# Virtual Memory Systems

Chapter 12

# Key concepts in chapter 12

- Page replacement algorithms
  - global
  - local
- Working set
- Load control
- Large page tables
  - two and three level paging
  - inverted page tables
- Segmentation

# Page replacement

- *Placement* is simple in paging systems (just put a page in a page frame)
  - but hard in segmentation systems
- The problem is *replacement*
  - which page to remove from memory to make room for a new page
- We need a *page replacement algorithm*
- Two categories of replacement algorithms
  - *local* algorithms always replace a page from the process that is bringing in a new page
  - *global* algorithm can replace any page

# Page references

- Processes continually reference memory
  - and so generate a sequence of page references
- The *page reference sequence* tells us everything about how a process uses memory
- We use page reference sequences to evaluate paging algorithms
  - we see how many page faults the algorithm generates on a particular page reference sequence with a given amount of memory

# Evaluating page replacement algorithms

- The goal of a page replacement algorithm is to produce the fewest page faults
- We can compare two algorithms
  - on a range of page reference sequences
- Or we can compare an algorithm to the best possible algorithm
- We will start by considering *global* page replacement algorithms

# Optimal replacement algorithm

- The one that produces the fewest possible page faults on all page reference sequences
- *Algorithm*: replace the page that will not be used for the longest time in the future
- *Problem*: it requires knowledge of the future
- Not realizable in practice
  - but it is used to measure the effectiveness of realizable algorithms

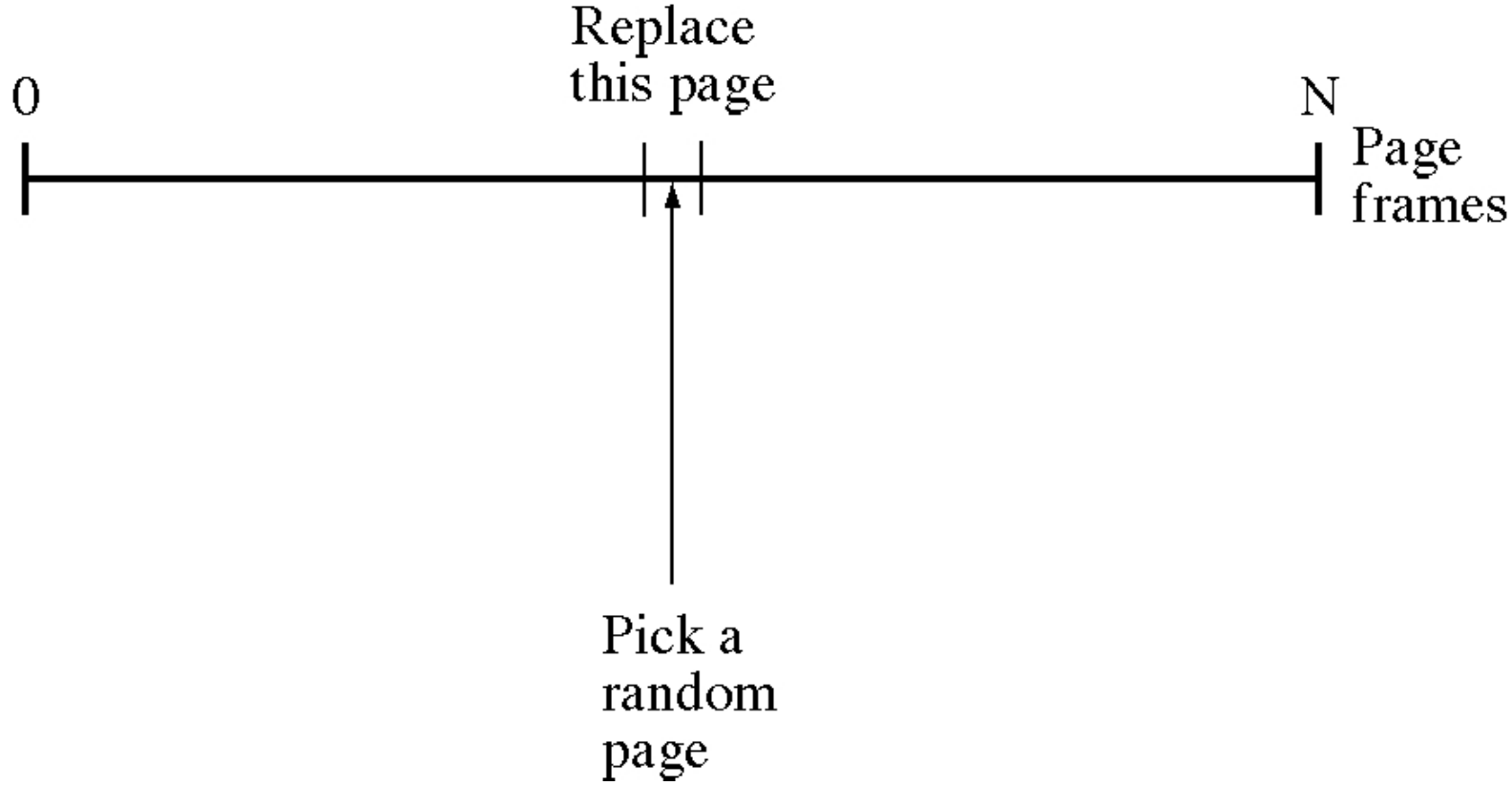
# Theories of program behavior

- All replacement algorithms try to predict the future and act like the optimal algorithm
- All replacement algorithms have a theory of how program behave
  - they use it to predict the future, that is, when pages will be referenced
  - then the replace the page that they think won't be referenced for the longest time.

# Random page replacement

- *Algorithm*: replace a random page
- *Theory*: we cannot predict the future at all
- *Implementation*: easy
- *Performance*: poor
  - but it is easy to implement
  - but the best case, worse case and average case are all the same

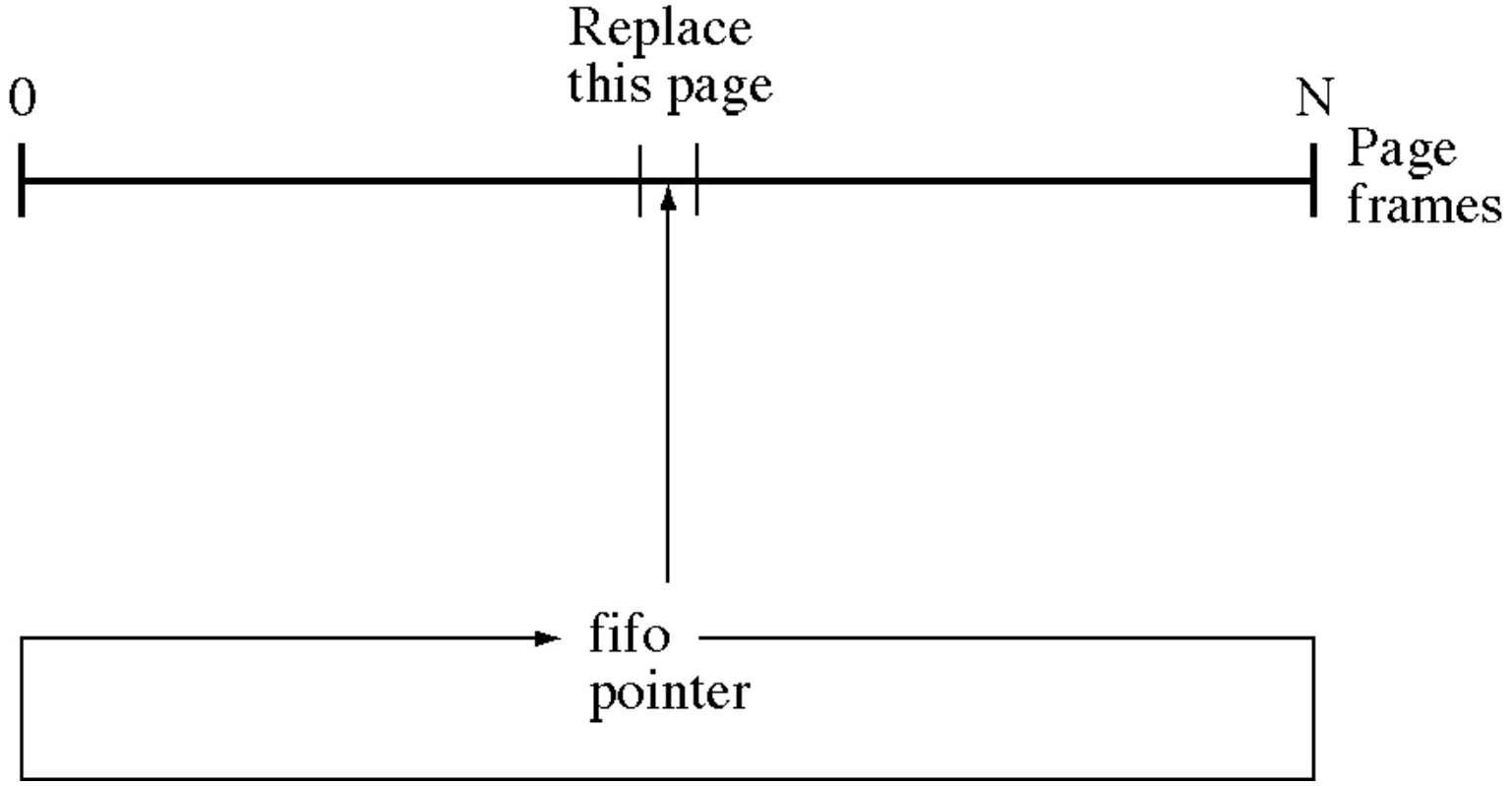
# Random page replacement



# FIFO page replacement

- *Algorithm*: replace the oldest page
- *Theory*: pages are use for a while and then stop being used
- *Implementation*: easy
- *Performance*: poor
  - because old pages are often accessed, that is, the theory if FIFO is not correct

