# Virtual Memory Design and Implementation

To do …

- Page replacement algorithms
- Design and implementation issues
- Next: Last on virtualization – VMMs

# Loading pages

- When should the OS load pages?
  - On demand or ahead of need
- Demand paging
  - Only load when reference
  - Just the code/data needed has to be loaded
    - But needs change over time…
- Anticipatory paging (prefetching)
  - If you know/can guess the pages a process will need
- Few systems try to anticipate future needs
  - The OS is not very good at prediction

# And if the page is not there

- If a process references a virtual address in an evicted o never loaded page …
  - When page was evicted, OS set PTE as invalid and noted disk location of page
    - In a data structure ~page table but holding disk addresses
  - When process tries to access page, page fault
  - OS runs the page fault handler
    - Handler uses the "~PT" data structure to find page on disk
    - … reads page in, updates PTE to point to it, set it to valid
    - OS restarts the faulting process
    - … there are a million and one details …

# Page replacement algorithms

- Need room for new page? Page replacement
- What do you do with the victim page?
  - If modified, save it, otherwise just write over it
  - Better not to choose an often used page
- How can any of this work?!?!
  - Locality!

# Locality!

- Temporal and spatial locality
  - Temporal – Recently referenced locations tend to be referenced again soon
  - Spatial – Referenced locations tend to be clustered
- Locality means paging could be infrequent
  - A page brought in, gets to be used many times
  - Some issues that may play against this
    - Degree of locality of application
    - Page replacement policy and application reference pattern
    - Amount of physical memory and application footprint

# The best page to evict

- Goal of the page replacement algorithm
  - Reduce fault rate by selecting best victim page
  - What’s your best candidate for removal?
    - The one you will never touch again – duh!
  - “Never” is a long time
    - For Belady’s algorithm – let’s say for the longest period

- Let’s look at some algorithms
  - for now, assume that a process pages against itself, using a fixed number of page frames

# Optimal algorithm (Belady's algorithm)

- Provably optimal
  - Replace page needed at the farthest point in future
  - Optimal but unrealizable
- Estimate by …
  - Logging page use on previous runs of process
  - Although impractical, useful fo on

Need room for this one!

You ideal victim!

| Reference | A | B | A | C | B | D | A | D | E | D | A | E | B | A | C |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | A | | + | | | | + | | | | | | | | |
| 2 | | B | | | + | | | | | | | | | | |
| 3 | | | | C | | | | | | | | | | | |
| 4 | | | | | | D | | + | | | | | | | |

Four page frames

Reference stream

Compulsory misses (others: capacity misses)

# FIFO algorithm

- Keep a linked list of pages – in order of arrival
- Victim is first page of list
  - Maybe the oldest page will not be used again …
- Disadvantage
  - But maybe it will – the fact is, you have no idea!
  - Increasing physical memory might increase page faults (Belady's anomaly)

Need room for this one!

| Reference | A | B | A | C | B | D | A | D | E | D | A | E | B | A | C |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | A | | + | | | | + | | E | | | + | | | |
| 2 | | | | | + | | | | | | A | | | + | |
| 3 | | | | C | | | | | | | | | B | | |
| 4 | | | | | | D | | + | | + | | | | | C |

First in

# Least recently used (LRU) algorithm

- Pages used recently will be used again soon
  - Throw out page unused for longest time
  - Idea: past experience as a predictor of future needs
    - LRU looks at the past, Belady's wants to look at the future
    - How is LRU different from FIFO?
- Must keep a linked list of pages
  - Most recently used at front, least recently used last
  - Update list with every memory reference!!
    - $$$ in mem. bandwidth, algorithm execution time, etc

# Second chance algorithm

- Simple modification of FIFO
  - Avoid removing heavily used pages – check the R bit
- Second chance
  - Pages sorted in FIFO order
  - If page has been used, gets another chance – move it to the end of the list of pages, clear R, update timestamp
  - Page list if fault occurs at time 20, A has R bit set (time is loading time)

