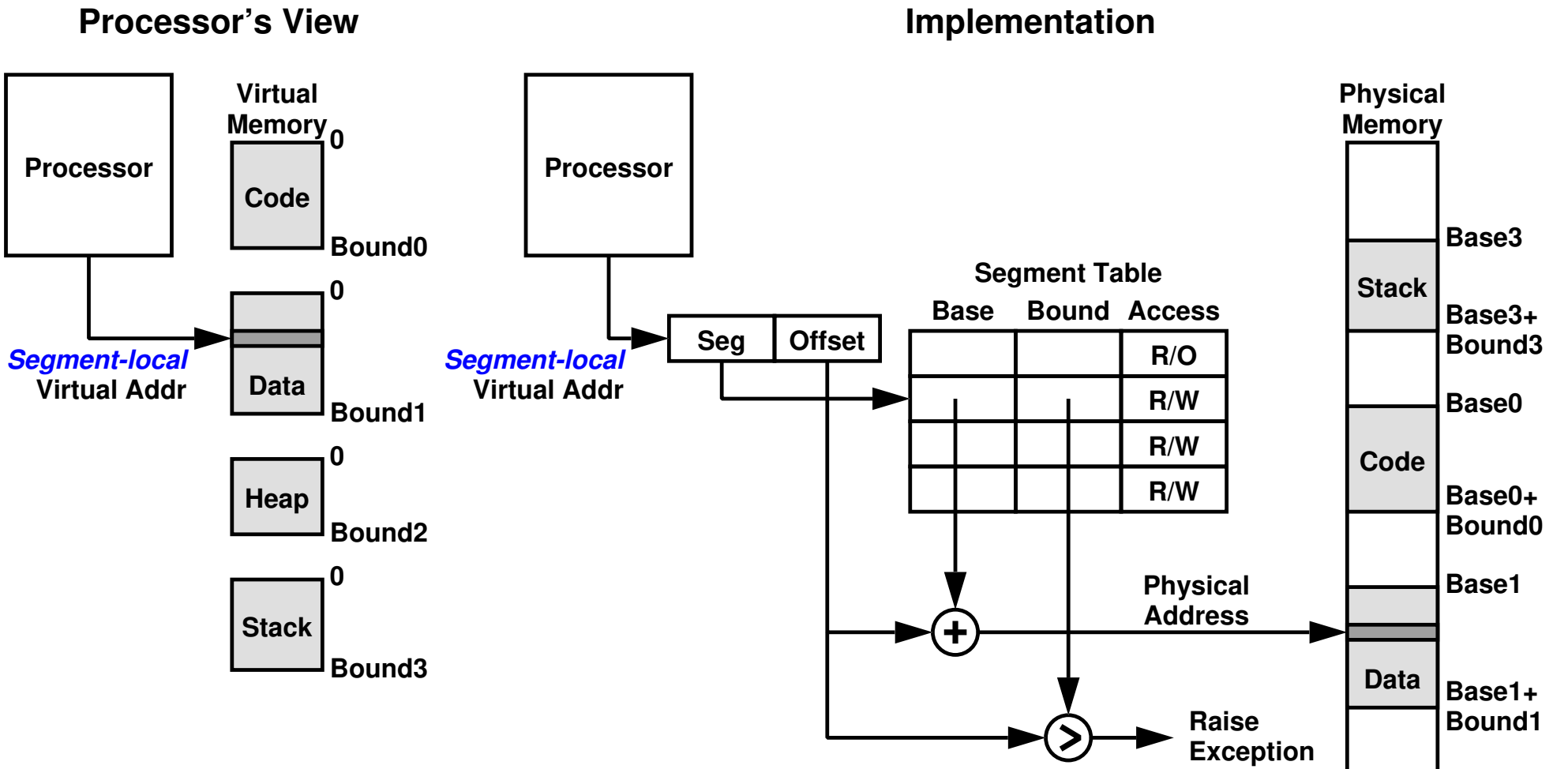


Segmentation

- A *segment* is a contiguous region of *virtual* memory
- Each process has a segment table (in hardware)
  - one each per segment
- Segment can be located anywhere in physical memory
  - each segment has: start, length, access permission
- Processes can share segments
  - if they have the same start and length, although they can have different access permissions