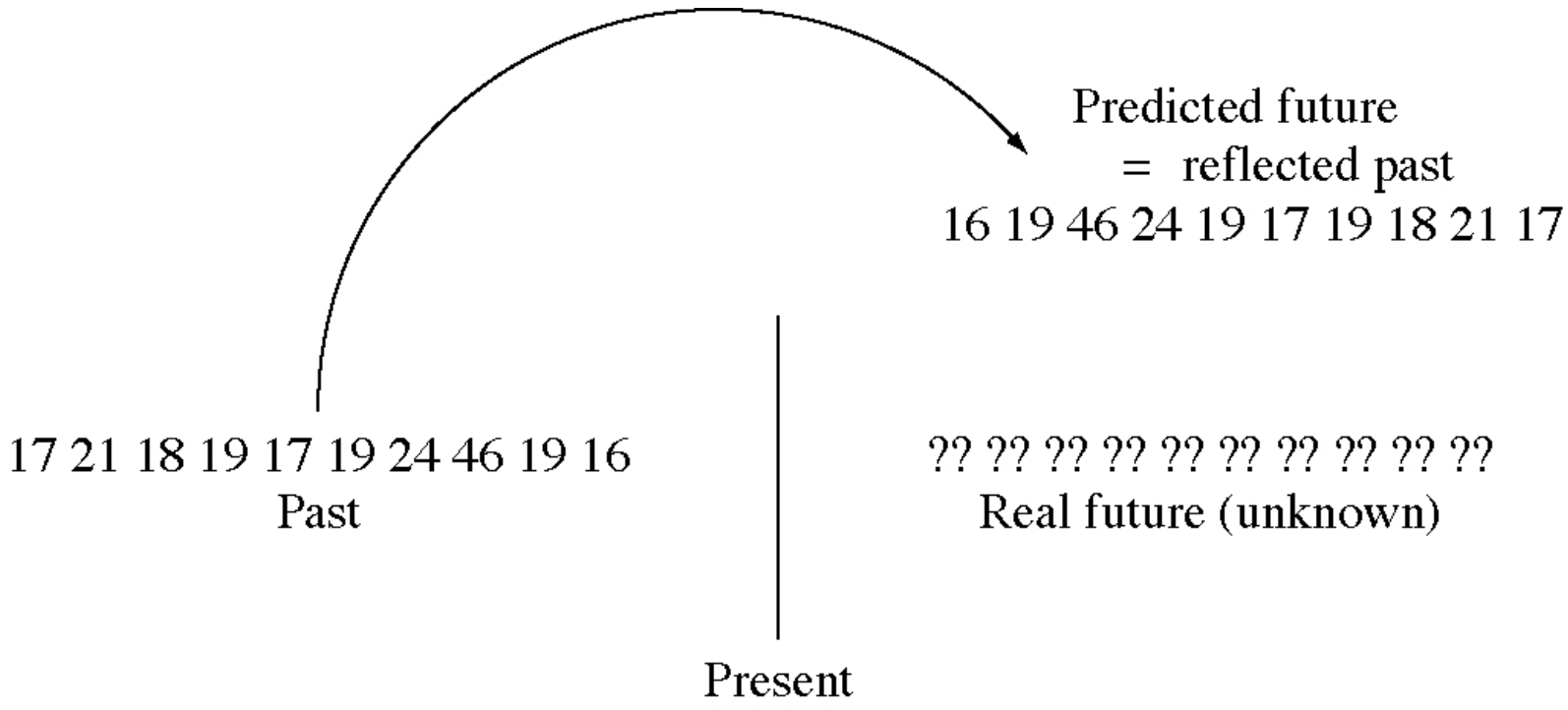
# FIFO page replacement



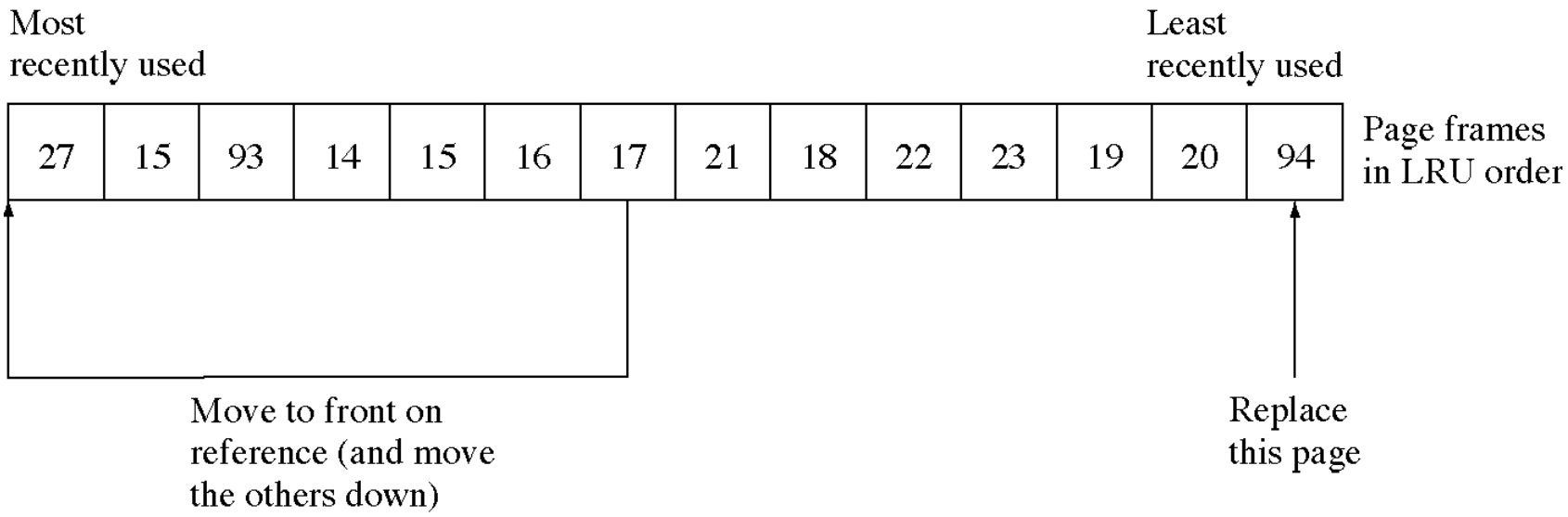
# LRU page replacement

- Least-recently used (LRU)
- *Algorithm*: remove the page that hasn't been referenced for the longest time
- *Theory*: the future will be like the past, page accesses tend to be clustered in time
- *Implementation*: hard, requires hardware assistance (and then still not easy)
- *Performance*: very good, within 30%-40% of optimal

# LRU model of the future



# LRU page replacement



# Approximating LRU

- LRU is difficult to implement
  - usually it is approximated in software with some hardware assistance
- We need a referenced bit in the page table entry
  - turned on when the page is accessed
  - can be turned off by the OS (with a privileged instruction)
    - there are instructions to read and write the accessed bit (and often to turn off all the accessed bits)

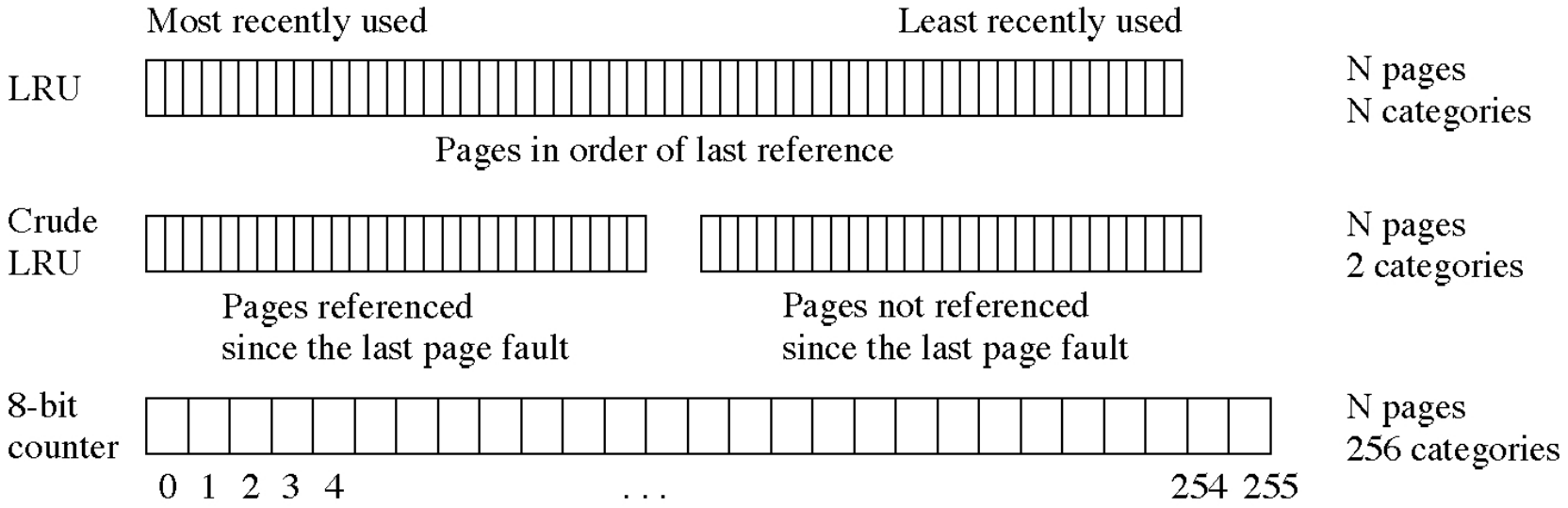
# First LRU approximation

- When you get a page fault
  - replace any page whose referenced bit is off
  - then turn off all the referenced bits
- Two classes of pages
  - Pages referenced since the last page fault
  - Pages not referenced since the last page fault
    - the least recently used page is in this class but you don’t know which one it is
- A crude approximation of LRU

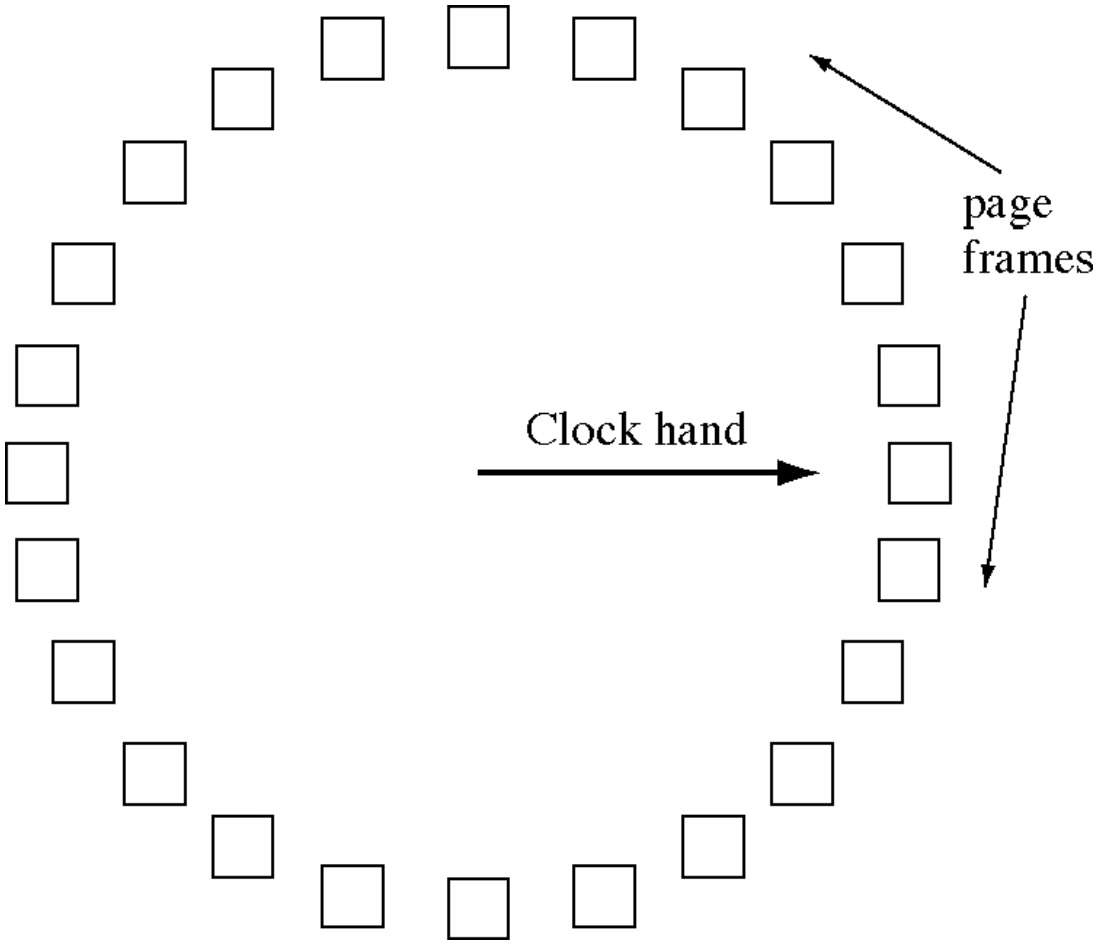
# Second LRU approximation

- Algorithm:
  - Keep a counter for each page
  - Have a daemon wake up every 500 ms and
    - add one to the counter of each page that has not been referenced
    - zero the counter of pages that have been referenced
    - turn off all referenced bits
  - When you get a page fault
    - replace the page whose counter is largest
- Divides pages into 256 classes

# LRU and its approximations



# A clock algorithm



# Clock algorithms

- Clock algorithms try to approximate LRU but without extensive hardware and software support
- The page frames are (conceptually) arranged in a big circle (the clock face)
- A pointer moves from page frame to page frame trying to find a page to replace and manipulating the referenced bits
- We will look at three variations
- FIFO is actually the simplest clock algorithm

# Basic clock algorithm

- When you need to replace a page
  - look at the page frame at the clock hand
  - if the referenced bit = 0 then replace the page
  - else set the referenced bit to 0 and move the clock hand to the next page
- Pages get one clock hand revolution to get referenced

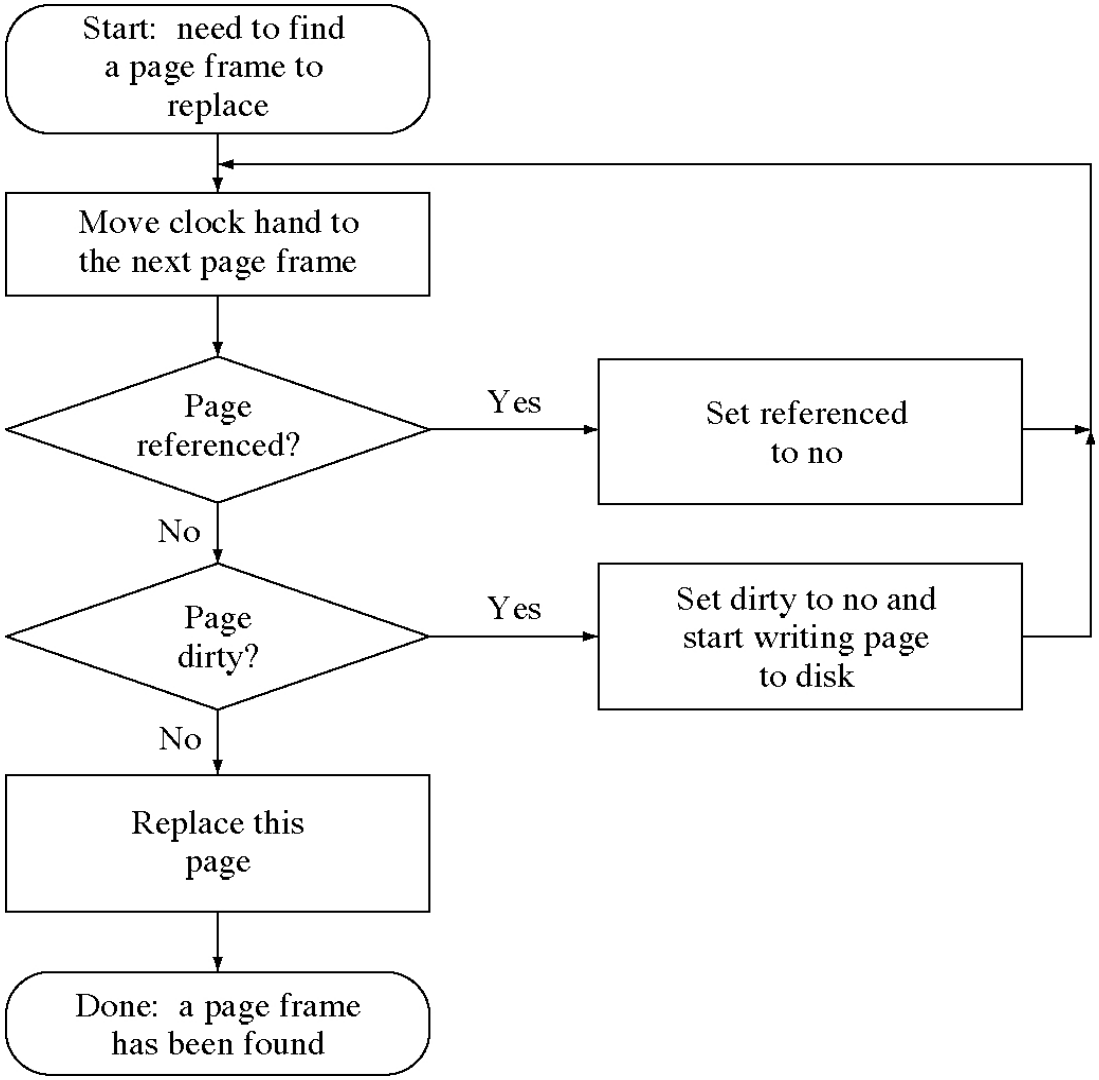
# Modified bit

- Most paging hardware also has a *modified bit* in the page table entry
  - which is set when the page is written to
- This is also called the “dirty bit”
  - pages that have been changed are referred to as “dirty”
  - these pages must be written out to disk because the disk version is out of date
    - this is called “cleaning” the page