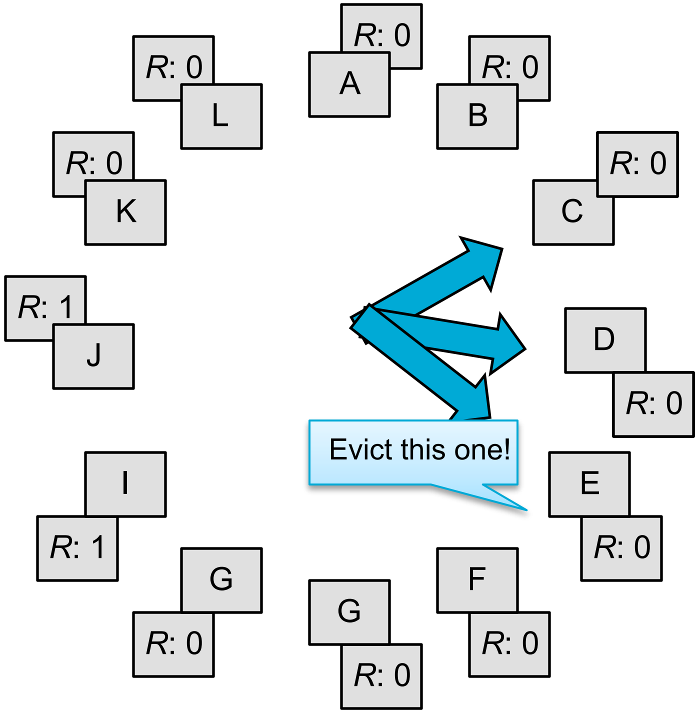
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Page ID | A | H | G | F | E | D | C | B | A |
| Timestamp | 20 | 18 | 15 | 14 | 12 | 8 | 7 | 3 | 0 |
| R bit | 0 | X | X | X | X | X | X | 0 | 1 |

Most recently loaded

Oldest page

# Clock algorithm

- Second chance is reasonable but inefficient
  - Quit moving pages around – move a pointer?
- Clock – ~Second chance but for implementation
  - Keep all pages in a circular list, as a clock, with the hand pointing to the oldest page
  - When page fault
    - Look at page pointed at by hand
      - If R = 0, evict page
      - If R = 1. clear R & move hand



# Not recently used (NRU) algorithm

- Each page has Reference and Modified bits
  - Set when page is referenced, modified
  - R bit set means recently referenced, so clear it regularly

- Pages are classified

*How can this occur?*

| R | M | Class |
|---|---|---|
| 0 | 0 | Not referenced, not modified (0,0 → 0) |
| 0 | 1 | Not referenced, modified (0,1 → 1) |
| 1 | 0 | Referenced, but not modified (1,0 → 2) |
| 1 | 1 | Referenced and modified (1,1 → 3) |

- NRU removes page at random
  - from lowest numbered, non-empty class
- Easy to understand, relatively efficient to implement and sort-of OK performance

# Approximating LRU

- With some extra help from hardware
  - Keep a counter in PTE
  - Equipped hardware with a counter, ++ after each instruction
  - After each reference, update PTE counter for the referenced with hardware counter
  - Choose page with lowest value counter

- In software, Not Frequently Used
  - Software counter associated with each page
  - At clock interrupt – add R to counter for each page
  - Problem - it never forgets!
    - A page heavily used early on and never touch again, will keep a high count for a long time and not be evicted

# Approximating LRU

- Better, with a small modification – Aging
  - Push R from the left, drop bit on the right
  - How is this not LRU? One bit per tick & a finite number of bits per counter
    - Precision – all pages referenced within the interval are the same
    - Limited past horizon – two pages with all 8 bits unset are the same, even if one dropped a set 9th bit just before

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

| R | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 4 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

| R | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 4 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

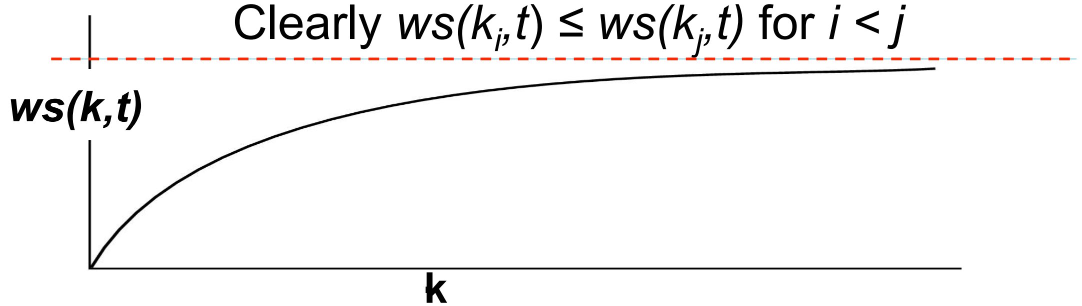
# Working set

- Most programs show locality of reference
  - Over a short time, just a few common pages
- Working set
  - Models the dynamic locality of a process’ memory usage
  - i.e. the set of pages currently needed by a process
- Intuitively, working set must be in memory, otherwise you’ll experience heavy faulting (thrashing)
  - What does it mean ‘how much memory does program x need?”
  - What is program x average/worst-case working set size?

