Operating System Concepts

Lecture 28: Frame Allocation Algorithms & Thrashing

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Today's class

- Page replacement algorithms
  - second-chance algorithm and its variants
  - counting-based algorithms
- Frame allocation algorithms
- Thrashing
- Kernel memory allocation

Approximating LRU

- **basic idea:** keep a reference bit for each page
  - on each access to the page, the hardware sets the reference bit to ‘1’
  - set it to ‘0’ at varying times depending on the page replacement algorithm

Approximating LRU

- **basic idea:** keep a reference bit for each page
  - on each access to the page, the hardware sets the reference bit to ‘1’
  - set it to ‘0’ at varying times depending on the page replacement algorithm
- Additional-Reference-Bits algorithm
  - maintain more than 1 bit, say for example 8 bits
  - at regular intervals (e.g., every 100 milliseconds) or on each memory access, right shift the byte (**an aging mechanism**) by one bit, placing a 0 in the high order bit; on a page fault, the lowest numbered page is kicked out
    - it is approximate, since it does not guarantee a total order on the pages; but it is faster, since it requires setting a single bit on each memory access
  - page fault still requires a search through all the pages

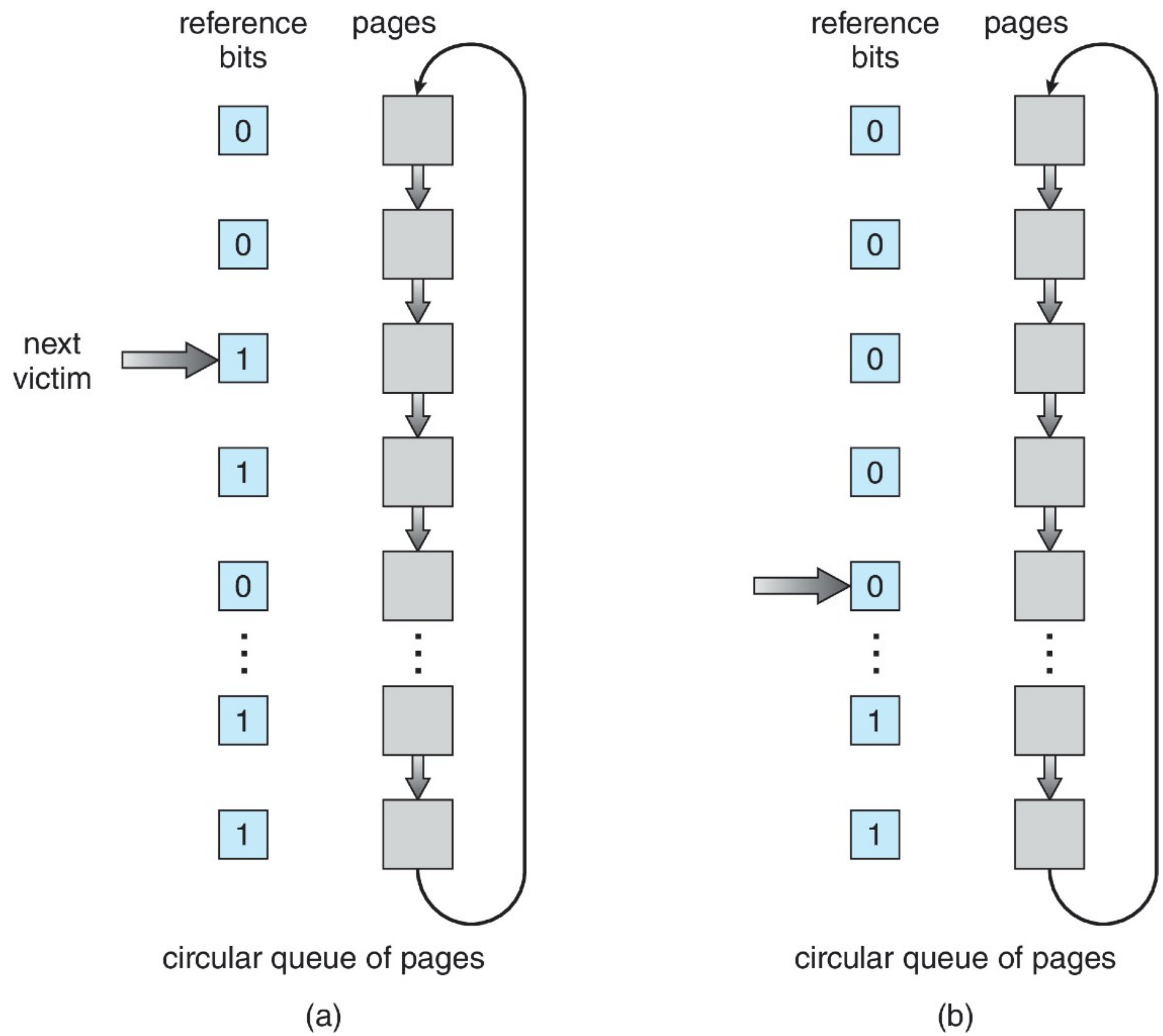
Second-chance page replacement algorithm