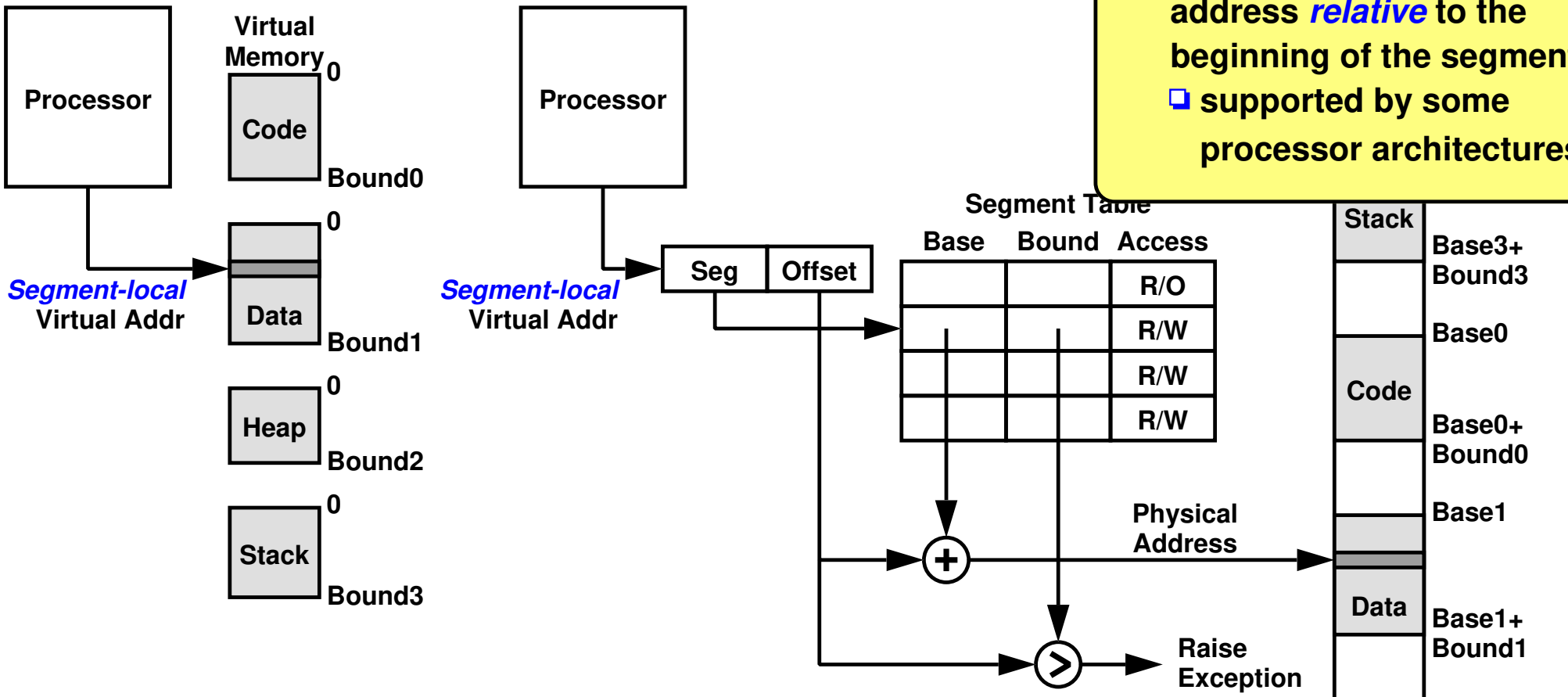
Segmentation



⇨ ***Segmentation fault:*** **if an exception is raised during address translation**

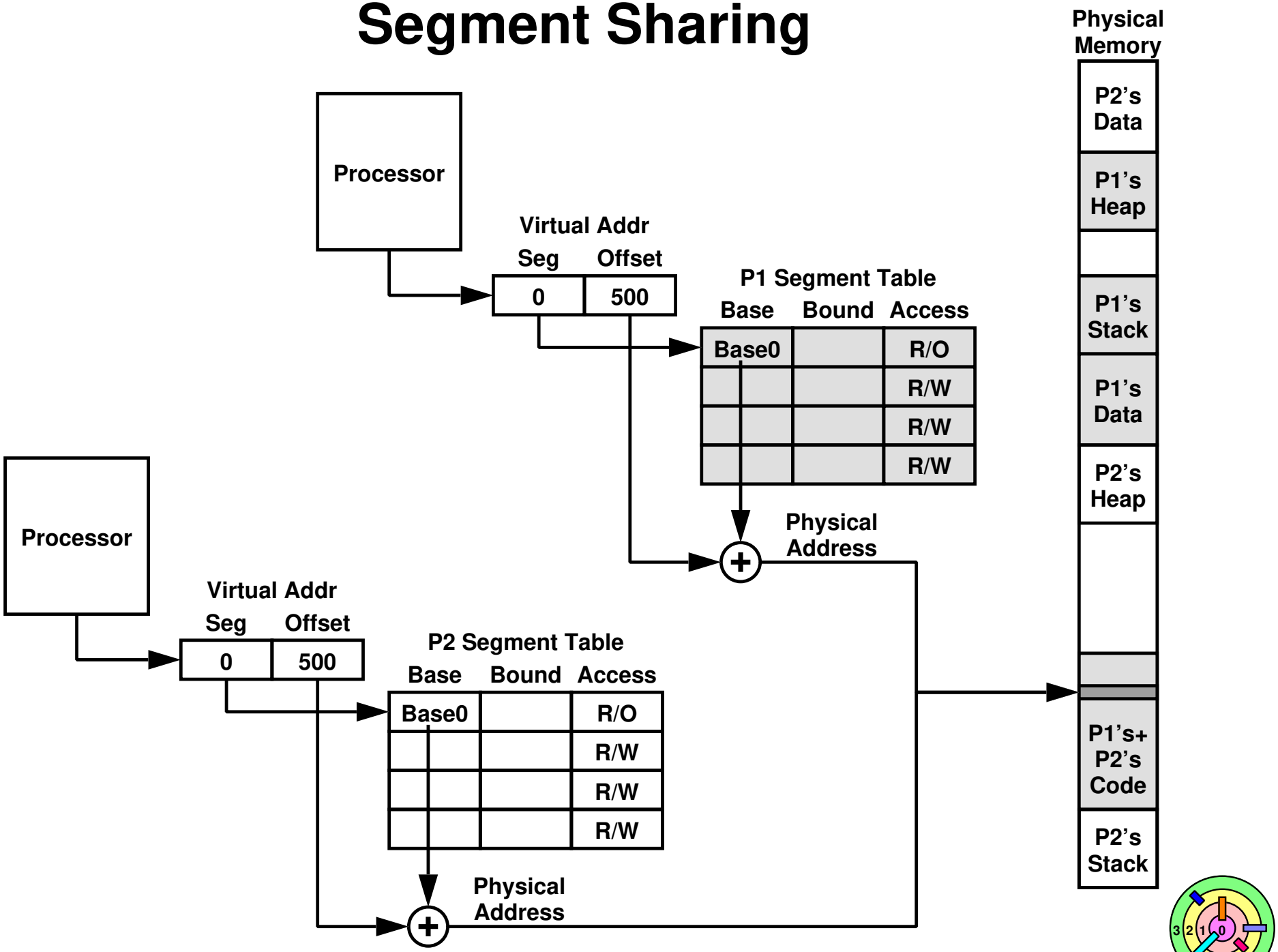
Segmentation

- *segment-local* virtual address is a virtual address *relative* to the beginning of the segment
  - supported by some processor architectures



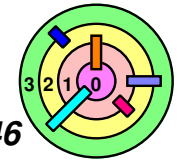
⇨ ***Segmentation fault:*** **if an exception is raised during address translation**