# Working set

- Demand paging
  - Simplest strategy, load page when needed
  - Can you do better knowing a process WS?
  - Can you use this to reduce turnaround time? Pre-paging
- Working set definition
  - ws($k$,$t$) = {$p$ such that $p$ was referenced in the $k$ most recent memory references at time $t$} ($k$ is WS window size)

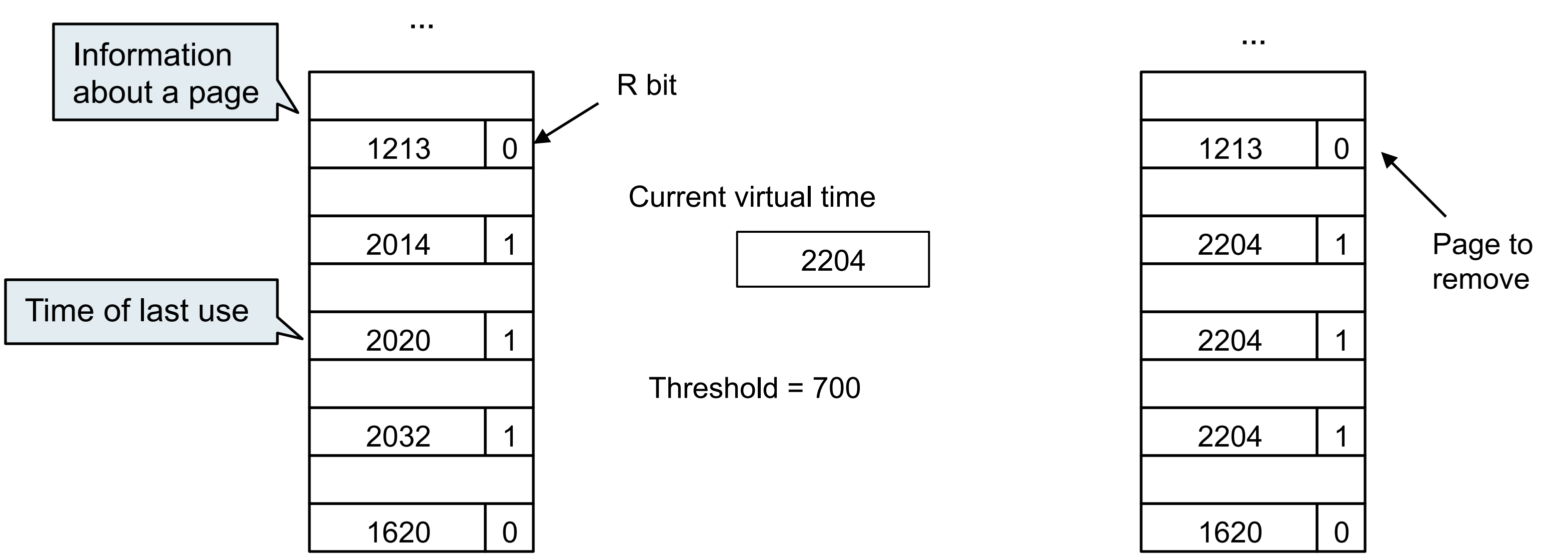
*What bounds ws(k, t) as you increase k?*




  - A more practical definition – instead of $k$ reference pages, τ msec of execution time (virtual time)