# Second chance algorithm

- When you need to replace a page
  - look at the page frame at the clock hand
  - if (referencedBit=0 && modifiedBit=0) then replace the page
  - else if(referencedBit=0 && modifiedBit=1) then set modifiedBit to 0 and move on
  - else set referencedBit to 0 and move on
- Dirty pages get a second chance because they are more expensive to replace

# Clock algorithm flow chart



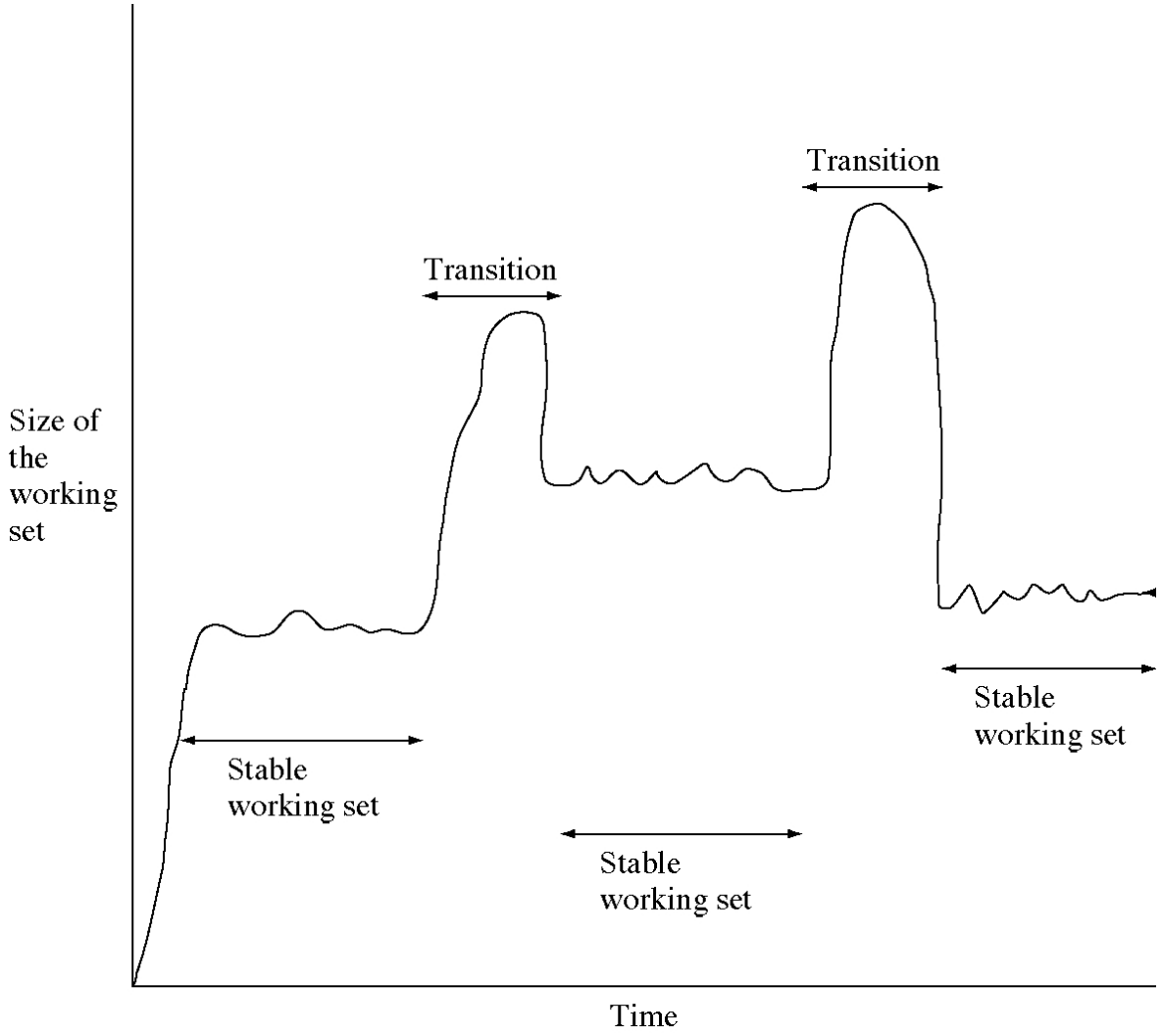
# Working set

- A page in memory is said to be *resident*
- The *working set* of a process is the set of pages it needs to be resident in order to have an acceptable low paging rate.
- Operationally:
  - Each page reference is a unit of time
  - The page reference sequence is: $r_1, r_2, \ldots, r_T$ where T is the current time
  - $W(T,\theta) = \{p \mid p = r_t \text{ where } T-\theta < t < T \}$ is the working set

# Program phases

- Processes tend to have stable working sets for a period of time called a phase
- Then they change phases to a different working set
- Between phases the working set is unstable
  - and we get lots of page faults

# Working set phases



# Working set algorithm

- Algorithm:
  - Keep track of the working set of each running process
  - Only run a process if its entire working set fits in memory
- Too hard to implement so this algorithm is only approximated

# WSClock algorithm

- A clock algorithm that approximates the working set algorithm
- Efficient
- A good approximation of working set

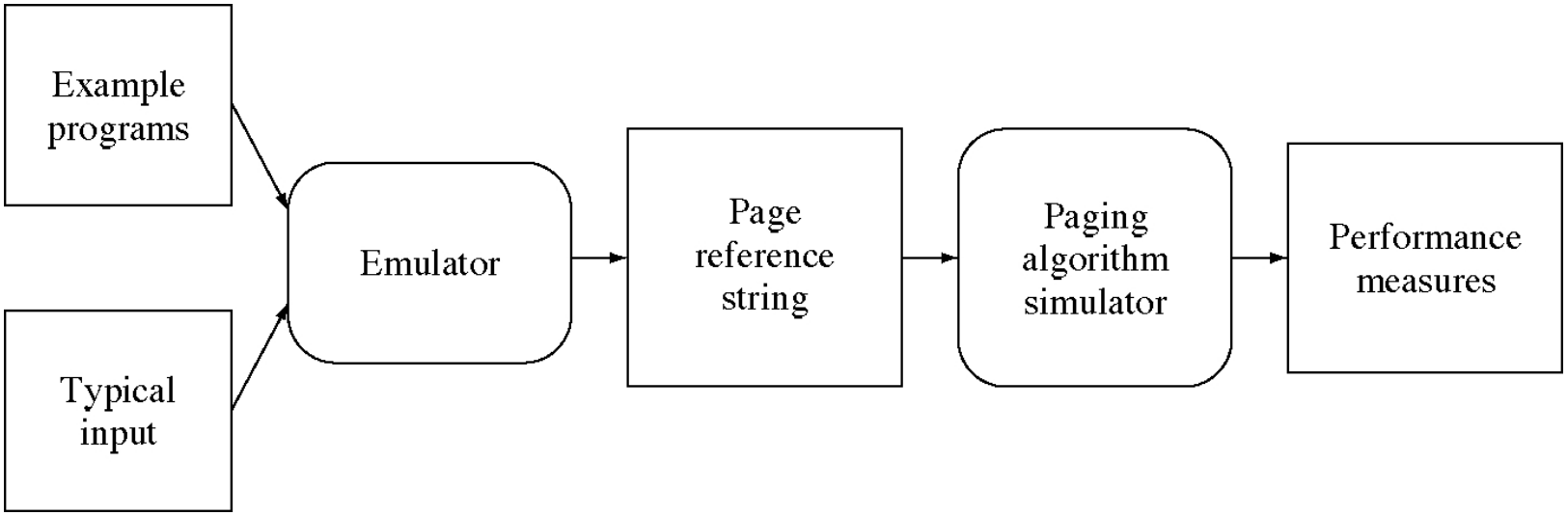
# Design technique: Working sets

- The working set idea comes up in many areas, it tells you how bit your cache needs to be to work effectively
- A user may need two or more windows for a task (e.g. comparing two documents)
  - unless both windows are visible the task may be much harder to do
- Multiple-desktop window managers are based on the idea of a working set of windows for a specific task

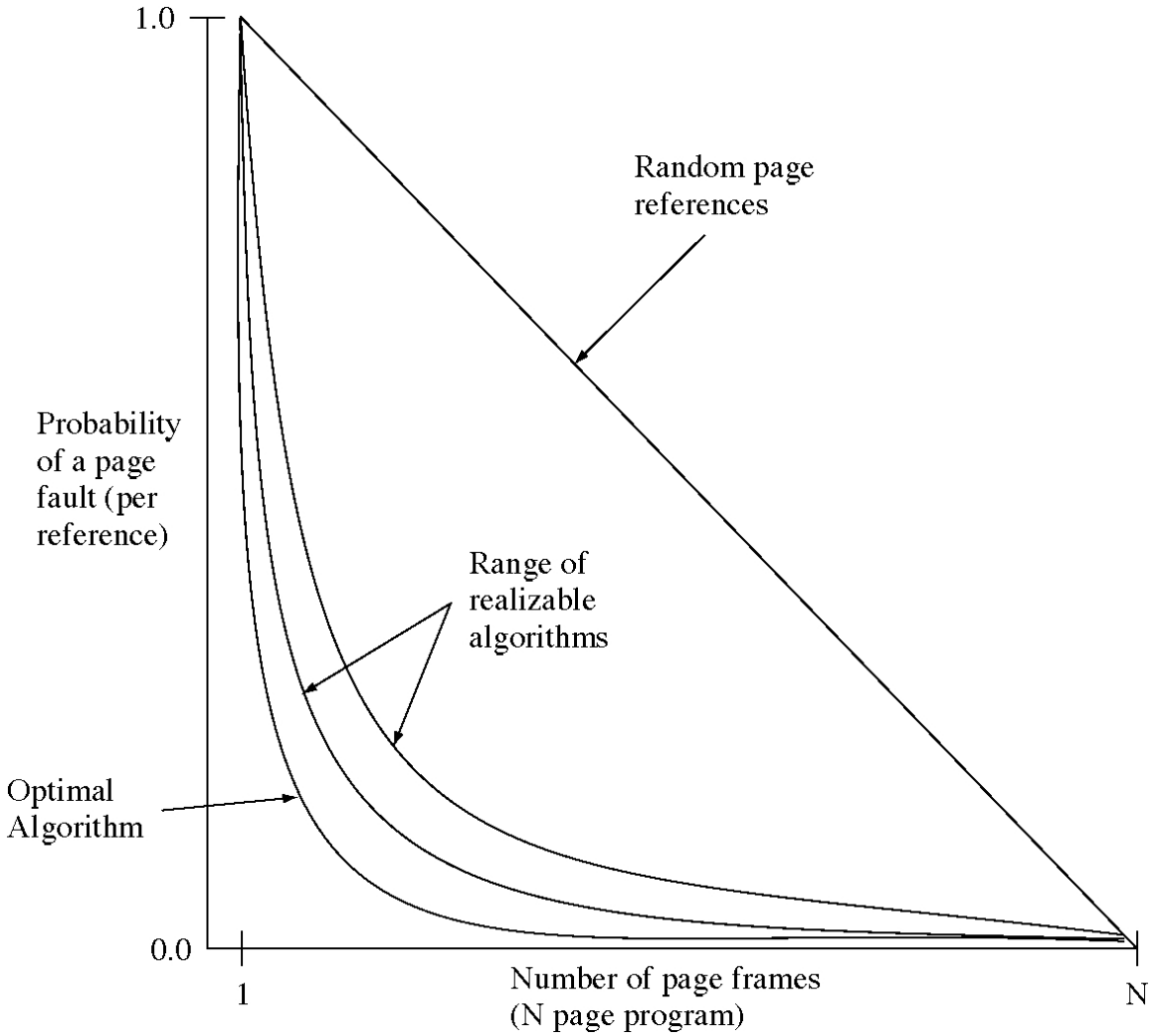
# Evaluating paging algorithms

- Mathematical modeling
  - powerful where it works
  - but most real algorithms cannot be analyzed
- Measurement
  - implement it on a real system and measure it
  - extremely expensive
- Simulation
  - reasonably efficient
  - effective