Segment Sharing



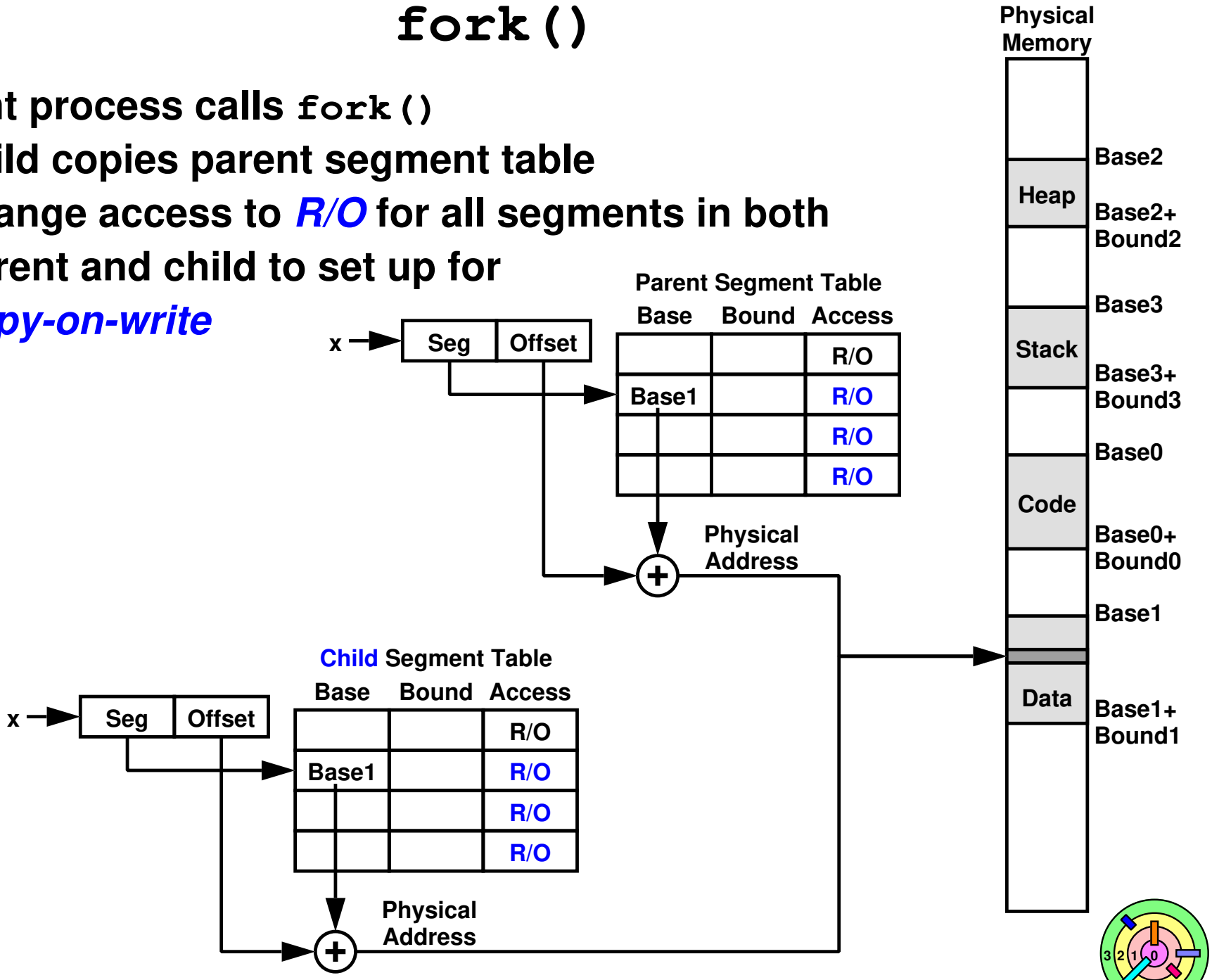
UNIX `fork()` And Copy-On-Write

- UNIX `fork()`
  - makes a complete copy of a process
- Segments allow more efficient implementation using *copy-on-write*
  - in `fork()`, copy *segment table* into child
  - mark parent and child segments read-only
  - start child process; return to the same place as parent, i.e., returning from the `fork()` system call
  - if child or parent writes to a segment (e.g., data, stack, heap)
    - trap into the kernel
    - make a copy of the segment, change base register to point to new segment, make new segment R/W, and resume
    - this type of *segmentation fault* can be *repaired*
- *Copy-on-write:* you can share all you want if you are just reading, but when you write to a segment *for the first time*, you make a copy of the segment and you write into the copy
  - all subsequent writes will not cause trap



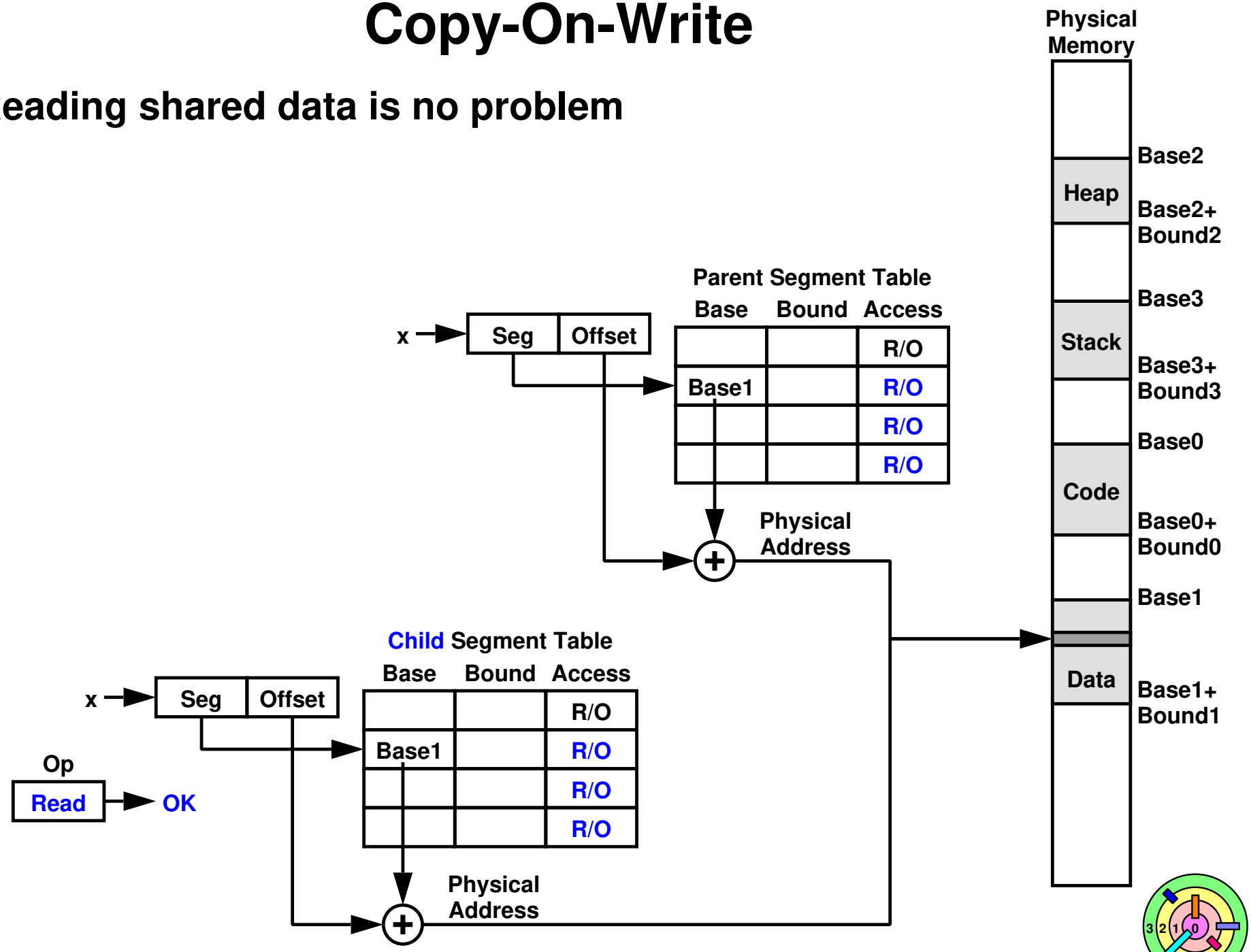
fork()