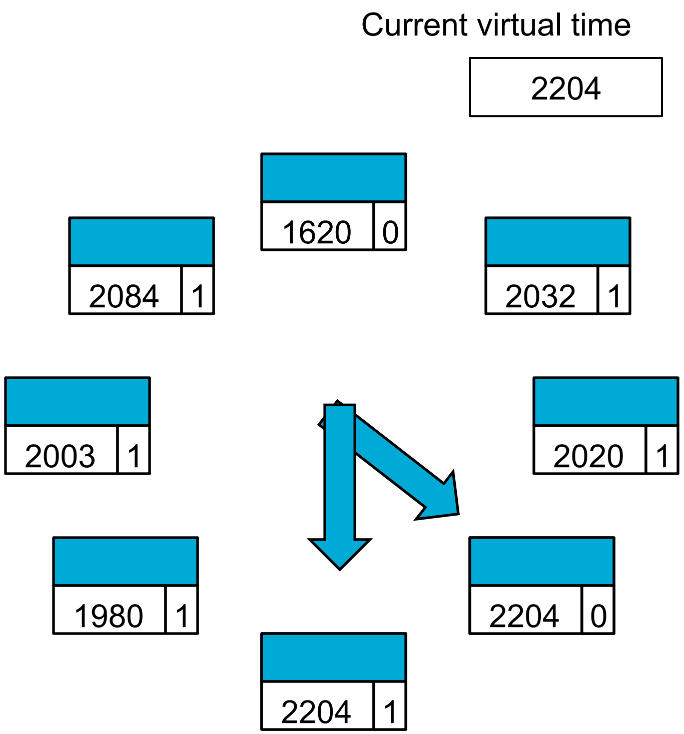
# Working set algorithm

- Working set and page replacement
  - Victim – a page not in the working set
- At each clock interrupt – scan the page table
  - R = 1? Write Current Virtual Time (CVT) into Time of Last Use
  - R = 0? CVT – Time of Last Use > Threshold ? out!
    - Else see if there's some other and evict oldest (w/ R=0)
  - If all are in the WS (all R = 1), random, preferably clean



# WSClock algorithm

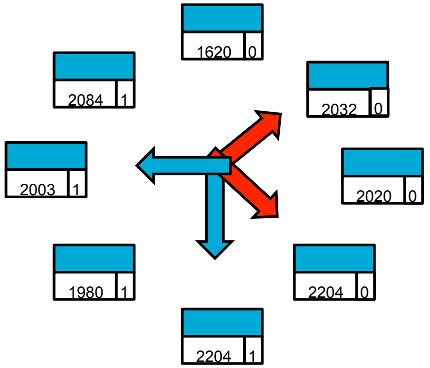
- Problem with WS algorithm – Scans the whole table
- Instead, scan only what you need to find a victim
- Combine clock & working set
  - If R = 1, unset it
  - If R = 0, if age > T and page clean, out
  - If dirty, schedule write and check next one
  - If loop around,
    - There’s 1+ write scheduled – you’ll have a clean page soon
    - There’s none, pick any one




*R = 0 & 2204 – 1213 > T*

# Cleaning policy

- To avoid having to write pages out when needed – paging daemon
  - Periodically inspects state of memory
  - Keep enough pages free
  - If we need the page before it's overwritten – reclaim it!
- Two hands for better performance (BSD)
  - First one clears R, second checks it
  - If hands are close, only heavily used pages have a chance
  - If back is just ahead of front hand (359°), original clock
  - Two key parameters, adjusted at runtime
    - Scanrate – rate at which hands move through the list
    - Handspread – gap between them



# Design issues – global vs. local policy

- When you need a page frame, pick a victim from
  - Among your own resident pages – Local
  - Among all pages – Global
- Local algorithms
  - Basically every process gets a fixed % of memory
- Global algorithms
  - Dynamically allocate frames among processes
  - Better, especially if working set size changes at runtime
  - How many page frames per process?
    - Start with basic set & react to Page Fault Frequency (PFF)
- Most replacement algorithms can work both ways except those based on working set
  - Why not working set based algorithms?

# Load control

- Despite good designs, system may still thrash
  – Sum of working sets > physical memory
- How do you know? Page Fault Frequency (PFF)
  – Some processes need more memory
  – But no process needs less …
- Way out: Swapping
  – So yes, even with paging you still need swapping
  – Reduce number of processes competing for memory
  – ~ two-level scheduling – careful with which process to swap out (there's more than just paging to worry about!)
  – What would you like of the remaining processes?

# Backing store

- How do we manage swap area?
  - Allocate space to process when started
  - Keep offset to process swap area in PCB
  - Process can be brought entirely when started or as needed

- Some problems
  - Size – process can grow … split text/data/stack segments in swap area
  - Do not allocate anything … you may need extra memory to keep track of pages in swap!

# Page fault handling

- Hardware traps to kernel
- General registers saved by assembler routine, OS called
- OS find which virtual page cause the fault
- OS checks address is valid, seeks page frame
- If selected frame is dirty, write it to disk (CS)
- Get new page (CS), update page table
- Back up instruction where interrupted
- Schedule faulting process
- Routine load registers & other state and return to user space

# Instruction backup

- With a page fault, the current instruction is stopped part way through … harder than you think!
  - Consider instruction: MOV.L #6(A1), 2(A0)

