# Memory Management: Virtual Memory and Paging
# CS 111
# Operating Systems
Peter Reiher

# Outline

- Paging
- Swapping and demand paging
- Virtual memory

# Paging

- What is paging?
  - What problem does it solve?
  - How does it do so?
- Paged address translation
- Paging and fragmentation
- Paging memory management units
- Paging and segmentation

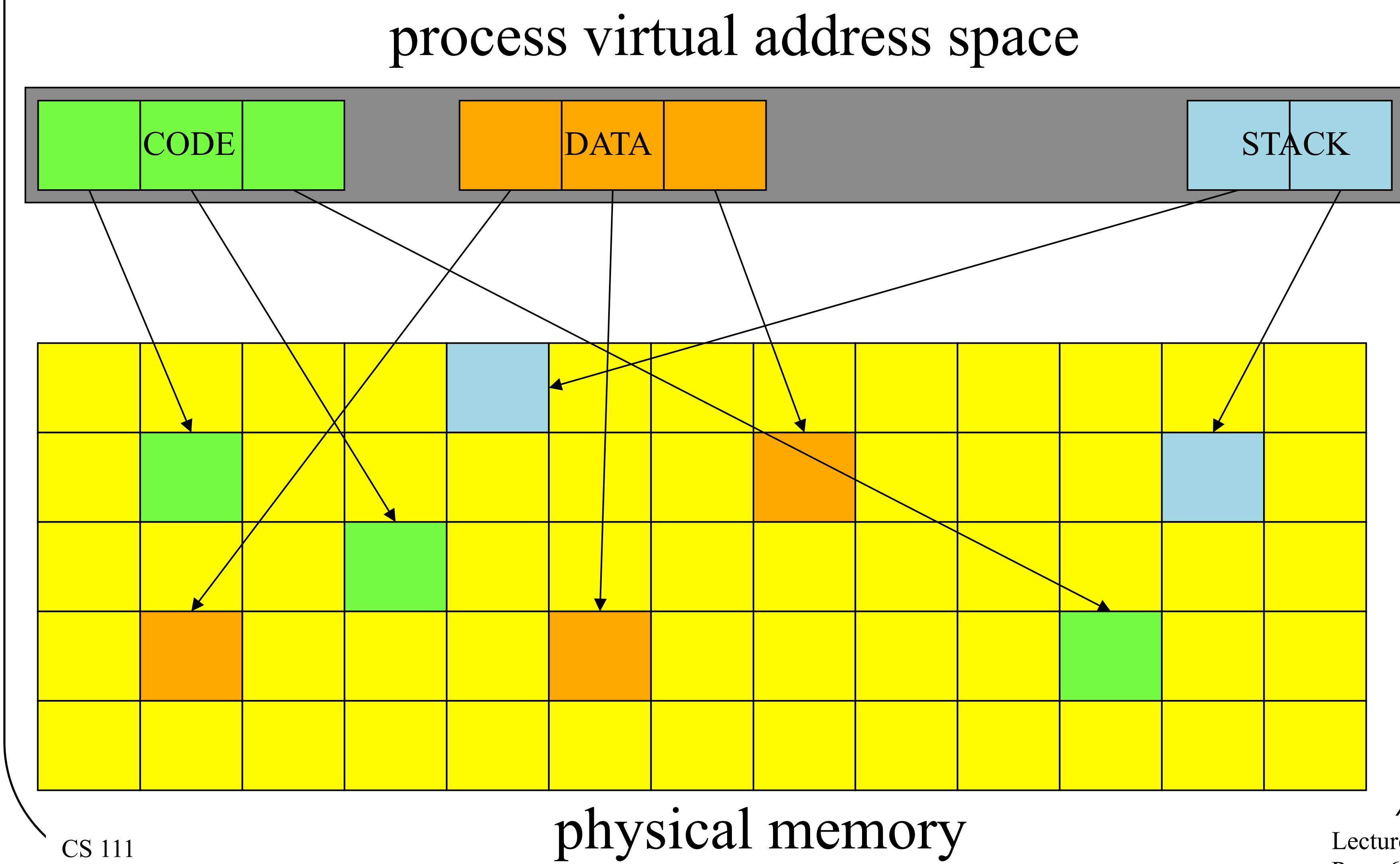
# Segmentation Revisited

- Segment relocation solved the relocation problem for us
- It used base registers to compute a physical address from a virtual address
  - Allowing us to move data around in physical memory
  - By only updating the base register
- It did nothing about external fragmentation
  - Because segments are still required to be <u>contiguous</u>
- We need to eliminate the "contiguity requirement"

# The Paging Approach

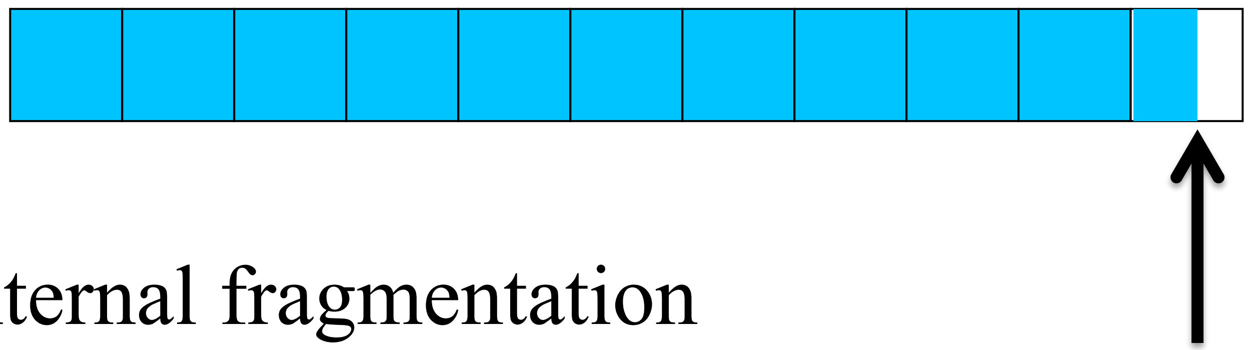
- Divide physical memory into units of a single fixed size
  - A pretty small one, like 1-4K bytes or words
  - Typically called a *page frame*
- Treat the virtual address space in the same way
- For each virtual address space page, store its data in one physical address page frame
- Use some magic per-page translation mechanism to convert virtual to physical pages