- Parent process calls `fork()`
  - child copies parent segment table
  - change access to *R/O* for all segments in both parent and child to set up for *copy-on-write*



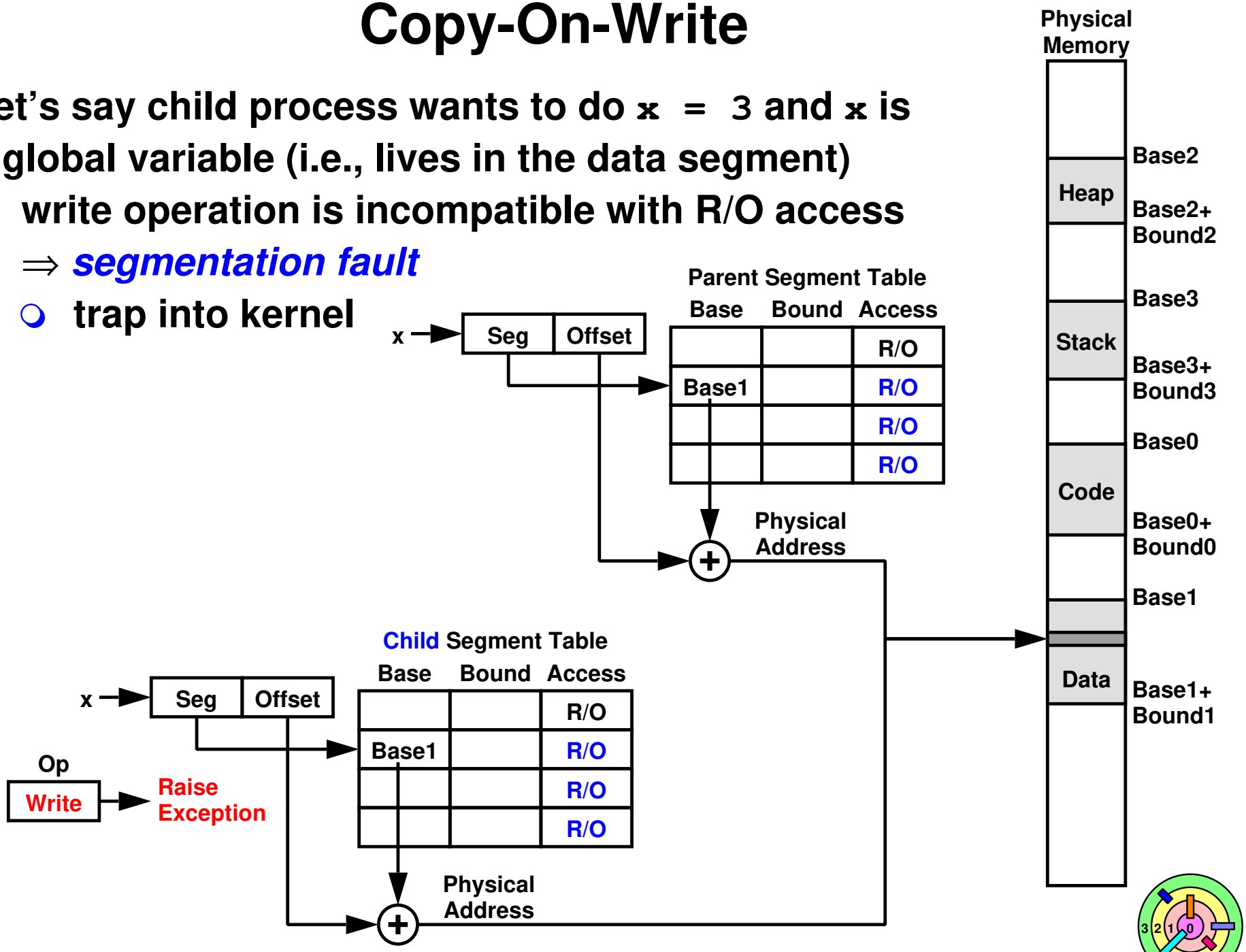
Copy-On-Write

- Reading shared data is no problem



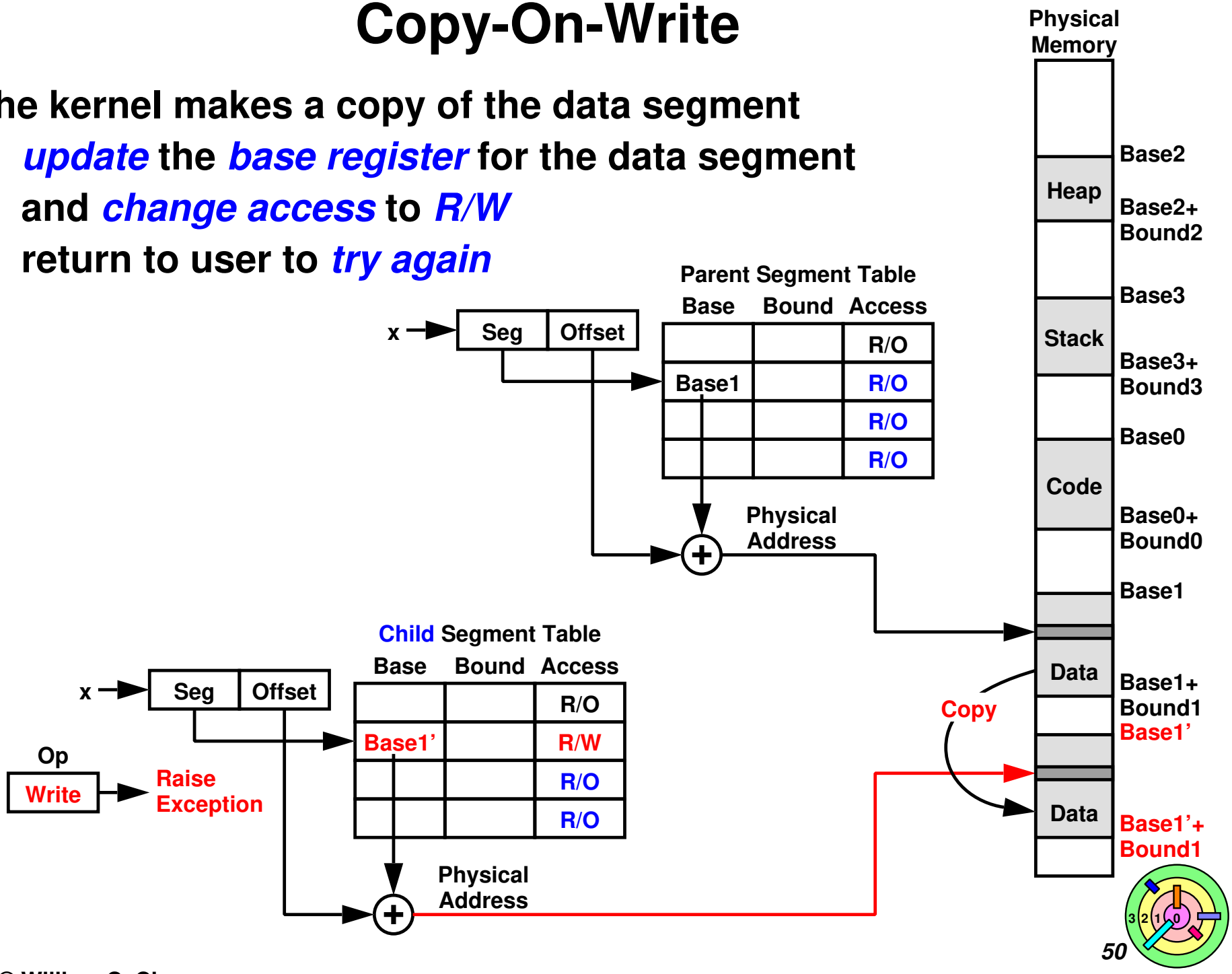
Copy-On-Write

- Let's say child process wants to do `x = 3` and `x` is a global variable (i.e., lives in the data segment)
  - write operation is incompatible with R/O access
    - ⇒ *segmentation fault*
    - trap into kernel