# Simulation of paging algorithms



# Performance of paging algorithms



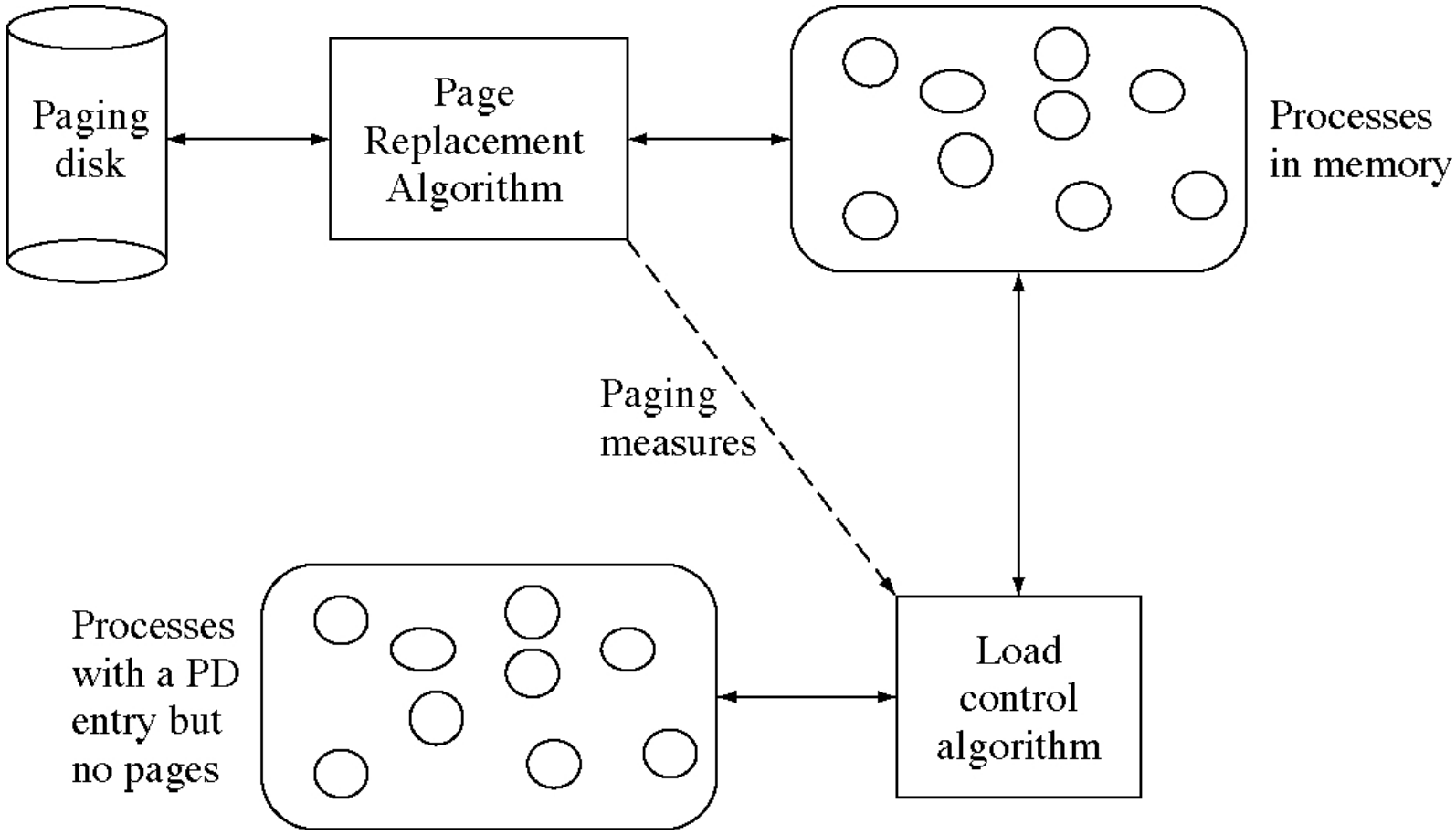
# Thrashing

- VM allows more processes in memory, so one is more likely to be ready to run
- If CPU usage is low, it is logical to bring more processes into memory
- But, low CPU use may to due to too many pages faults because there are too many processes competing for memory
- Bringing in processes makes it worse, and leads to *thrashing*

# Load control

- Load control: deciding how many processes should be competing for page frames
  - too many leads to thrashing
  - too few means that memory is underused
- Load control determines which processes are running at a point in time
  - the others have no page frames and cannot run
- CPU load is a bad load control measure
- Page fault rate is a good load control measure

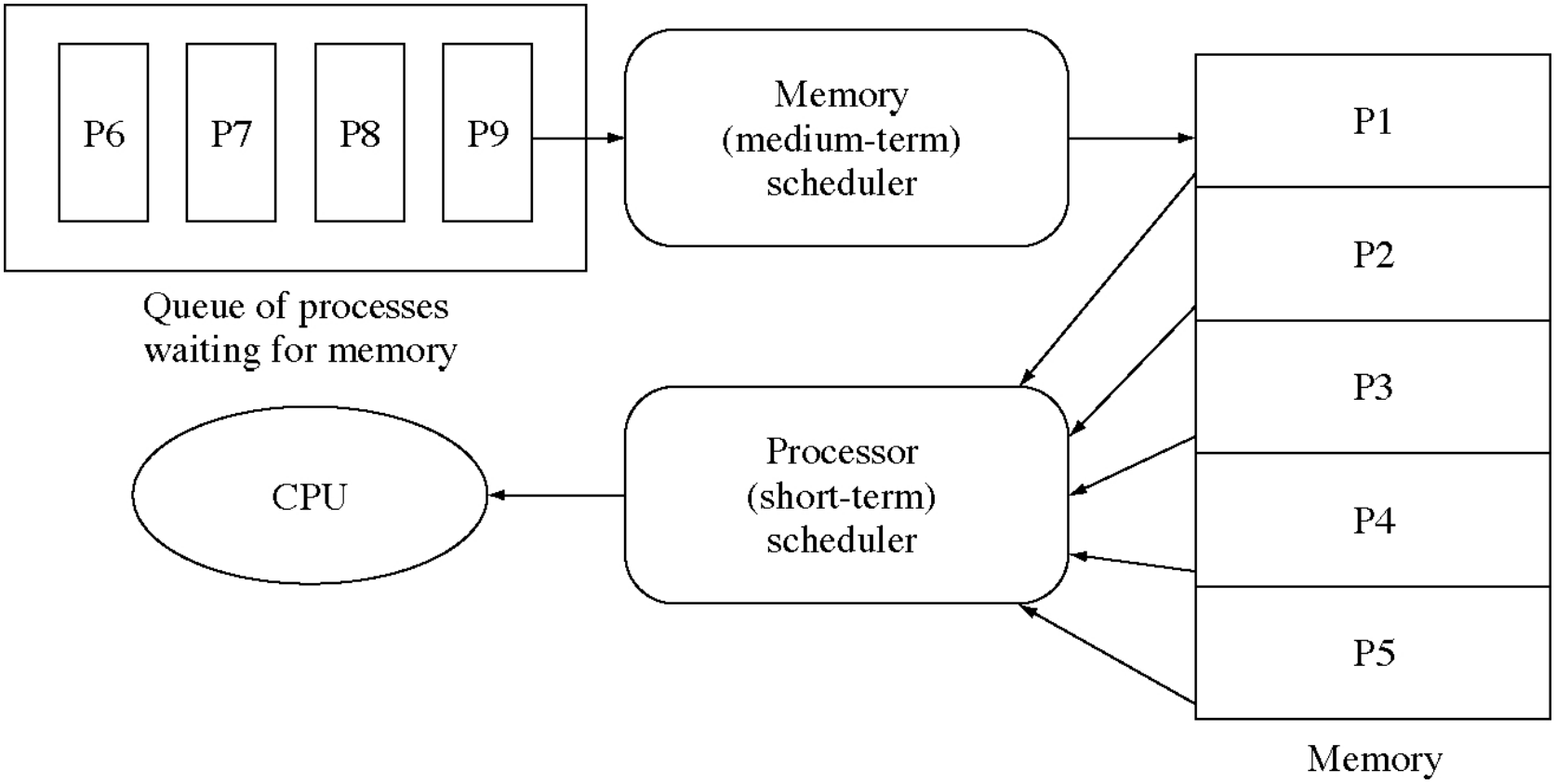
# Load control and page replacement



# Swapping

- Swapping originally meant to write one process out to disk and read another process into memory
  - swapping was used in early time-sharing systems
- Now “swapping” a process out means to not schedule it and let the page replacement algorithm (slowly) take all its page frames
  - it is not all written out at once

# Two levels of scheduling



# Load control algorithms

- A load control algorithm measures memory load and swaps processes in and out depending on the current load
- Load control measures
  - rotational speed of the clock hand
  - average time spent in the standby list
  - page fault rate

# Page fault frequency load control

- L = mean time between page faults
- S = mean time to service a page fault
- Try to keep L = S
  - if L < S, then swap a process out
  - if L > S, then swap a process in
- If L = S, then the paging system can just keep up with the page faults

# Predictive load control

- Page fault frequency reacts after the page fault rate is already too high or too low
  - it is a form of *reactive load control*
- It would be better to predict when the page fault rate is going to be too high and prevent it from happening
  - for example we know that a newly loading processes will cause a lot of page faults
  - so, only allow one loading process at a time (this is called the LT/RT algorithm)

# Demand paging

- So far we have seen *demand paging*: bring a page in when it is requested
- We could do predictive paging, known as *prepaging*
  - but how do we know when a page will be needed in the near future?
  - Generally we don't and that is why prepaging is not commonly done.

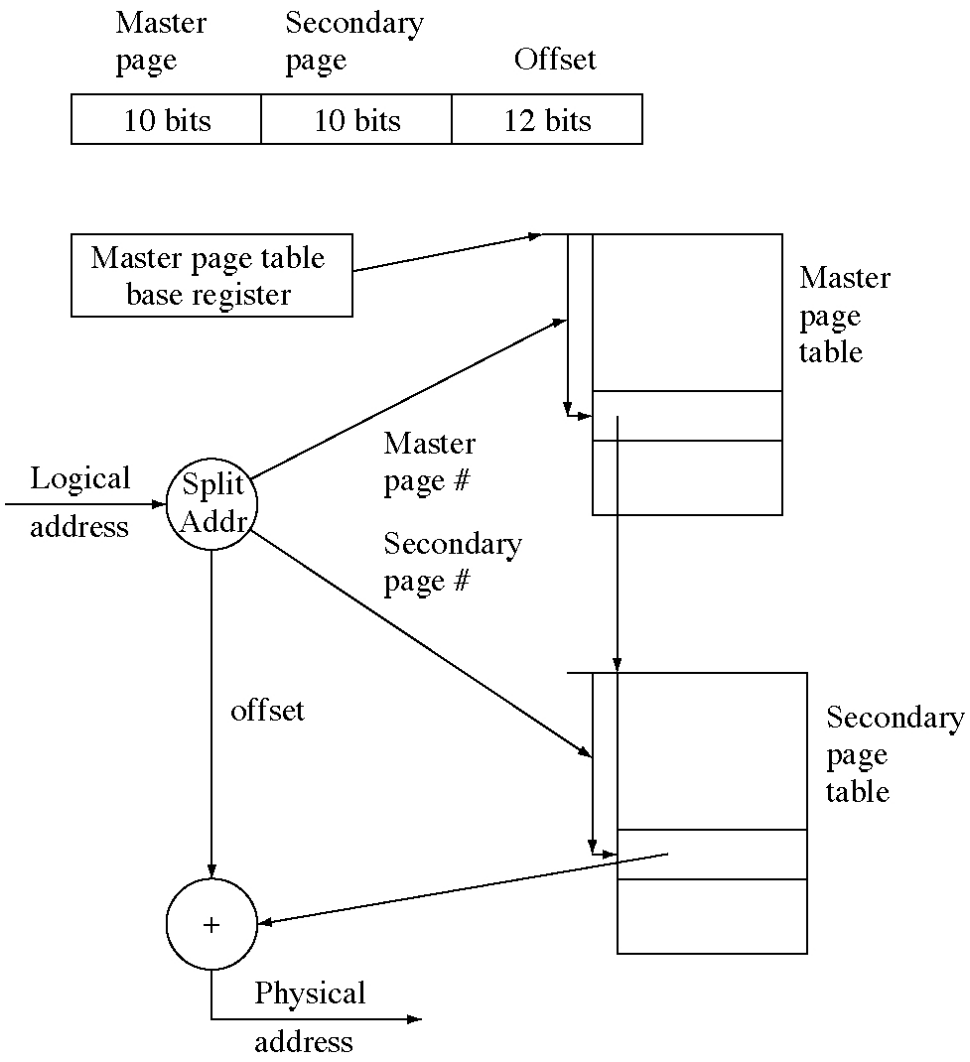
# Large address spaces

- Large address spaces need large page tables
- Large page tables take a lot of memory
  - 32 bit addresses, 4K pages => 1 M pages
  - 1 M pages => 4 Mbytes of page tables
- But most of these page tables are rarely used because of locality