- use a single reference bit per page
  - OS keeps all pages in a circular list and a pointer to the next frame that must be considered
  - on a page fault, OS checks the reference bit of the frame that the pointer refers to
    - if the reference bit is ‘0’, replaces the page, sets its reference bit to ‘1’, and advances the pointer to the next frame
    - if the reference bit is ‘1’, clears the reference bit (sets bit to ‘0’), and advances the pointer to the next frame

Second-chance page replacement algorithm

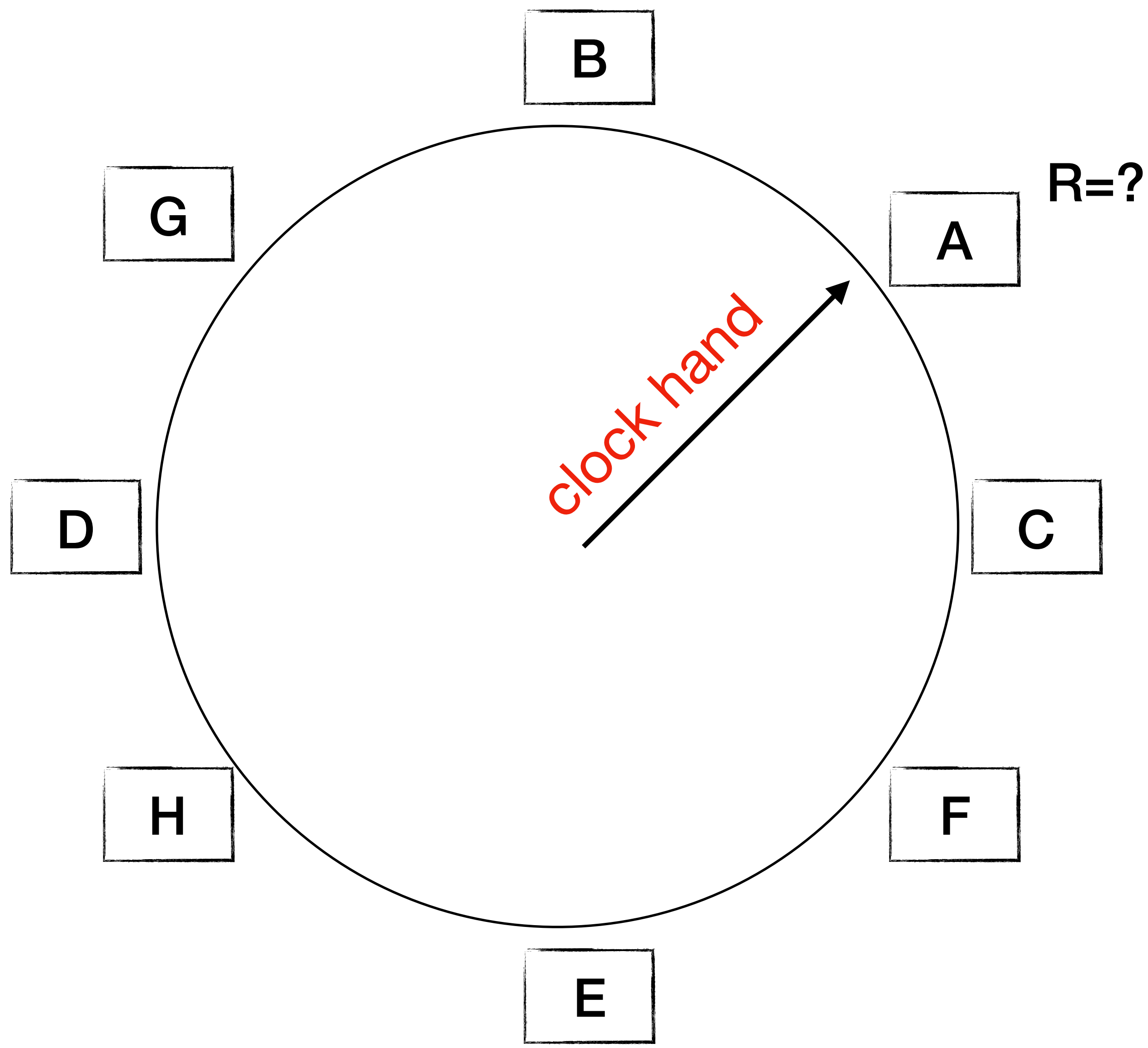
- use a single reference bit per page
  - OS keeps all pages in a circular list and a pointer to the next frame that must be considered
  - on a page fault, OS checks the reference bit of the frame that the pointer refers to
    - if the reference bit is ‘0’, replaces the page, sets its reference bit to ‘1’, and advances the pointer to the next frame
    - if the reference bit is ‘1’, clears the reference bit (sets bit to ‘0’), and advances the pointer to the next frame
- disadvantage
  - it is less accurate than the additional-reference-bits algorithm since the reference bit only indicates if the page was used at all since the last time it was checked by the algorithm
- advantage
  - it is fast as it requires setting a single bit on each memory access (no need for a shift)
  - handling a page fault is also faster as we only search the pages until we find one with a reference bit that is not set