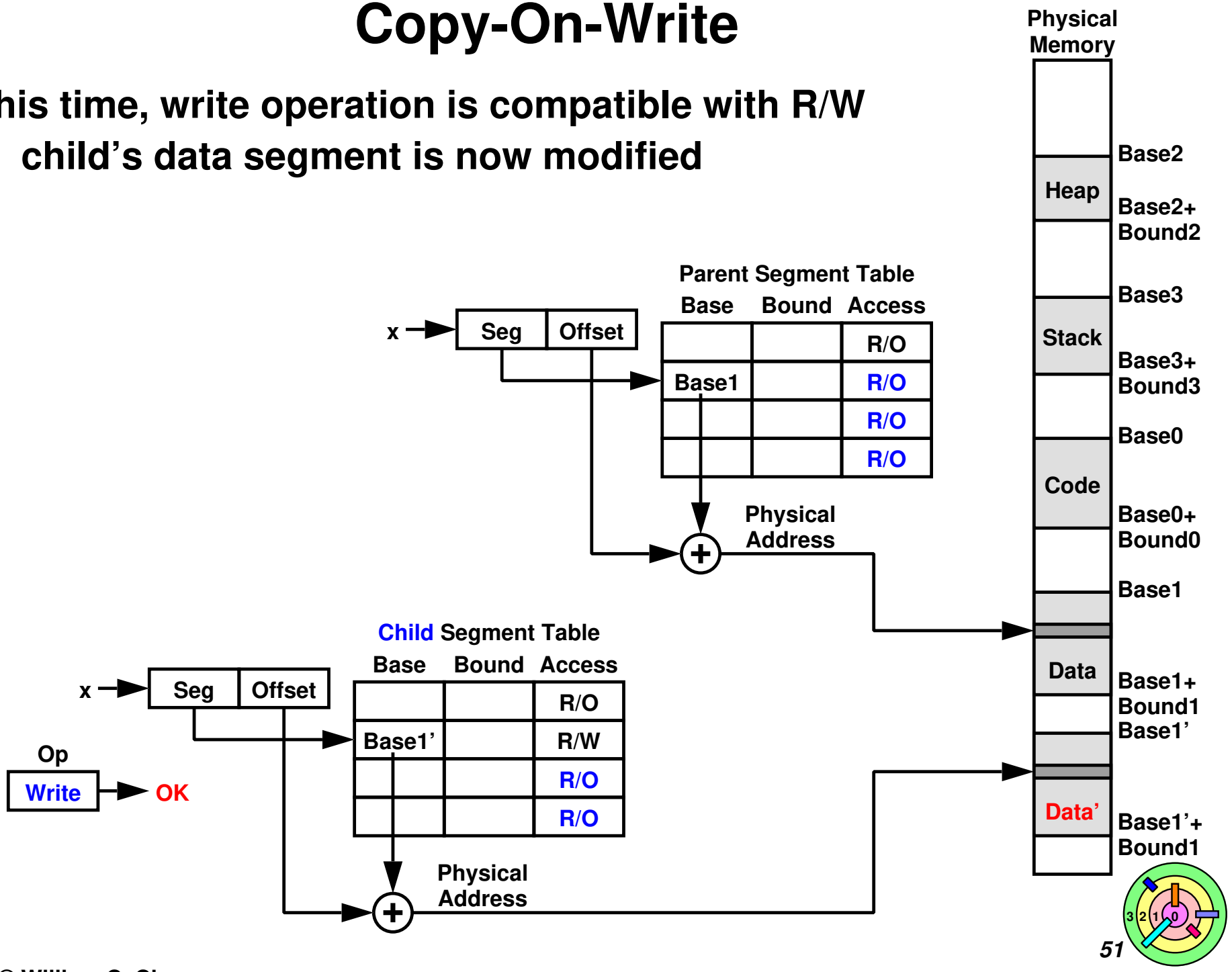
Copy-On-Write

- The kernel makes a copy of the data segment
  - *update* the *base register* for the data segment and *change access* to *R/W*
  - return to user to *try again*



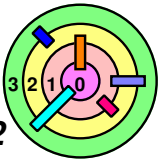
Copy-On-Write

- This time, write operation is compatible with R/W
  - child's data segment is now modified



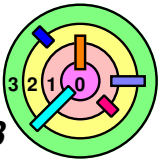
Zero-On-Reference

- How much physical memory is needed for the stack or heap?
  - only what is currently in use
- When program uses memory beyond end of stack
  - segmentation fault into OS kernel
  - kernel allocates some memory
    - how much?
  - zeroes the memory
    - security: avoid accidentally leaking information
  - modify segment table
  - resume process



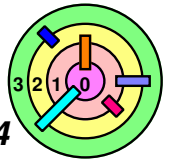
Segmentation

- Pros
  - can share code/data segments between processes
  - can protect code segment from being overwritten
  - can transparently grow stack/heap as needed
  - can detect if need to copy-on-write
- Cons