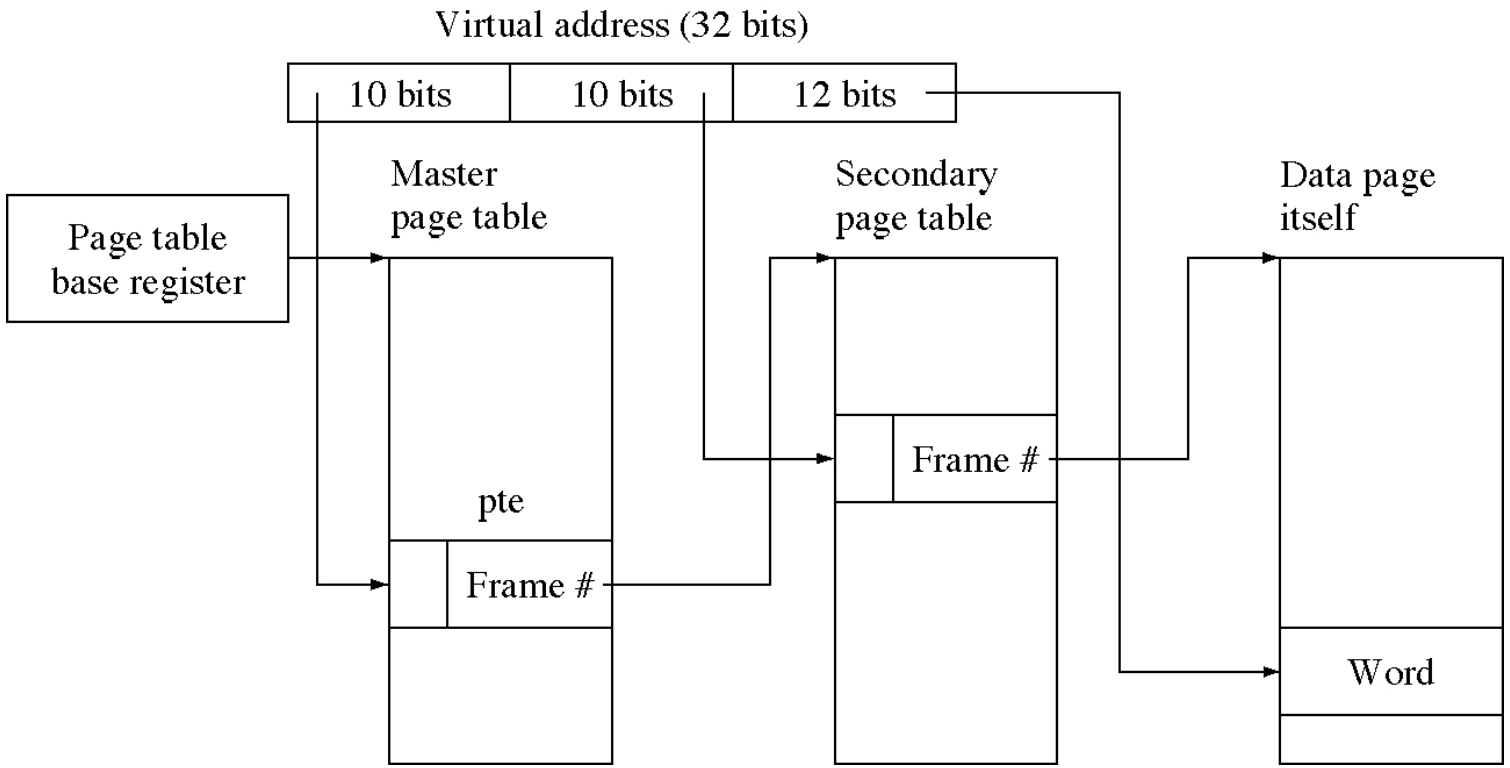
# Two-level paging

- Solution: reuse a good idea
  - We paged programs because their memory use of highly localized
  - So let's page the page tables
- Two-level paging: a tree of page tables
  - Master page table is always in memory
  - Secondary page tables can be on disk

# Two-level paging



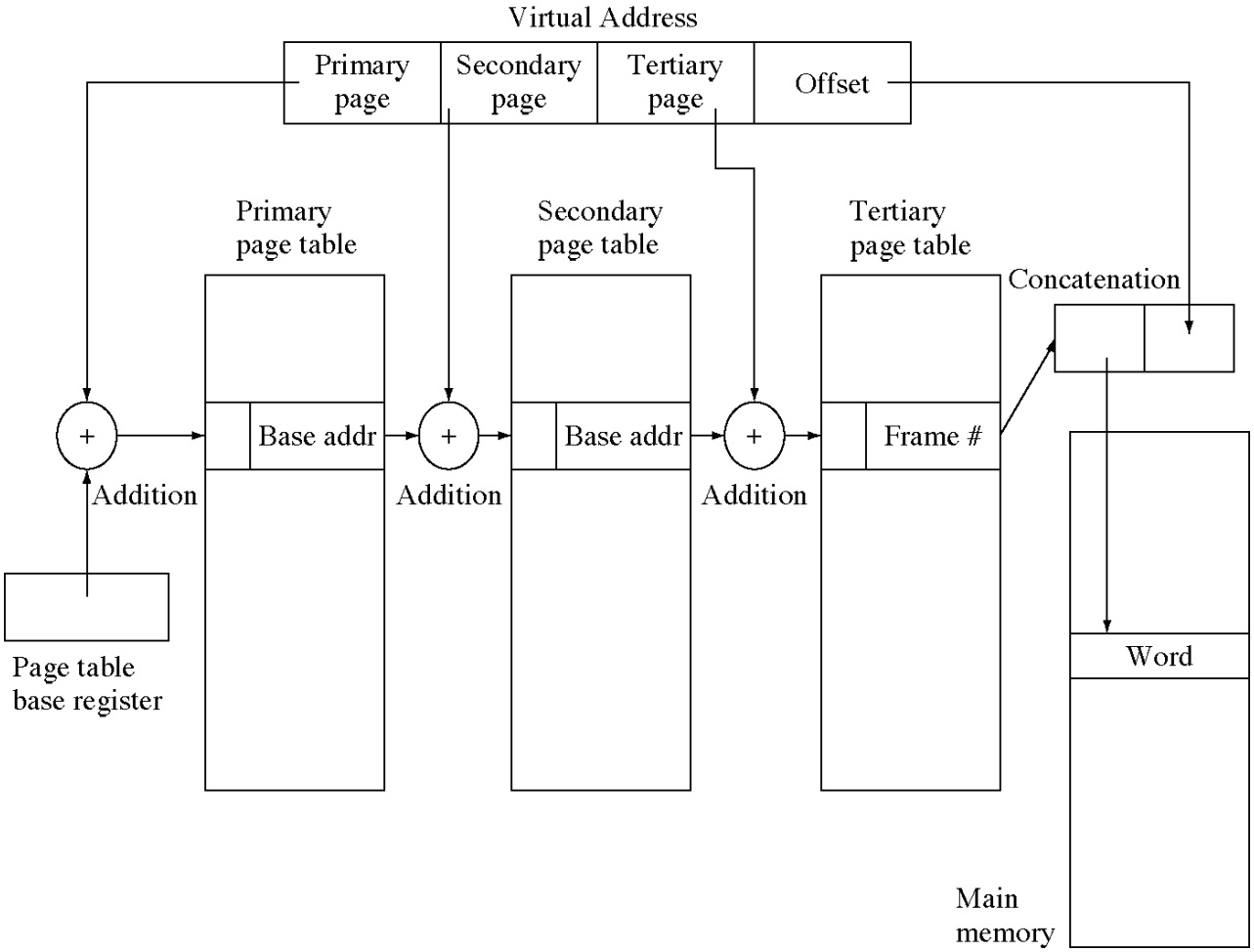
# Another view of two-level paging



# Two-level paging

- Benefits
  - page table need not be in contiguous memory
  - allow page faults on secondary page tables
    - take advantage of locality
  - can easily have “holes” in the address space
    - this allows better protection from overflows of arrays, etc
- Problems
  - two memory accesses to get to a PTE
  - not enough for *really* large address spaces
    - three-level paging can help here

# Three-level paging



# Software page table lookups

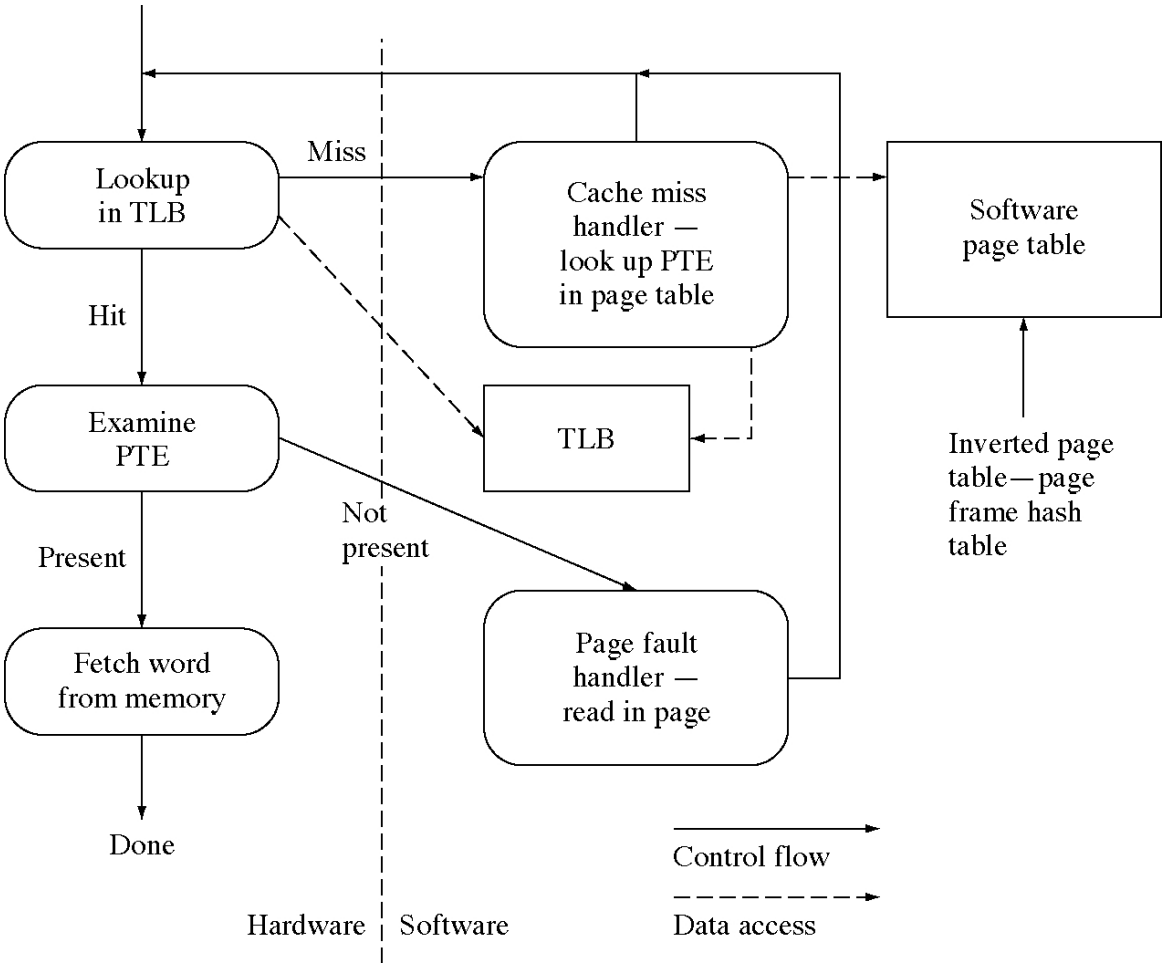
- If TLB misses are uncommon, we can afford to handle them in software
  - and it can structure the page tables any way it wants
- Time to handle various paging events

| *Event* | 1-level | 2-level | 3-level | software |
|---|---|---|---|---|
| *TLB hit* | 1 | 1 | 1 | 1 |
| *TLB miss* | 2 | 3 | 4 | 10-50 |
| *Page fault* | 100K | 100K | 100K | 100K |