# Paged Address Translation



# Paging and Fragmentation

- A segment is implemented as a set of virtual pages



- Internal fragmentation
  - Averages only ½ page (half of the last one)
- External fragmentation
  - Completely non-existent
  - We never carve up pages

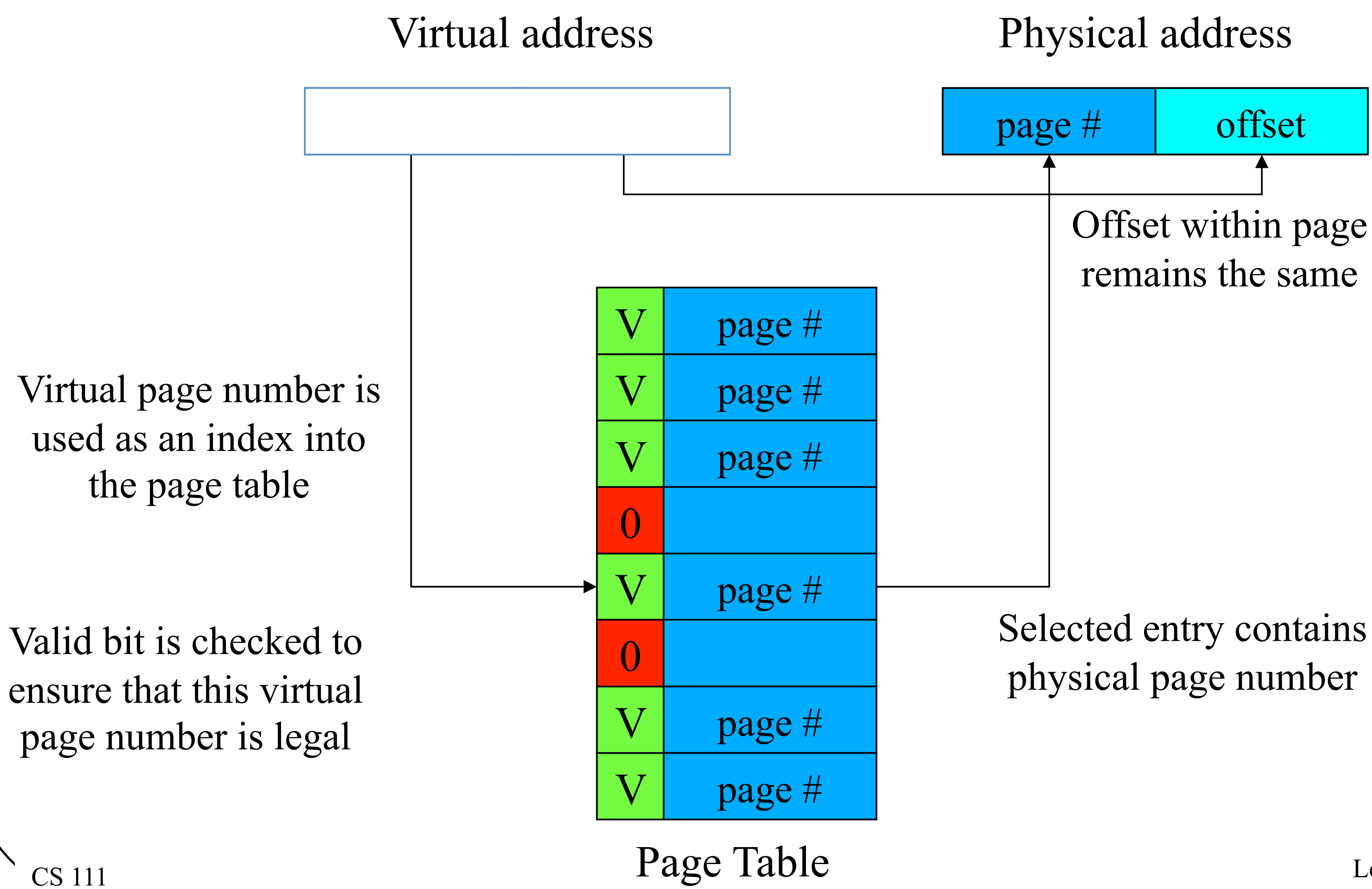
# How Does This Compare To Segment Fragmentation?

- Consider this scenario:
  - Average requested allocation is 128K
  - 256K fixed size segments available
  - In the paging system, 4K pages
- For segmentation, average internal fragmentation is 50% (128K of 256K used)
- For paging?
  - Only the last page of an allocation is not full
  - On average, half of it is unused, or 2K
  - So 2K of 128K is wasted, or around 1.5%
- **Segmentation: 50% waste**
- **Paging: 1.5% waste**