Implementation



if R=0 then evict the page and set reference bit of the new page to 1 and advance the pointer otherwise clear R and advance the pointer

Why is it called the clock algorithm?



hand-spread is the size of the active list

a closer look…

- the clock algorithm is a crude partitioning of pages into two categories: young and old pages

a closer look…

- the clock algorithm is a crude partitioning of pages into two categories: young and old pages

- why not partition pages into more than two categories?

    - because handling a page fault would require finding the youngest page

a closer look…

- the clock algorithm is a crude partitioning of pages into two categories: young and old pages
- why not partition pages into more than two categories?
  - because handling a page fault would require finding the youngest page
- should we replace any young page?
  - of course it is cheaper to replace a page that has not been modified because the OS does not need to write it out

Enhanced second-chance algorithm

- let’s keep a modify bit in addition to the reference bit for each page
  - ‘1’ indicates that the page is modified, hence it is different from the copy residing on disk
  - ‘0’ indicates that the page is the same as the copy residing on disk

Enhanced second-chance algorithm

- let’s keep a modify bit in addition to the reference bit for each page
  - ‘1’ indicates that the page is modified, hence it is different from the copy residing on disk
  - ‘0’ indicates that the page is the same as the copy residing on disk
- with these two bits (r, m) we have four possible classes
  - (0,0) neither recently used nor modified
    - best candidate for replacement!
  - (0,1) not recently used, but modified
    - it might not be needed anymore, but not as good to replace since the OS must write out the page
  - (1,0) recently used and unmodified
    - probably will be used again soon, but the OS doesn’t need to write it out before replacing it
  - (1,1) recently used and modified
    - probably will be used again soon and the OS must write it out before replacing it

Page replacement in the enhanced second-chance algorithm

- one the first pass if there is a page with (0,0) then replace it
  - if there is a page with (0,1) then initiate an I/O to write out the page, lock the page in memory until the I/O completes, clear the modified bit and continue the search
  - for pages with the reference bit set, clear the reference bit
  - if the hand goes completely around once without replacing a page, there was no (0,0) page

Page replacement in the enhanced second-chance algorithm

- one the first pass if there is a page with (0,0) then replace it
  - if there is a page with (0,1) then initiate an I/O to write out the page, lock the page in memory until the I/O completes, clear the modified bit and continue the search
  - for pages with the reference bit set, clear the reference bit
  - if the hand goes completely around once without replacing a page, there was no (0,0) page
- on the second pass, a page that was originally (0,1) or (1,0) might have been changed to (0,0); if so replace that page
  - if the page is being written out, wait for the I/O to complete and then remove the page
  - a (0,1) page is treated the same way as a (0,1) page was treated on the first pass