  - complex memory management
    - need to find chunk of a particular size
  - may need to rearrange memory from time to time to make room for new segment or growing segment
    - *external fragmentation:* wasted space between allocated chunks

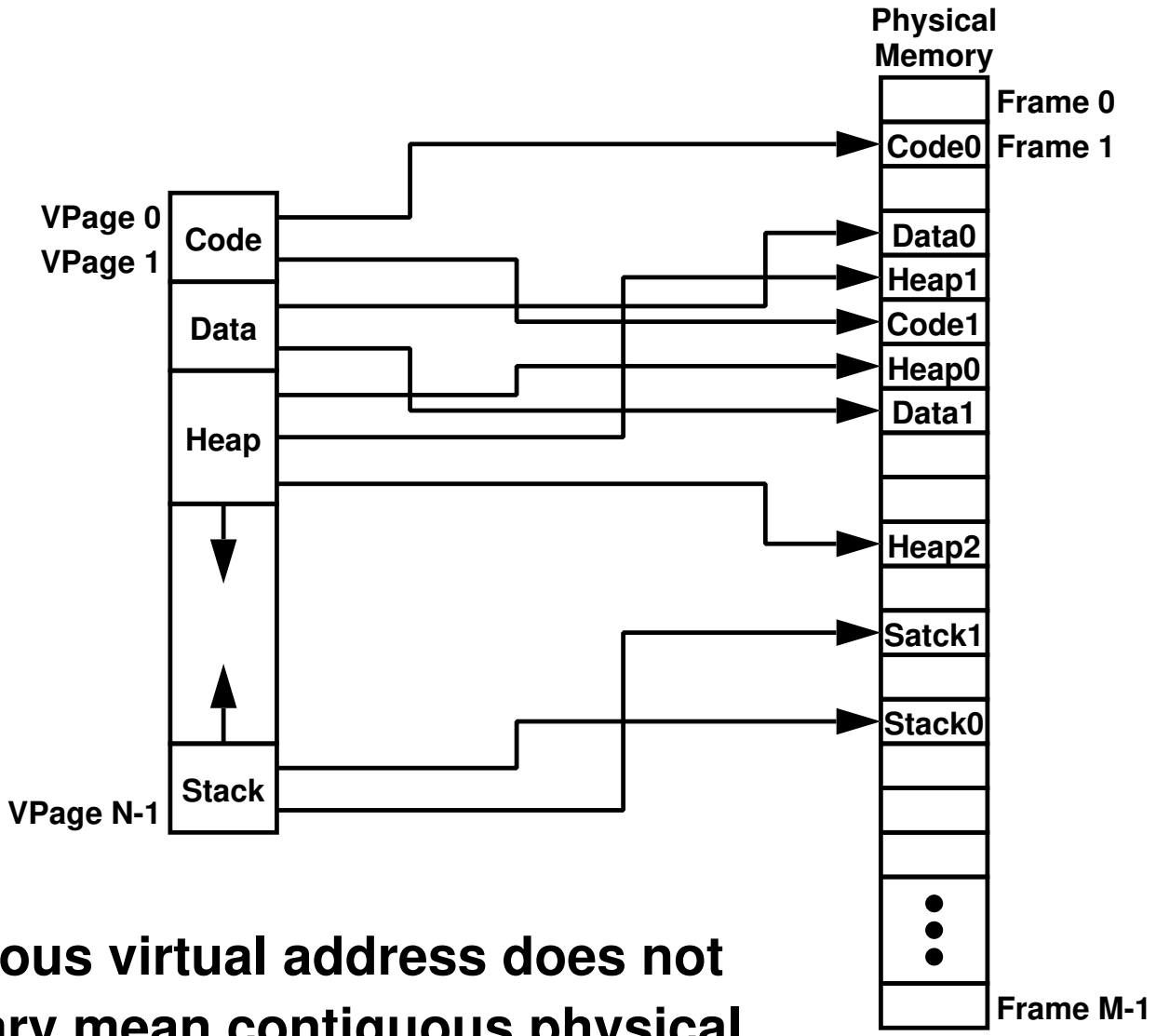


Paged Memory / Paging

- Manage memory in fixed size units, called *page frames*
- Finding a free page is easy (compared to segmentation)
  - allocation bitmap: `0011111100000001100`
    - 1 means allocated and 0 means unallocated
  - each bit represents one physical page frame
- Each process has its own *page table* for performing address translation
  - a page table is a *kernel data structure*
    - conceptually, a page table as an *array of page table entries*
      - array index is a *virtual page number*
      - virtual page size = physical page size
  - stored in physical memory (accessible via virtual addr as well)
  - hardware registers
    - pointer to start of page table (i.e., contains the first physical memory address of the page table)
    - page table length