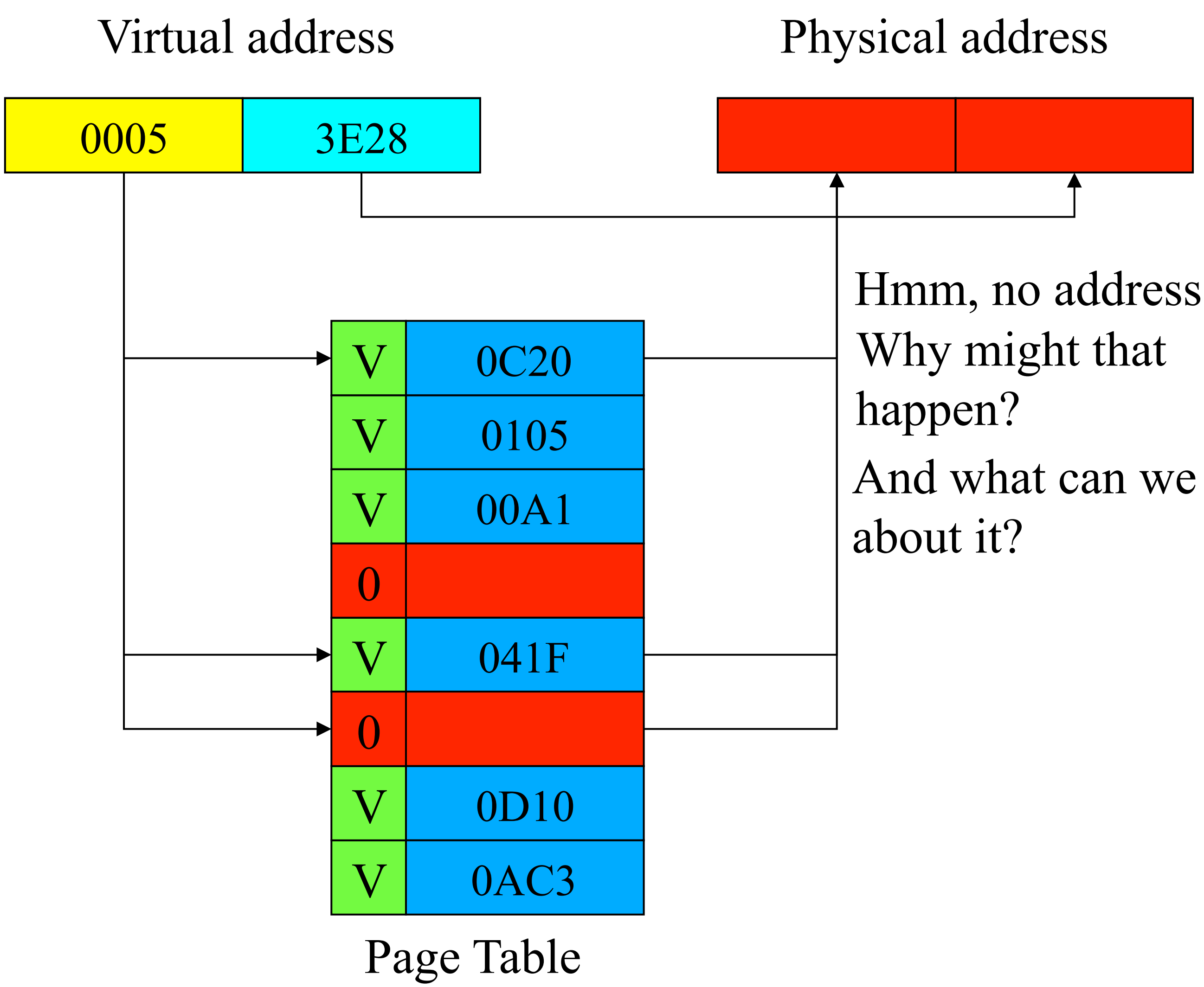
# Providing the Magic Translation Mechanism

- On per page basis, we need to change a virtual address to a physical address
- Needs to be fast
  – So we'll use hardware
- The Memory Management Unit (MMU)
  – A piece of hardware designed to perform the magic quickly

# Paging and MMUs



# Some Examples




Hmm, no address
Why might that happen?
And what can we do about it?

# The MMU Hardware

- MMUs used to sit between the CPU and bus
  - Now they are typically integrated into the CPU
- What about the page tables?
  - Originally implemented in special fast registers
  - But there’s a problem with that today
  - If we have 4K pages, and a 64 Gbyte memory, how many pages are there?
  - $2^{36}/2^{12} = 2^{24}$
  - Or 16 M of pages
  - We can’t afford 16 M of fast registers

# Handling Big Page Tables

- 16 M entries in a page table means we can't use registers
- So now they are stored in normal memory
- But we can't afford 2 bus cycles for each memory access
  - One to look up the page table entry
  - One to get the actual data
- So we have a very fast set of MMU registers used as a cache
  - Which means we need to worry about hit ratios, cache invalidation, and other nasty issues
  - TANSTAAFL

# The MMU and Multiple Processes

- There are several processes running
- Each needs a set of pages
- We can put any page anywhere
- But if they need, in total, more pages than we've physically got,
- Something's got to go
- How do we handle these ongoing paging requirements?

# Ongoing MMU Operations

- What if the current process adds or removes pages?
  - Directly update active page table in memory
  - Privileged instruction to flush (stale) cached entries
- What if we switch from one process to another?
  - Maintain separate page tables for each process
  - Privileged instruction loads pointer to new page table
  - A reload instruction flushes previously cached entries
- How to share pages between multiple processes?
  - Make each page table point to same physical page
  - Can be read-only or read/write sharing

# So Is Paging Perfect?

- Pages are a very nice memory allocation unit
  - They eliminate internal and external fragmentation
  - They require a very simple but powerful MMU
- They are not a particularly natural unit of data
  - Programmers don't think in terms of pages
  - Programs are comprised of, and operate on, segments
  - Segments are the natural "chunks" of virtual address space
    - E.g., we map a new segment into the virtual address space
  - Each code, data, stack segment contains many pages