Page replacement in the enhanced second-chance algorithm

- one the first pass if there is a page with (0,0) then replace it
  - if there is a page with (0,1) then initiate an I/O to write out the page, lock the page in memory until the I/O completes, clear the modified bit and continue the search
  - for pages with the reference bit set, clear the reference bit
  - if the hand goes completely around once without replacing a page, there was no (0,0) page
- on the second pass, a page that was originally (0,1) or (1,0) might have been changed to (0,0); if so replace that page
  - if the page is being written out, wait for the I/O to complete and then remove the page
  - a (0,1) page is treated the same way as a (0,1) page was treated on the first pass
- by the third pass, all the pages will be in the (0,0) class

Page replacement in the enhanced second-chance algorithm

- one the first pass if there is a page with (0,0) then replace it
  - if there is a page with (0,1) then initiate an I/O to write out the page, lock the page in memory until the I/O completes, clear the modified bit and continue the search
  - for pages with the reference bit set, clear the reference bit
  - if the hand goes completely around once without replacing a page, there was no (0,0) page
- on the second pass, a page that was originally (0,1) or (1,0) might have been changed to (0,0); if so replace that page
  - if the page is being written out, wait for the I/O to complete and then remove the page
  - a (0,1) page is treated the same way as a (0,1) page was treated on the first pass
- by the third pass, all the pages will be in the (0,0) class

OS goes around at most 3 times searching for the (0,0) class

Counting-based algorithms

- count the number of references that have been made to each page and use this number to identify a victim page

Counting-based algorithms

- count the number of references that have been made to each page and use this number to identify a victim page
  - **Least frequently used (LFU)**: throw out the page with the smallest count
    - what if a page is used heavily during the first phase of a process but never used again

Counting-based algorithms

- count the number of references that have been made to each page and use this number to identify a victim page
  - **Least frequently used (LFU)**: throw out the page with the smallest count
    - what if a page is used heavily during the first phase of a process but never used again
  - **Most frequently used (MFU)**: throw out the page with the largest count

Counting-based algorithms

- count the number of references that have been made to each page and use this number to identify a victim page
    - **Least frequently used (LFU)**: throw out the page with the smallest count
        - what if a page is used heavily during the first phase of a process but never used again
    - **Most frequently used (MFU)**: throw out the page with the largest count
- counting based algorithms are expensive and do not approximate OPT (or LRU algorithm)

Page buffering

- handling a page fault takes some time especially when the victim page must be written out

Page buffering

- handling a page fault takes some time especially when the victim page must be written out
- so to make sure that the faulting process can restart as soon as possible, OS keeps a pool of free frames
  - when a page fault occurs, a victim page is chosen as before and disk I/O is initiated to write out this page
  - instead of waiting for I/O completion, the desired page is quickly read into a free frame from the pool
  - the victim page is evicted when convenient and the corresponding frame is added to the pool

Page buffering

- handling a page fault takes some time especially when the victim page must be written out

- so to make sure that the faulting process can restart as soon as possible, OS keeps a pool of free frames

  - when a page fault occurs, a victim page is chosen as before and disk I/O is initiated to write out this page

  - instead of waiting for I/O completion, the desired page is quickly read into a free frame from the pool

  - the victim page is evicted when convenient and the corresponding frame is added to the pool

- how to size this pool? depends on the page fault service time

Frame allocation

- each process needs a minimum number of frames to hold all pages that any single instruction can reference
  - otherwise a page fault occurs in the middle of executing an instruction causing it to restart

Frame allocation

- each process needs a minimum number of frames to hold all pages that any single instruction can reference
  - otherwise a page fault occurs in the middle of executing an instruction causing it to restart

- allocation algorithms
  - **equal allocation**: give each process an equal share of available frames in the system for example, if there are 128 frames available in a system, the kernel takes 35 frames, the pool contains 3 frames, and there are 5 active processes, so each process is given 18 frames

Frame allocation

- each process needs a minimum number of frames to hold all pages that any single instruction can reference
  - otherwise a page fault occurs in the middle of executing an instruction causing it to restart