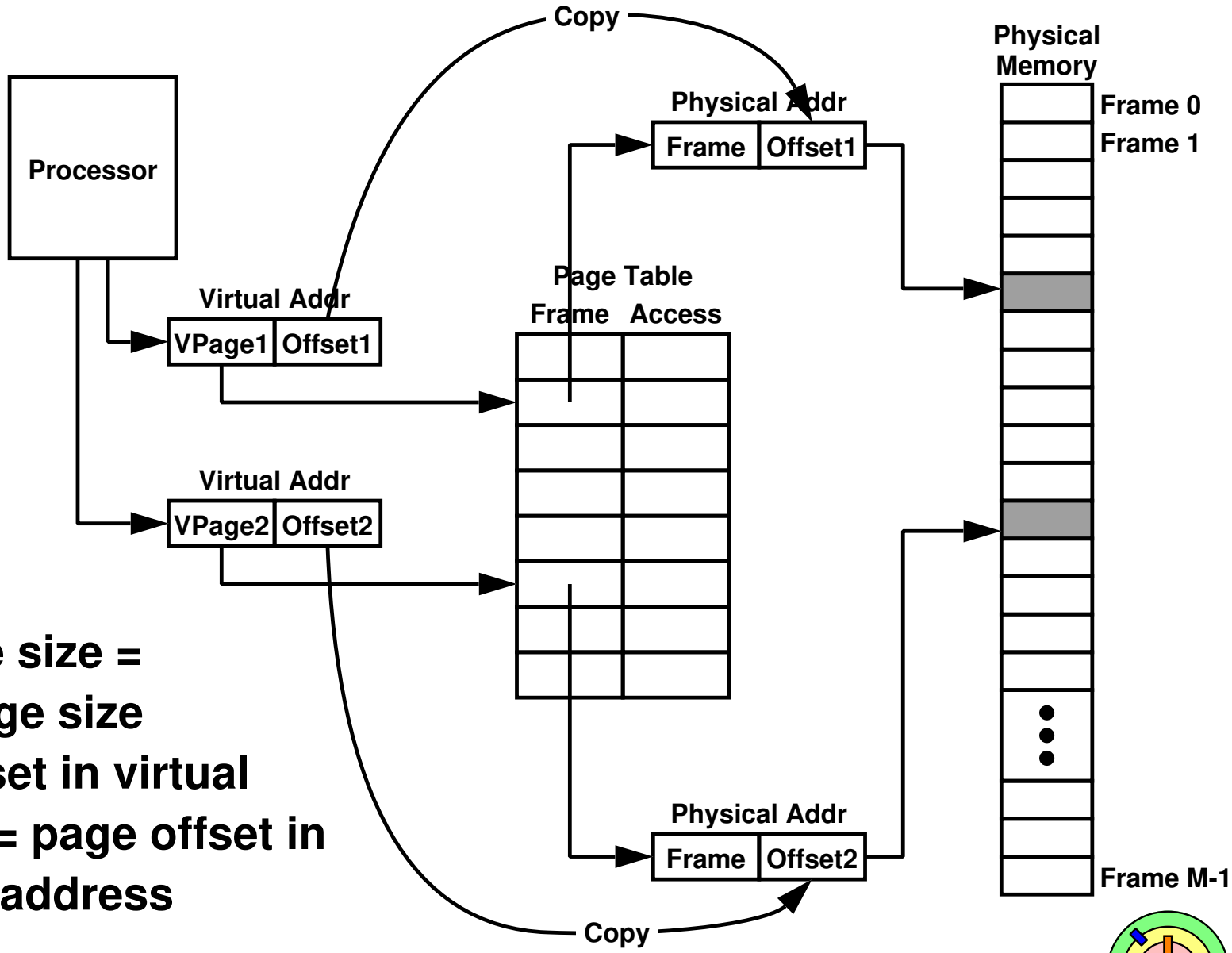


Page Translation (Logical View)



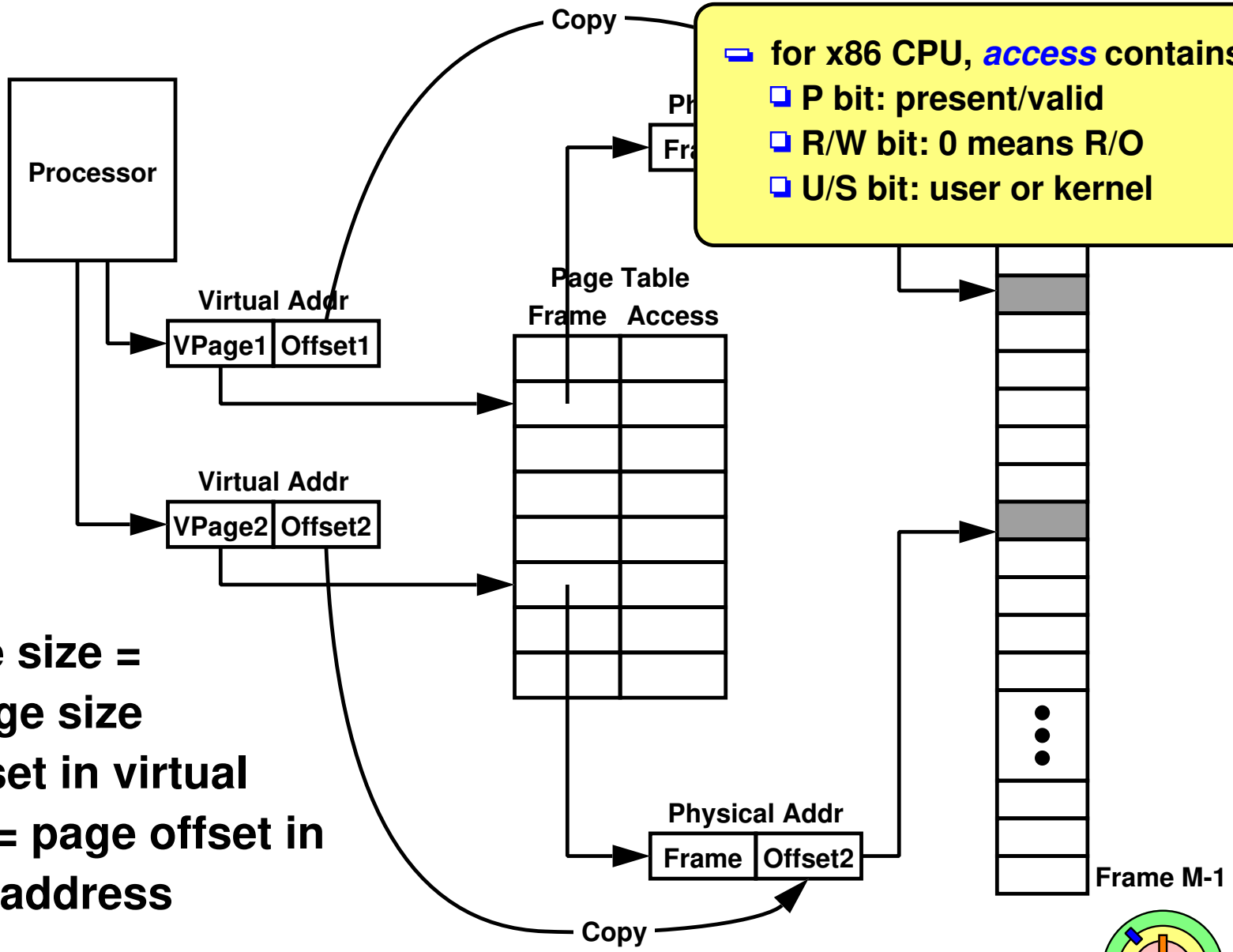
Contiguous virtual address does not necessary mean contiguous physical address (when crossing page boundary)