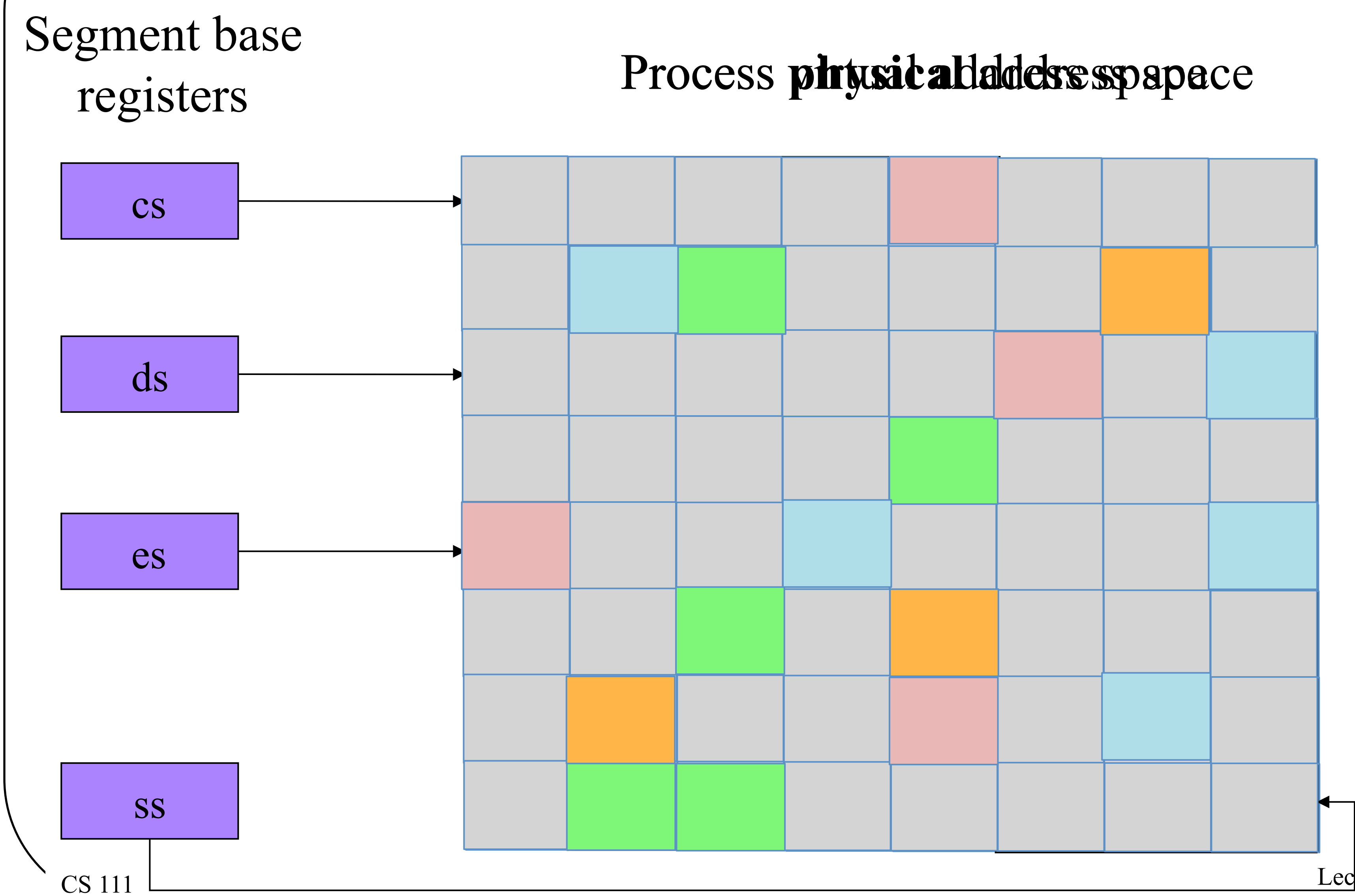
# Paging and Segmentation

- We can use both segments and pages
- Programs request segments
  - Each code, data, stack segment contains many pages
- Requires two levels of memory management abstraction
  - A virtual address space is comprised of segments
  - Relocation & swapping is done on a page basis
  - Segment based addressing, with page based relocation
- User processes see segments, paging is invisible

# Segments and Pages

- A segment is a named collection of pages
  - With contiguous virtual addresses
- Operations on segments:
  - Create/open/destroy
  - Map/unmap segment to/from process
  - Find physical page number of virtual page *n*
- Connection between paging & segmentation
  - Segment mapping implemented with page mapping
  - Page faulting uses segments to find requested page

# Segmentation on Top of Paging



# Swapping

- Segmented paging allows us to have (physically) non-contiguous allocations
  – Virtual addresses in one segment still contiguous
- But it still limits us to the size of physical RAM
- How can we avoid that?
- By keeping some segments somewhere else
- Where?
- Maybe on a disk

# Swapping Segments To Disk

- An obvious strategy to increase effective memory size
- When a process yields, copy its segments to disk
- When it is scheduled, copy them back
- Paged segments mean we need not put any of this data in the same place as before yielding
- Each process could see a memory space as big as the total amount of RAM

# Downsides To Segment Swapping

- If we actually move everything out, the costs of a context switch are <u>very</u> high
  - Copy all of RAM out to disk
  - And then copy other stuff from disk to RAM
  - Before the newly scheduled process can do anything
- We're still limiting processes to the amount of RAM we actually have
  - Even overlays could do better than that

# Demand Paging

- What is paging?
  - What problem does it solve?
  - How does it do so?
- Locality of reference
- Page faults and performance issues

# What Is Demand Paging?

- A process doesn't actually need all its pages in memory to run
- It only needs those it actually references
- So, why bother loading up all the pages when a process is scheduled to run?
- And, perhaps, why get rid of all of a process' pages when it yields?
- Move pages onto and off of disk "on demand"

# How To Make Demand Paging Work

- The MMU must support “not present” pages
  - Generates a fault/trap when they are referenced
  - OS can bring in page and retry the faulted reference
- Entire process needn’t be in memory to start running
  - Start each process with a subset of its pages
  - Load additional pages as program demands them
- The big challenge will be performance

# Achieving Good Performance for Demand Paging

- Demand paging will perform poorly if most memory references require disk access
  – Worse than bringing in all the pages at once, maybe
- So we need to be sure most don’t
- How?
- By ensuring that the page holding the next memory reference is already there
  – Almost always

# Demand Paging and Locality of Reference

- How can we predict what pages we need in memory?
  - Since they'd better be there when we ask
- Primarily, rely on *locality of reference*
  - Put simply, the next address you ask for is likely to be close to the last address you asked for
- Do programs typically display locality of reference?
- Fortunately, yes!

# Reasons Why Locality of Reference Works

- For program instructions?
- For stack access?
- For data access?