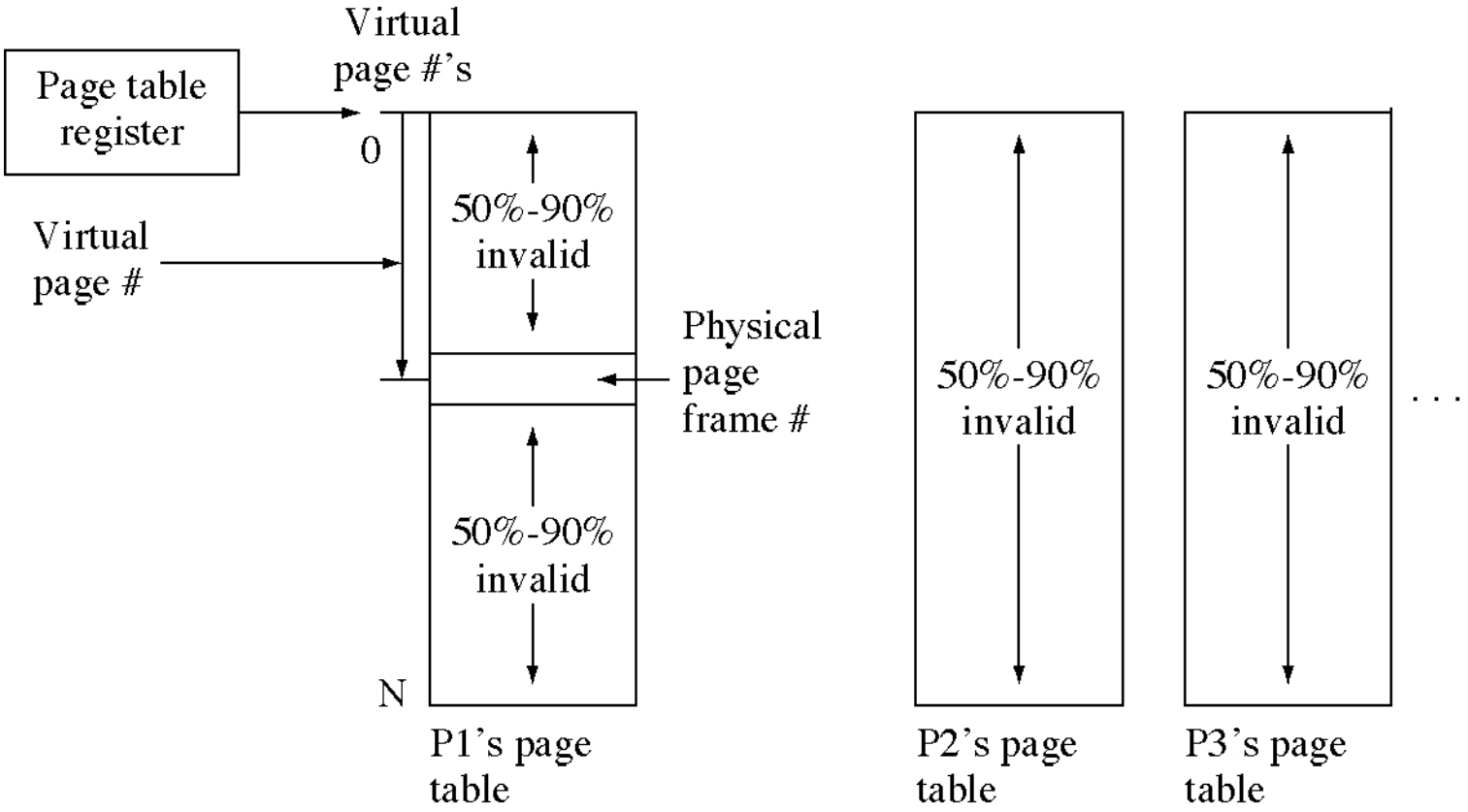
# Inverted page tables

- An common data structure for software-managed page tables
- Keep a table of pages *frames* not pages
  - Problem: we want to look up pages by page number, not page frame number
  - Solution: use clever data structures (e.g. a hash table)

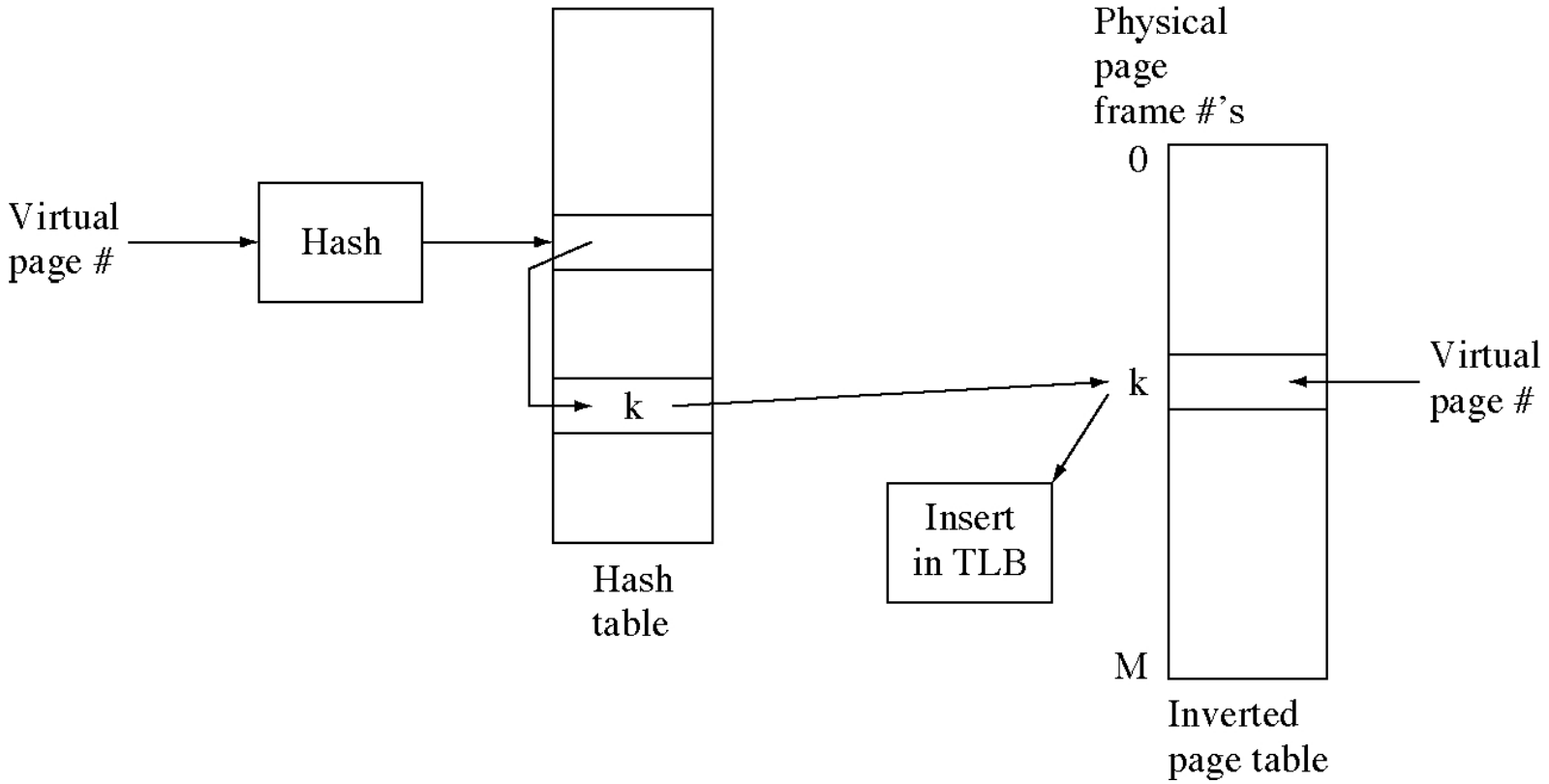
# Software page table lookup



# Normal page tables



# Inverted page tables



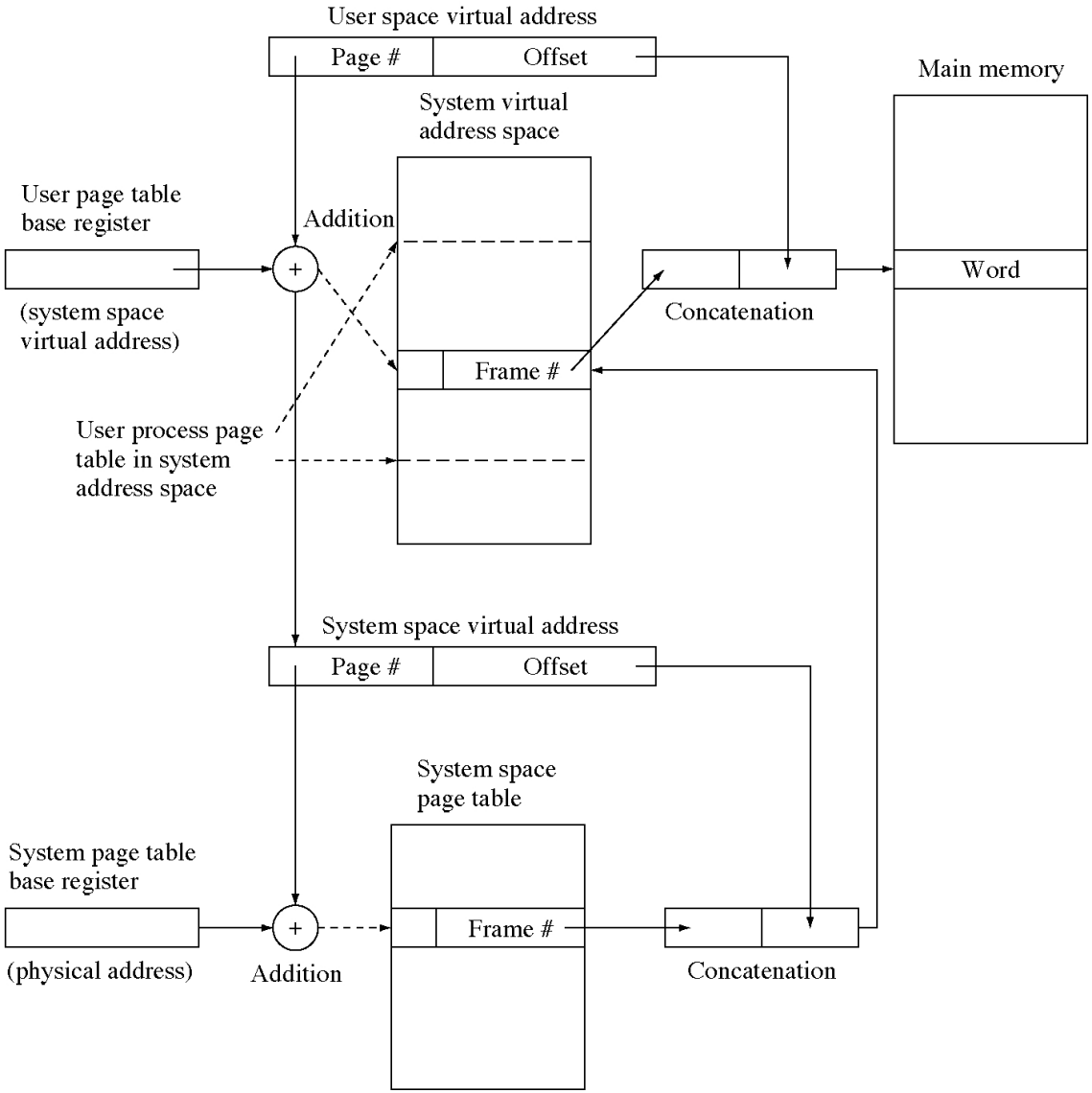
# Design technique: Changing with technology

- Early on people tried software paging
  - but it was way too slow
  - so they tried using hardware register
- They they changed to page table in memory
  - and TLBs to make it fast enough
  - but TLBs with high hit rates meant that software paging became efficient
  - so we let the OS handle the paging after a TLB miss

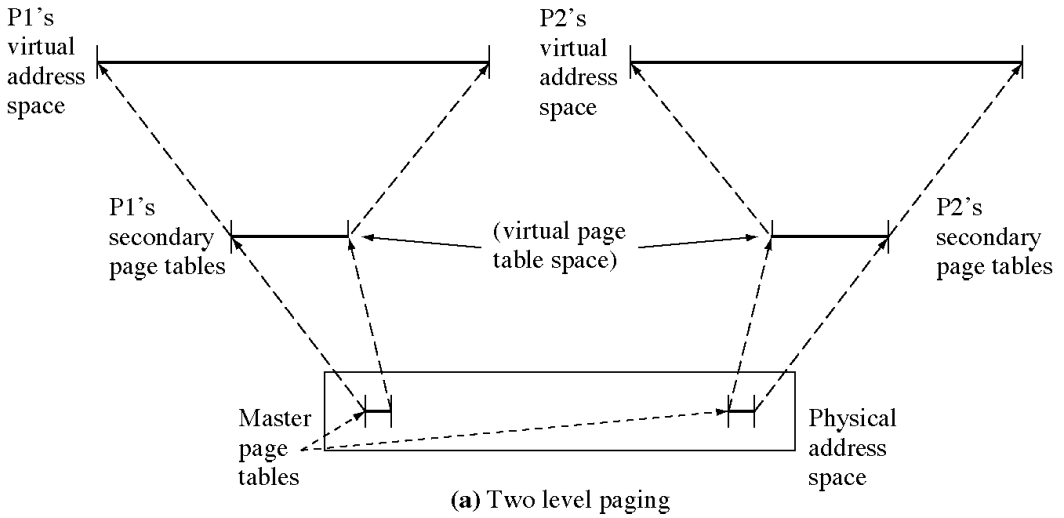
# Recursive address spaces

- Another way to “use a good idea twice”
- The OS runs in the system virtual address space
- The user page tables are in the system virtual address space