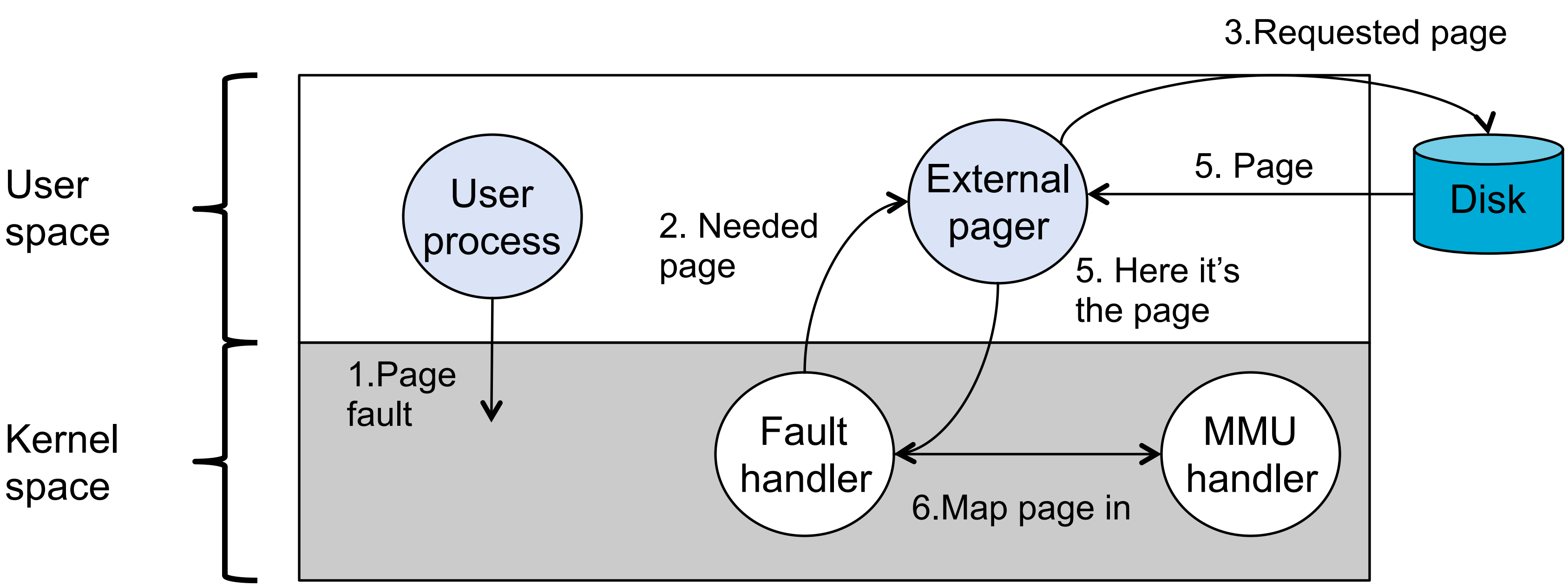
*One instruction, three memory references (instruction word itself, two offsets for operands*

| | |
|---|---|
| **1000** | **MOVE** |
| 1002 | 6 |
| 1004 | 2 |

  - Which one caused the page fault? What's the PC then?
  - Worse – autodecr/incr as a side-effect of execution?
- Some CPU design include hidden registers to store
  - Beginning of instruction
  - Indicate autodecr./autoincr. and amount

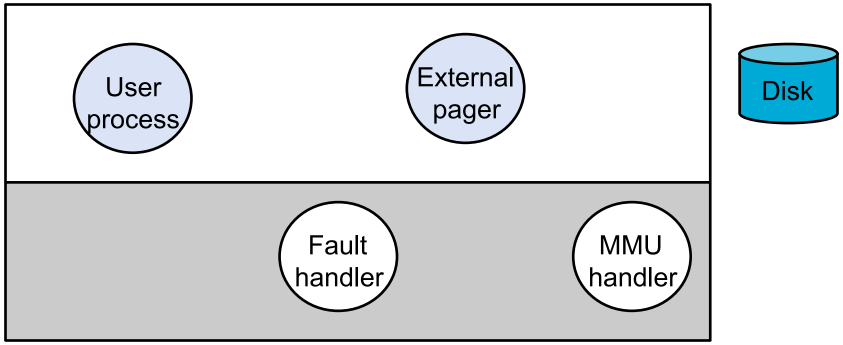
# Last: Separating policy & mechanism

- How to structure the memory management system for easy separation? (based on Mach)
  - 1. Low-level MMU handler – machine dependent
  - 2. Page-fault handler in kernel – machine independent, most of paging mechanism
  - 3. External pager running in user space – policy is here



# Separation of policy & mechanism

- Where should we put the page replacement algorithm?
  - Cleanest: external pager, but no access to R and M bits
    - Either pass this info to the pager or
    - Fault handler informs external pager which page is victim




- Pros and cons
  - More modular, flexible
  - Overhead of crossing user-kernel + msg exchange
  - As HW get faster and SW more complex …

# Next time

- Virtualize the CPU, virtualize memory, …
- Let’s virtualize the whole machine

# And now a short break …