# Instruction Locality of Reference

- Code usually executes sequences of consecutive instructions
- Most branches tend to be relatively short distances (into code in the same routine)
- Even routine calls tend to come in clusters
  - E.g., we'll do a bunch of file I/O, then we'll do a bunch of list operations

# Stack Locality of Reference

- Obvious locality here
- We typically need access to things in the current stack frame
  - Either the most recently created one
  - Or one we just returned to from another call
- Since the frames usually aren't huge, obvious locality here

# Heap Data Locality of Reference

- Many data references to recently allocated buffers or structures
  - E.g., creating or processing a message
- Also common to do a great deal of processing using one data structure
  - Before using another
- But more chances for non-local behavior than with code or the stack

# Page Faults

- Page tables no longer necessarily contain points to pages of RAM
- In some cases, the pages are not in RAM, at the moment
  - They're out on disk
- When a program requests an address from such a page, what do we do?
- Generate a *page fault*
  - Which is intended to tell the system to go get it

# Handling a Page Fault

- Initialize page table entries to "not present"
- CPU faults if "not present" page is referenced
  - Fault enters kernel, just like any other trap
  - Forwarded to page fault handler
  - Determine which page is required, where it resides
  - Schedule I/O to fetch it, then block the process
  - Make page table point at newly read-in page
  - Back up user-mode PC to retry failed instruction
  - Return to user-mode and try again
- Meanwhile, other processes can run

# Pages and Secondary Storage

- When not in memory, pages live on secondary storage
  - Typically a disk
  - In an area called "swap space"
- How do we manage swap space?
  - As a pool of variable length partitions?
    - Allocate a contiguous region for each process
  - As a random collection of pages?
    - Just use a bit-map to keep track of which are free
  - As a file system?
    - Create a file per process (or segment)
    - File offsets correspond to virtual address offsets

# Swap Space and Segments

- Should the swap space be organized somehow by segments?
- A paging MMU eliminates need to store consecutive virtual pages in contiguous physical pages
- But locality of reference suggests pages in segments are likely to be used together
- Disk pays a big performance penalty particularly for spreading operations across multiple cylinders
- Well-clustered allocation may lead to more efficient I/O when we are moving pages in and out
- Organizing swap by segments can help

# Demand Paging Performance

- Page faults may result in shorter time slices
  - Standard overhead/response-time tradeoff
- Overhead (fault handling, paging in and out)
  - Process is blocked while we are reading in pages
  - Delaying execution and consuming cycles
  - Directly proportional to the number of page faults
- Key is having the "right" pages in memory
  - Right pages -> few faults, little paging activity
  - Wrong pages -> many faults, much paging
- We can't control what pages we read in
  - Key to performance is choosing which to kick out

# Virtual Memory

- A generalization of what demand paging allows
- A form of memory where the system provides a useful abstraction
  - A very large quantity of memory
  - For each process
  - All directly accessible via normal addressing
  - At a speed approaching that of actual RAM
- The state of the art in modern memory abstractions

# The Basic Concept

- Give each process an address space of immense size
  – Perhaps as big as your hardware's word size allows
- Allow processes to request segments within that space
- Use dynamic paging and swapping to support the abstraction
- The key issue is how to create the abstraction when you don't have that much real memory

# The Key VM Technology: Replacement Algorithms

- The goal is to have each page already in memory when a process accesses it
- We can't know ahead of time what pages will be accessed
- We rely on locality of access
  - In particular, to determine what pages to move out of memory and onto disk
- If we make wise choices, the pages we need in memory will still be there

# The Basics of Page Replacement

- We keep some set of all possible pages in memory
  - Perhaps not all belonging to the current process
- Under some circumstances, we need to replace one of them with another page that's on disk
  - E.g., when we have a page fault
- Paging hardware and MMU translation allows us to choose any page for ejection to disk
- Which one of them should go?

# The Optimal Replacement Algorithm

- Replace the page that will be next referenced furthest in the future
- Why is this the right page?
  - It delays the next page fault as long as possible
  - Fewer page faults per unit time = lower overhead
- A slight problem:
  - We would need an oracle to know which page this algorithm calls for
  - And we don't have one

# Do We Require Optimal Algorithms?

- Not absolutely
- What's the consequence of the algorithm being wrong?
  - We take an extra page fault that we shouldn't have
  - Which is a performance penalty, not a program correctness penalty
  - Often an acceptable tradeoff
- The more often we're right, the fewer page faults we take

# Approximating the Optimal

- Rely on locality of reference
- Note which pages have recently been used
  - Perhaps with extra bits in the page tables
  - Updated when the page is accessed
- Use this data to predict future behavior
- If locality of reference holds, the pages we accessed recently will be accessed again soon

# Candidate Replacement Algorithms

- Random, FIFO
  - These are dogs, forget ‘em
- Least Frequently Used
  - Sounds better, but it really isn’t
- Least Recently Used
  - Assert that near future will be like the recent past
  - If we haven’t used a page recently, we probably won’t use it soon
  - The computer science equivalent to the “*unseen hand*”

# How To Evaluate Page Replacement Algorithms

- We can't predict the future, so we approximate
- Which algorithm approximates best?
- Based on the number of page faults each gets while executing a standard test
- What should the standard test be?
  - Different algorithms will behave very differently in different situations
- To test replacement algorithms, you need a clear notion of what your load is like

# Naïve LRU

- Each time a page is accessed, record the time
- When you need to eject a page, look at all timestamps for pages in memory
- Choose the one with the oldest timestamp
- Will require us to store timestamps somewhere
- And to search all timestamps every time we need to eject a page

# True LRU Page Replacement

Reference stream

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | a | b | d | e | f | a | b | c | d | a | e | d |

Page table using true LRU

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 0 | a | | | | ! | | | | f | | | | d | | | ! |
| frame 1 | | b | | | | ! | | | | a | | | | ! | | |
| frame 2 | | | c | | | | | e | | | | c | | | | |
| frame 3 | | | | d | | | ! | | | | b | | | | e | |