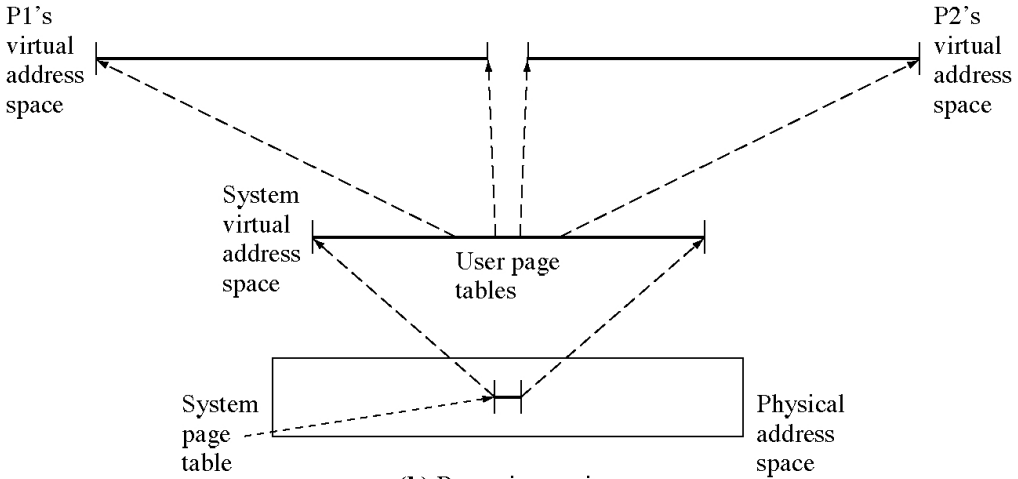
# Two levels of virtual memory



# Two ways to do two-level paging




**(a)** Two level paging




**(b)** Recursive paging

# Page the OS?

- Yes, we can page OS code and data too
  - but some code must always be in memory
  - like the paging code and the dispatching code
- Locking pages in memory
  - prevent them from being paged out
  - for vital part of the OS
  - for pages involved in an I/O operation

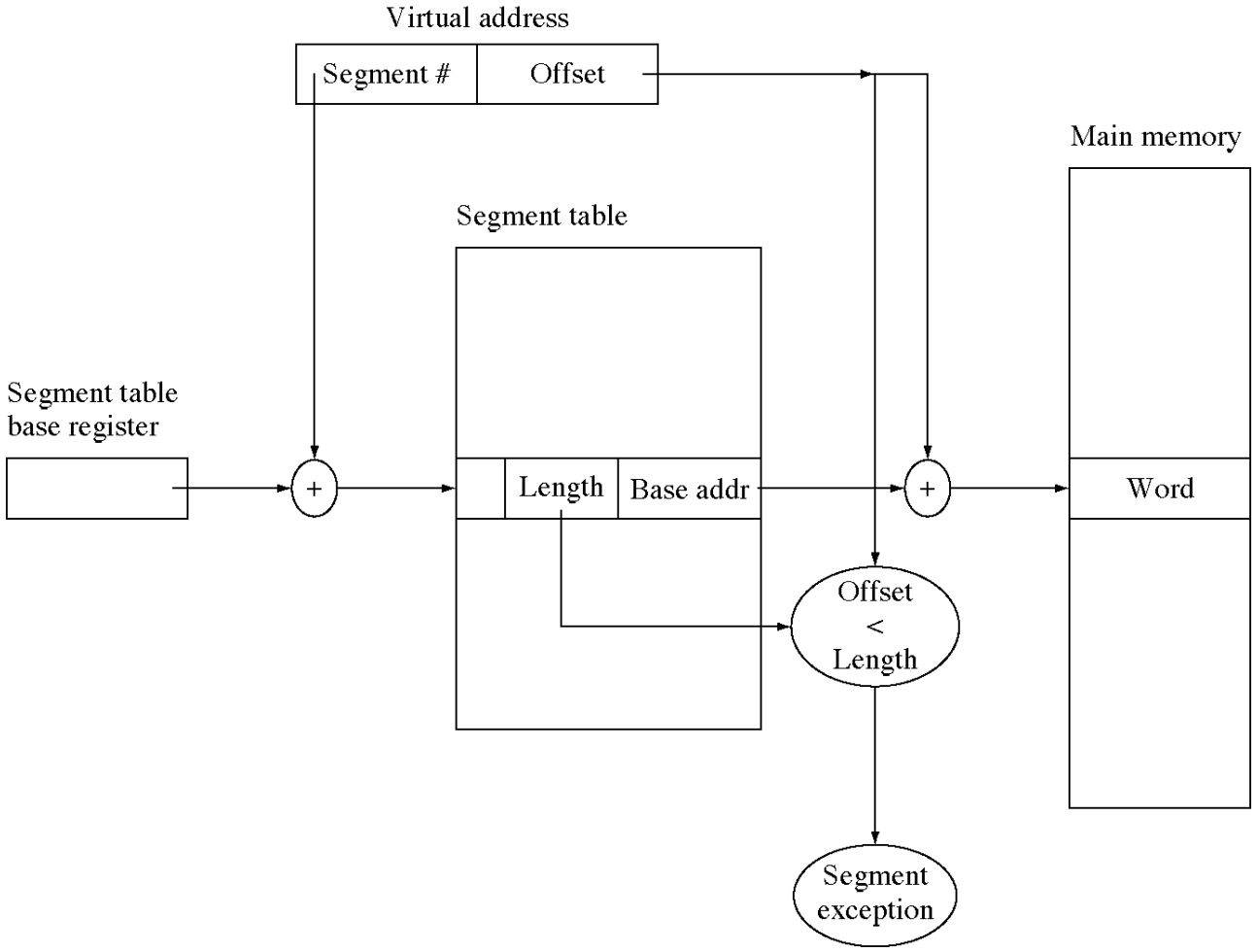
# Page size factors

- Large pages
  - load nearly as fast as small pages
  - load with fewer page faults
  - mean smaller page tables
- Small pages
  - have less internal fragmentation
  - bring less useful code and data into memory
- Page clustering
  - loads several pages together (in a cluster)
  - allows fast loading even with small pages

# Segmentation

- A segment is a variable-sized section of memory that is used for one purpose
  - one procedure, one array, one structure, etc.
- Segmentation moves code/data between disk and memory in variable-sized segments not fixed-size pages
  - otherwise it is just like paging
- The address space is two-dimensional <segment,offset>

# Segmentation



# History of segmentation

- Segmentation has been tried on several machine, most notably Burroughs machines
  - but it was never too successful, the advantage of fixed size units was too great
- Segmentation only existed now in the form of large segments that are themselves paged
  - a variation on two-level paging

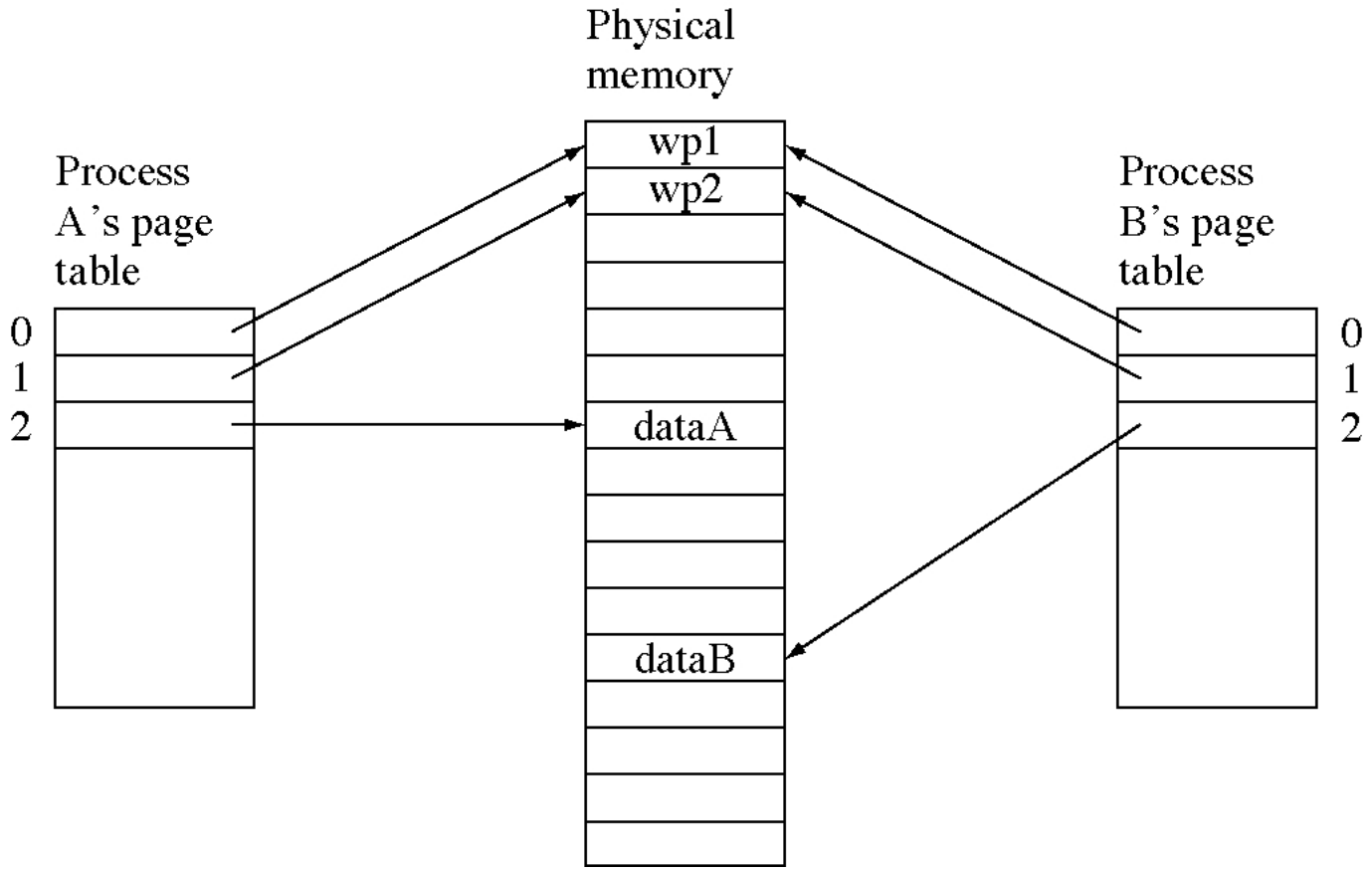
# Design technique: use fixed-size objects

- Paging wins because dynamic storage allocation is too slow
  - pages have fixed size and placement is trivial
- Fixed-sized objects are much easier to manage, no searching, they are all the same
- Often we can convert variable sized objects to a sequence of fixed size objects
  - For example, keep variable-length strings in a linked list of blocks with 32 characters each

# Sharing memory

- Processes have separate address spaces
  - but sometime they want to share memory
    - it is a fast way to communicate between processes, it is a form of IPC
    - it is a way to share common data or code
- We can map the same page into two address spaces (that is, map it in two page tables)
  - but they might have different addresses in the two address spaces