Page Translation (Implementation)



- Virtual page size = physical page size
  - page offset in virtual address = page offset in physical address
- Frame number = physical page number

Page Translation (Implementation)



- for x86 CPU, *access* contains:
  - P bit: present/valid
  - R/W bit: 0 means R/O
  - U/S bit: user or kernel

- Virtual page size = physical page size
  - page offset in virtual address = page offset in physical address
- Frame number = physical page number

Example

- Suppose page size is 4 bytes
  - where does virtual address 6 map to?
  - where does virtual address 9 map to?

Process View

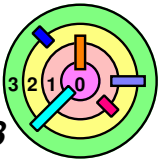
0	A	VPage 0
1	B	
2	C	
3	D	
4	E	VPage 1
5	F	
6	G	
7	H	
8	I	VPage 2
9	J	
10	K	
11	L	

Page Table

Frame	Access
4	
3	
1	

Physical Memory

0	Frame 0
1	
2	
3	
4	Frame 1
5	
6	
7	
8	Frame 2
9	
10	
11	
12	Frame 3
13	
14	
15	
16	Frame 4
17	
18	
19	



Example

- Suppose page size is 4 bytes
  - where does virtual address 6 map to?
  - where does virtual address 9 map to?

Process View

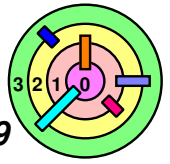
0	A	VPage 0
1	B	
2	C	
3	D	
4	E	VPage 1
5	F	
6	G	
7	H	
8	I	VPage 2
9	J	
10	K	
11	L	

Page Table

Frame	Access
4	
3	
1	

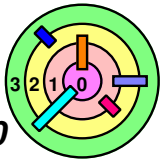
Physical Memory

0		Frame 0
1		
2		
3		
4	I	Frame 1
5	J	
6	K	
7	L	
8		Frame 2
9		
10		
11		
12	E	Frame 3
13	F	
14	G	
15	H	
16	A	Frame 4
17	B	
18	C	
19	D	



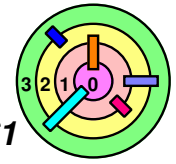
Paging

- With paging, what is saved/restored on a process context switch?
  - pointer to page table, size of page table
  - page table itself is in main memory
- What if page size is very small?
  - page table will be large
- What if page size is very big?
  - *internal fragmentation:* if we cannot use all of the space inside a fixed size chunk



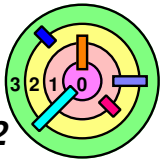
Paging And Copy-On-Write

- Can we share memory between processes?
  - set entries in both page tables to point to same page frames
  - need a *core map* to track which processes are pointing to which page frames
    - a core map is a data structure used to perform reverse lookup, i.e., given a page frame, which processes are sharing that page frame
      - a reference counting mechanism is use to know when to free up page frame
- UNIX `fork()` with copy on write
  - on `fork()`, copy page table of parent into child process
  - mark all pages (in new and old page tables) as *read-only*
  - trap into kernel on write (in child or parent)
  - copy page
  - mark both as writable
  - resume execution



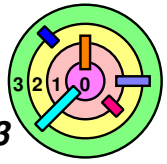
Fill On Demand

- Can I start running a program before its code is in physical memory?
  - set all page table entries to *invalid*
  - when a page is referenced for first time, if the page table entry is invalid, will cause a *page fault* and trap into kernel
  - kernel brings page in from disk
  - resume execution
    - returning from page fault will be retried
  - remaining pages can be transferred in the background while program is running
- This is also referred as *demand paging* or *on-demand paging*



Sparse Address Spaces

- Might want to have many separate dynamically created segments
  - per-thread stacks
  - *memory-mapped files:* when you map a file (or part of a file) into your address space, you must create a new segment
- What if virtual address space is large?
  - 32-bits, 4KB pages ⇒ 500K page table entries (assuming that the user portion of the address space is 2GB)
  - 64-bits ⇒ 4 quadrillion page table entries



Multi-level Translation

- How many page table entries are valid?
  - for small programs, not many ⇒ array is not a good choice
  - trees and hash tables are better for sparse data
  - we will first look at tree data structures
- Tree of translation tables
  - paged segmentation
  - multi-level page tables
  - multi-level paged segmentation
- Fixed-size page as lowest level unit of allocation
  - efficient for sparse addresses (compared to array-based paging)
  - efficient memory allocation (compared to segments)
  - efficient disk transfers (fixed size units)
  - efficient lookup with *translation lookaside buffers* (next section)
  - efficient reverse lookup *(from physical to virtual, using core map)*
  - page-granularity for protection and sharing