**Loads 4**

**Replacements 7**

# Maintaining Information for LRU

- Can we keep it in the MMU?
  - MMU notes the time whenever a page is referenced
  - MMU translation must be blindingly fast
    - Getting/storing time on every fetch would be very expensive
  - At best they will maintain a *read* and a *written* bit per page
- Can we maintain this information in software?
  - Mark all pages invalid, even if they are in memory
  - Take a fault first time each page is referenced, note the time
  - Then mark this page valid for the rest of the time slice
  - Causing page faults to reduce the number of page faults???
- We need a <u>cheap</u> software surrogate for LRU
  - No extra page faults
  - Can't scan entire list each time, since it's big

# Clock Algorithms

- A surrogate for LRU
- Organize all pages in a circular list
- MMU sets a reference bit for the page on access
- Scan whenever we need another page
  - For each page, ask MMU if page has been referenced
  - If so, reset the reference bit in the MMU & skip this page
  - If not, consider this page to be the least recently used
  - Next search starts from this position, not head of list
- Use position in the scan as a surrogate for age
- No extra page faults, usually scan only a few pages

# Clock Algorithm Page Replacement

Reference Stream

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | a | b | d | e | f | a | b | c | d | a | e | d |

LRU clock

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 0 | a | | | | ! | ! | ! | ! | f | | | | d | | | ! |
| frame 1 | | b | | | | ! | ! | ! | | a | | | | ! | ! | |
| frame 2 | | | c | | | | | e | | | b | | | | e | |
| frame 3 | | | | d | | | ! | ! | ! | | | c | | | | |
| clock pos | 0 | 1 | 2 | 3 | 0 | 0 | 0 | 0/2 | 0/3 | 1 | 2 | 3 | 0 | 1 | 1/2 | 3 |

loads 4, replacements 7

True LRU

Loads 4

Replacements 7

# Comparing True LRU To Clock Algorithm

- Same number of loads and replacements
  - But didn't replace the same pages
- What, if anything, does that mean?
- Both are just approximations to the optimal
- If LRU clock's decisions are 98% as good as true LRU
  - And can be done for 1% of the cost (in hardware and cycles)
  - It's a bargain!

# Page Replacement and Multiprogramming

- We don’t want to clear out all the page frames on each context switch
- How do we deal with sharing page frames?
- Possible choices:
  - Single global pool
  - Fixed allocation of page frames per process
  - Working set-based page frame allocations

# Single Global Page Frame Pool

- Treat the entire set of page frames as a shared resource
- Approximate LRU for the entire set
- Replace whichever process' page is LRU
- Probably a mistake
  - Bad interaction with round-robin scheduling
  - The guy who was last in the scheduling queue will find all his pages swapped out
  - And not because he isn't using them
  - When he gets in, lots of page faults

# Per-Process Page Frame Pools

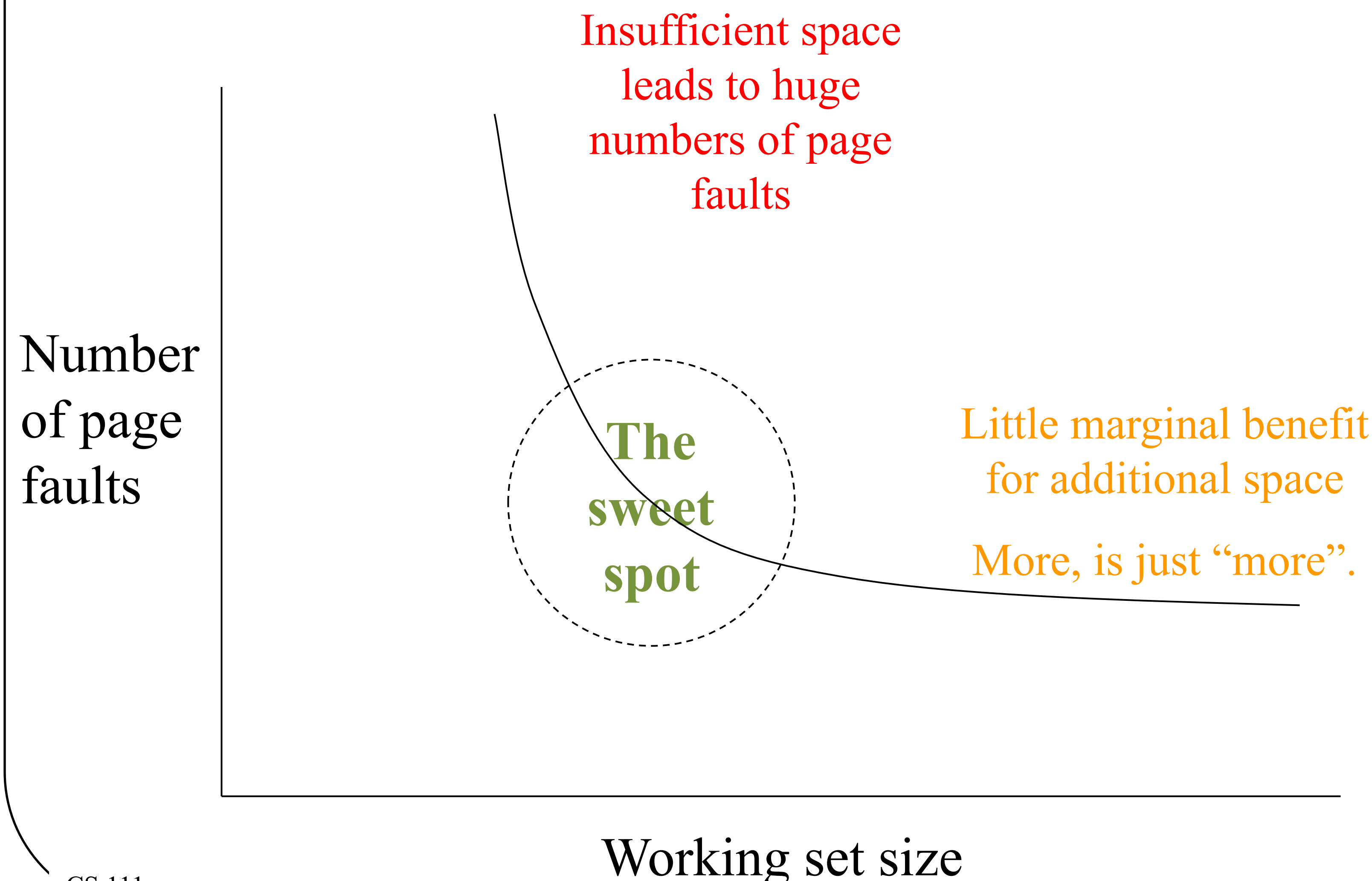
- Set aside some number of page frames for each running process
  - Use an LRU approximation separately for each
- How many page frames per process?
- Fixed number of pages per process is bad
  - Different processes exhibit different locality
    - Which pages are needed changes over time
    - Number of pages needed changes over time
  - Much like different natural scheduling intervals
- We need a dynamic customized allocation