# Two processes sharing code



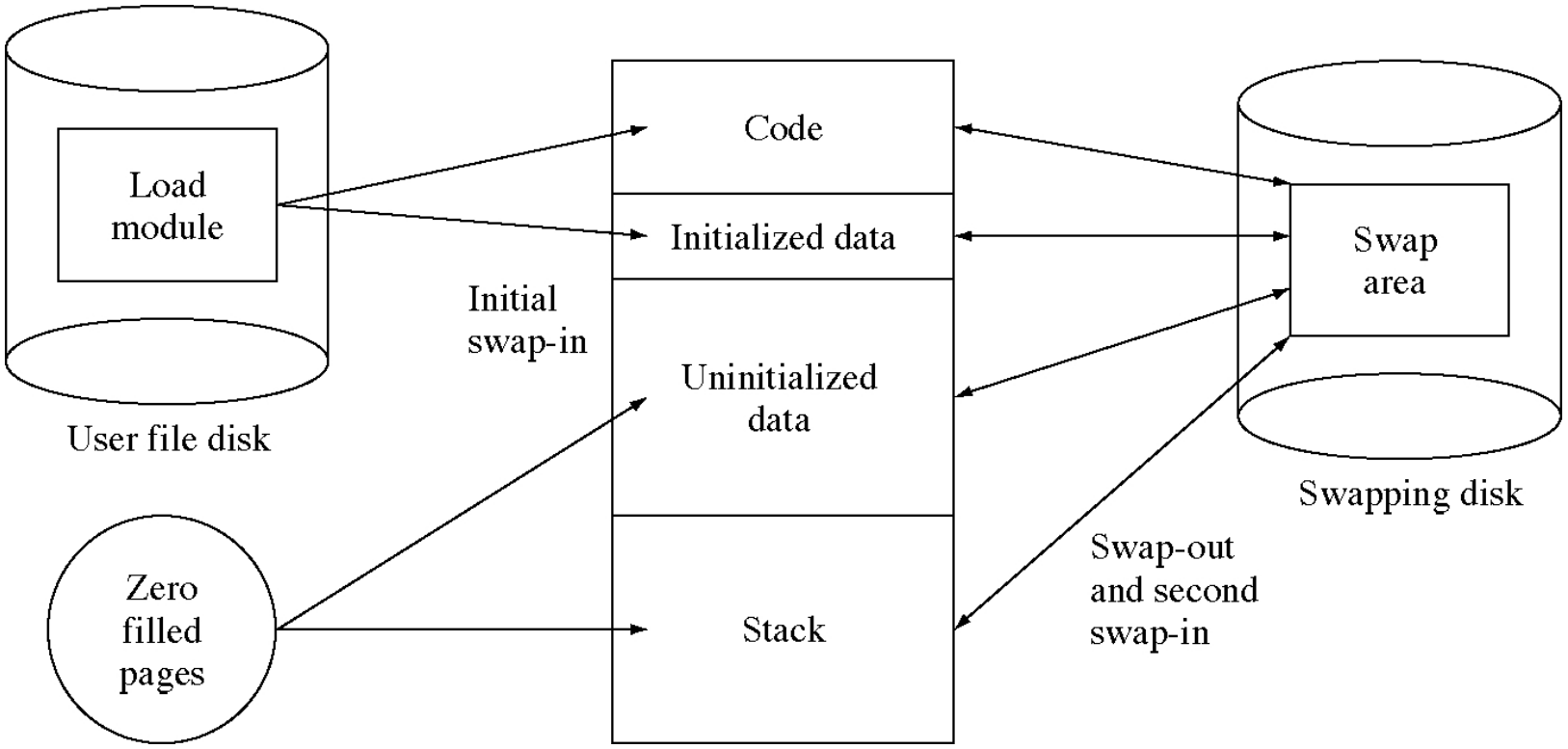
# Swap area variations

- Usually the swap area is a dedicated disk partition
- It can also be a regular file in the file system
  - but this is less efficient
- We can also assume swap space dynamically, just before we have to page out for the first time
  - this is called *virtual swap space*

# Page initialization

- We discussed creating a process image on disk before the process starts
  - but we can avoid this overhead
- Code and initialized data pages can come initially from the load module
- Uninitialized data and stack pages can be initialized as zero-filled page frames.

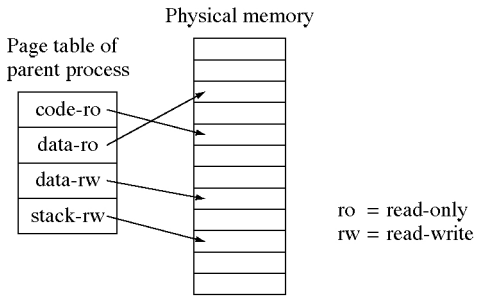
# Initialization of process pages



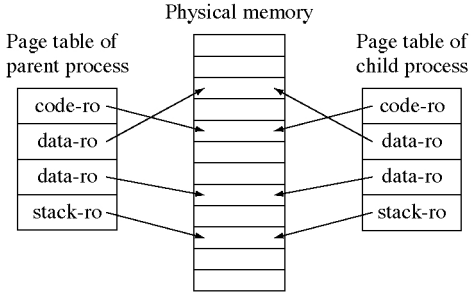
# Copy-on-write sharing

- It is expensive to copy memory
  - and unnecessary if the memory is read-only
  - read-only pages can be mapped into two page tables
- Copy-on-write: a lazy copy
  - copy by copying page table entries
    - all marked read-only
  - Only copy a page when a write occurs on it

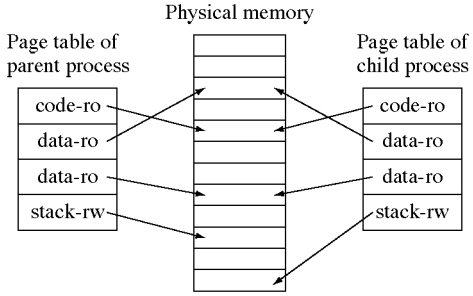
# Implementing copy-on-write




**(a)** Before the fork

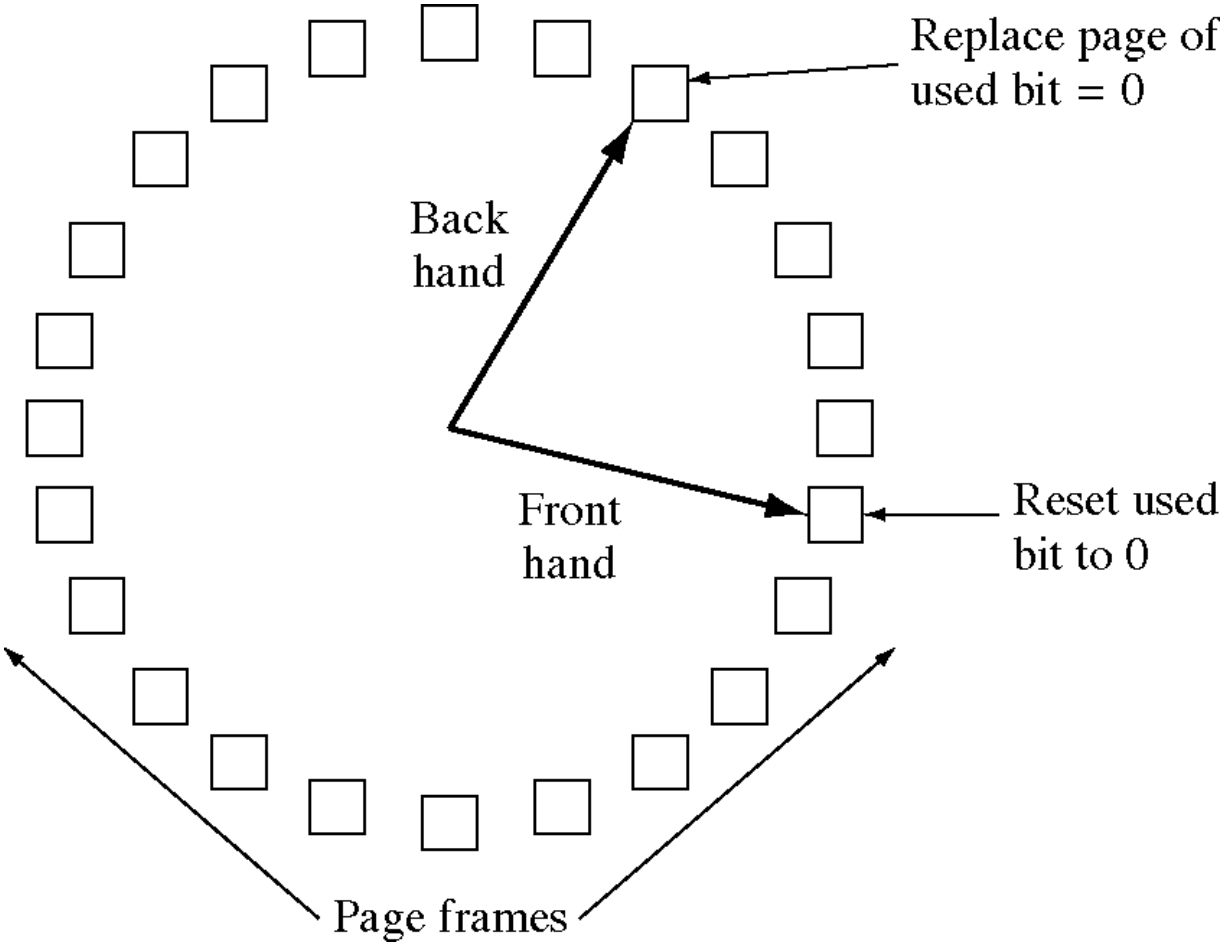



**(b)** After the fork




**(c)** After the child writes the stack

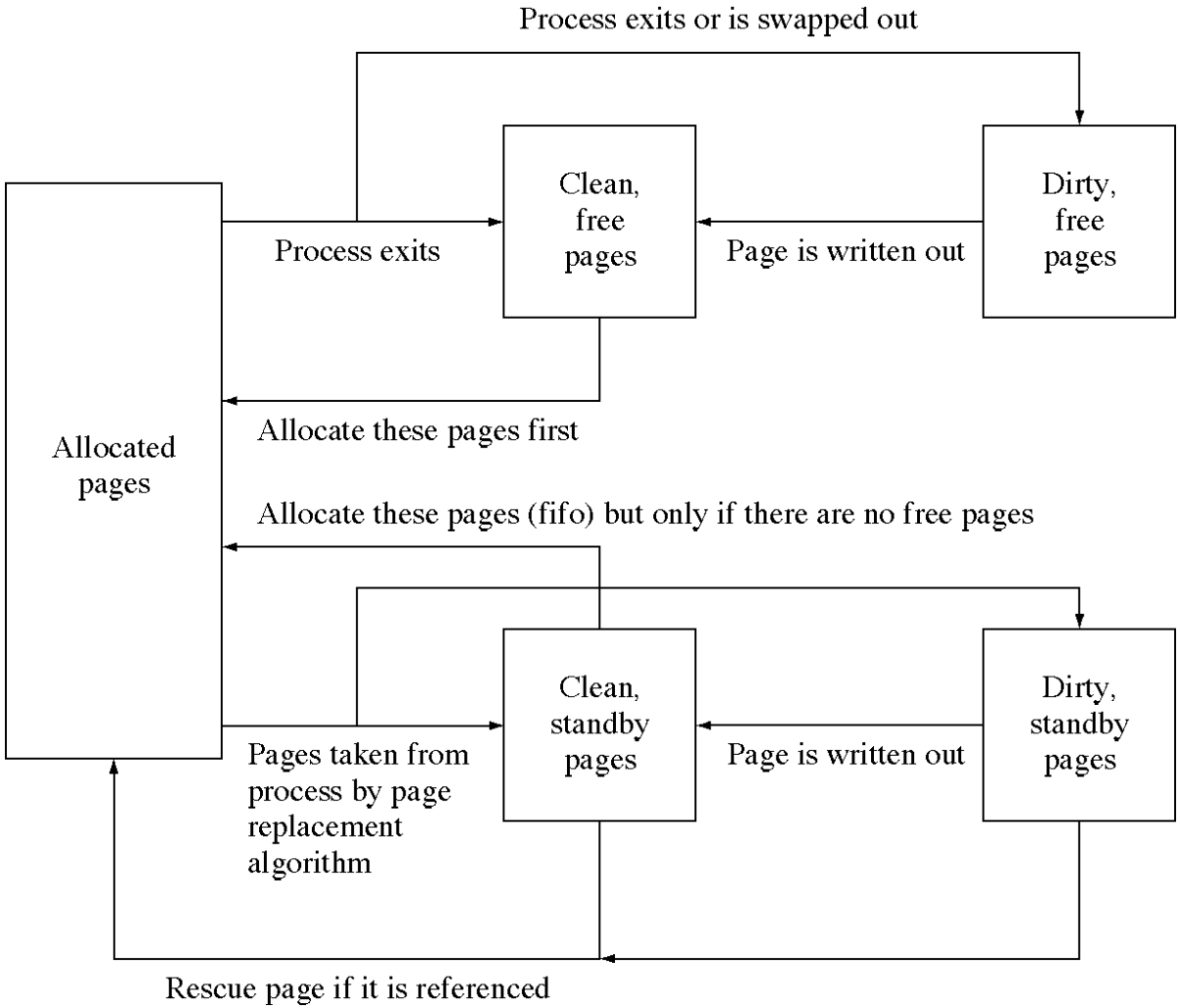
# Two-handed clock



# Standby page lists

- OSs normally keep a pool of free pages
- We can rescue pages for this pool if they are referenced again
  - then the page pool is called a *standby list*
- Regularly referenced pages are always rescued, only not-recently-referenced pages are reused
  - this approximates LRU very well

# Standby page list



# Sparse address spaces

- Non-contiguous logical address spaces can be useful
  - an array with unmapped space around it will catch errors that run past the end of the array
  - we can detect when an array or stack overflows and automatically expand it
- This is most useful with two-level paging

# Very large address spaces

- Newer processors use 64 bit integers and addresses
  - although often not all 64 address bits are used
- 64 bits allows a huge virtual address space
- Future OSs may map all processes into a single 64-bit address space
  - this makes sharing very easy
  - some things can be assigned address space permanently