# Working Sets

- Give each running process an allocation of page frames matched to its needs
- How do we know what its needs are?
- Use *working sets*
- Set of pages used by a process in a fixed length sampling window in the immediate past[1]
- Allocate enough page frames to hold each process' working set
- Each process runs replacement within its own set

[1]This definition paraphrased from Peter Denning's definition

# The Natural Working Set Size



# Optimal Working Sets

- What is optimal working set for a process?
  - Number of pages needed during next time slice
- What if try to run the process in fewer pages?
  - Needed pages will replace one another continuously
  - This is called *thrashing*
- How can we know what working set size is?
  - By observing the process' behavior
- Which pages should be in the working-set?
  - No need to guess, the process will fault for them

# Implementing Working Sets

- Manage the working set size
  - Assign page frames to each in-memory process
  - Processes page against themselves in working set
  - Observe paging behavior (faults per unit time)
  - Adjust number of assigned page frames accordingly
- Page stealing algorithms
  - E.g., Working Set-Clock
  - Track last use time for each page, for owning process
  - Find page least recently used (by its owner)
  - Processes that need more pages tend to get more
  - Processes that don't use their pages tend to lose them

# Working Set Clock Algorithm

| page frame | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| referenced | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| process | $P_0$ | $P_0$ | $P_1$ | $P_2$ | $P_2$ | $P_1$ | $P_1$ | $P_0$ | $P_2$ | $P_0$ | $P_1$ | $P_2$ | $P_0$ | $P_1$ | $P_2$ |
| last ref | 15 | 51 | 69 | 65 | 80 | 15 | 75 | 33 | 72 | 54 | 23 | 25 | 45 | 25 | 47 |

Clock pointer (at page frame 6)

current execution times: $P_0 = 55$ | $P_1 = 75$ | $P_2 = 80$ — $t = 15$

## $P_0$ gets a fault

- page 6 was just referenced
  - clear ref bit, update time
- page 7 is (55-33=22) ms old
  - $P_0$ replaces his own page

# Stealing a Page

| Page frame | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| referenced | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| process | $P_0$ | $P_0$ | $P_1$ | $P_2$ | $P_2$ | $P_1$ | $P_1$ | $P_0$ | $P_2$ | $P_0$ | $P_0$ | $P_2$ | $P_0$ | $P_1$ | $P_2$ |
| last ref | 15 | 51 | 69 | 65 | 80 | 15 | 75 | 33 | 72 | 54 | | 25 | 45 | 25 | 47 |

Clock pointer

current execution times $P_0 = 55$ $P_1 = 75$ $P_2 = 80$ t = 25

$P_0$ has been experiencing too many page faults recently

$P_0$ gets a fault

page 6 was just referenced

page 7 is (55-33=22) ms old

page 8 is (80-72=8) ms old

page 9 is (55-54=1) ms old

page 10 is (75-23=52) ms old

$P_0$ steals this page from $P_1$

# Thrashing

- Working set size characterizes each process
  - How many pages it needs to run for τ milliseconds
- What if we don't have enough memory?
  - Sum of working sets exceeds available memory
  - We will thrash unless we do something
- We cannot squeeze working set sizes
  - This will also cause thrashing
- Reduce number of competing processes
  - Swap some of the ready processes out
  - To ensure enough memory for the rest to run
- We can round-robin who is in and out

# Pre-Loading

- What happens when a process comes in from disk?
- Pure swapping
  - All pages present before process is run, no page faults
- Pure demand paging
  - Pages are only brought in as needed
  - Fewer pages per process, more processes in memory
- What if we pre-loaded the last working set?
  - Far fewer pages to be read in than swapping
  - *Probably* the same disk reads as pure demand paging
  - Far fewer initial page faults than pure demand paging

# Doesn’t This Compromise LRU?

- Why wouldn’t a process already have its working set in memory?
- Because it was swapped out to prevent thrashing
- Pre-fetching is not trying to be smarter than LRU
- It is merely resetting a process’ working set to what it was before we swapped it out

# Clean Vs. Dirty Pages

- Consider a page, recently paged in from disk
  - There are two copies, one on disk, one in memory
- If the in-memory copy has not been modified, there is still a valid copy on disk
  - The in-memory copy is said to be "clean"
  - Clean pages can be replaced without writing them back to disk
- If the in-memory copy has been modified, the copy on disk is no longer up-to-date
  - The in-memory copy is said to be "dirty"
  - If swapped out of memory, must be written to disk

# Dirty Pages and Page Replacement

- Clean pages can be replaced at any time
  - The copy on disk is already up to date
- Dirty pages must be written to disk before the frame can be reused
  - A slow operation we don't want to wait for
- Could only swap out clean pages
  - But that would limit flexibility
- How to avoid being hamstrung by too many dirty page frames in memory?

# Pre-Emptive Page Laundering

- Clean pages give memory scheduler flexibility
  - Many pages that can, if necessary, be replaced
- We can increase flexibility by converting dirty pages to clean ones
- Ongoing background write-out of dirty pages
  - Find and write-out all dirty, non-running pages
    - No point in writing out a page that is actively in use
  - On assumption we will eventually have to page out
  - Make them clean again, available for replacement
- An outgoing equivalent of pre-loading

# Paging and Shared Segments

- Some memory segments will be shared
  - Shared memory, executables, DLLs
- Created/managed as mappable segments
  - One copy mapped into multiple processes
  - Demand paging same as with any other pages
  - Secondary home may be in a file system
- Shared pages don't fit working set model
  - May not be associated with just one process
  - Global LRU may be more appropriate
  - Shared pages often need/get special handling

# The OS and Virtual Memory

- The OS needs physical memory of its own
- How does that fit into the VM model?
- Kernel address space may be virtual or physical
  - Includes all system code and data structures
  - Also includes mapped I/O space
- Physical memory divided into two classes
  - Most managed as pages, for use by processes
  - Some managed as storage heap for kernel allocation

# Moving Data Between Kernel and User Spaces

- Kernel often needs to access user data
  - To access system call parameters
  - To perform read and write system calls
- Kernel may run in a virtual address space
  - Which includes current process' address space
- Kernel may execute with physical addresses
  - Software translation of user-space addresses

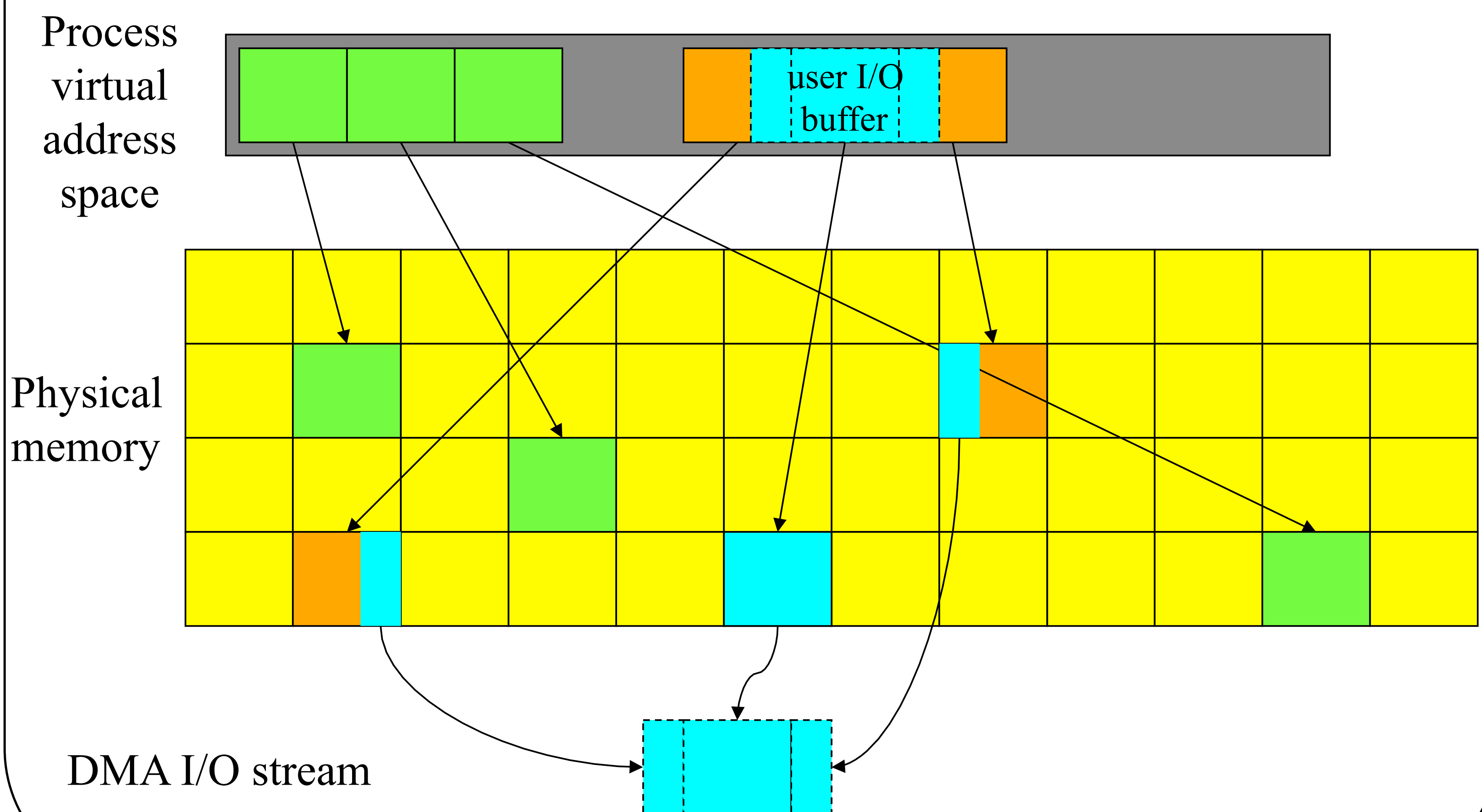
# Virtual Memory and I/O

- User I/O requests use virtual buffer address
  - How can a device controller find that data?
- Kernel can copy data into physical buffers
  - Accessing user data through standard mechanisms
- Kernel may translate virtual to physical
  - Give device the corresponding physical address
- CPU may include an I/O MMU
  - Page tables to translate virtual addrs to physical
  - All DMA I/O references go through the I/O MMU

# Scatter/Gather I/O

- Many controllers support DMA transfers
  - Entire transfer must be contiguous in physical memory
- User buffers are in paged virtual memory
  - User buffer may be spread all over physical memory
  - *Scatter*: read from device to multiple pages
  - *Gather*: writing from multiple pages to device
- Same three basic approaches are possible
  1. Copy all user data into contiguous physical buffer
  2. Split logical request into chain-scheduled page requests
  3. I/O MMU may automatically handle scatter/gather

# Gather Writes From User Memory



# Scatter Reads Into User Buffer