- allocation algorithms
  - **equal allocation**: give each process an equal share of available frames in the system for example, if there are 128 frames available in a system, the kernel takes 35 frames, the pool contains 3 frames, and there are 5 active processes, so each process is given 18 frames
  - **proportional allocation**: allocate available frames to each process according to its size (or its priority) so larger processes get more page frames

$$s_i = \text{size of process } p_i$$

$$S = \sum s_i$$

$$m = \text{total number of frames}$$

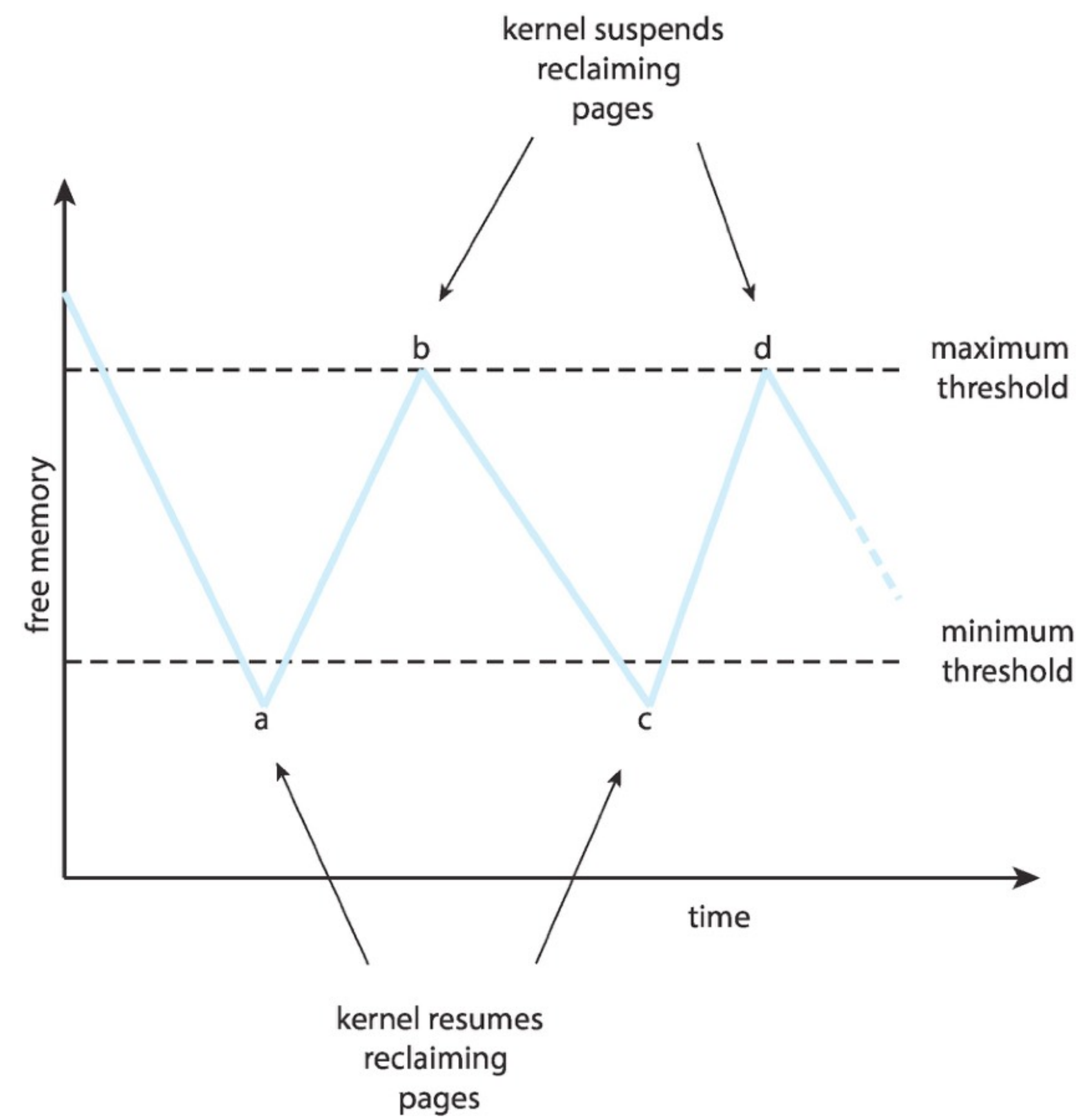
$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

Frame allocation policies

- global replacement: each process selects a replacement frame from the set of all frames in the system
  - one process can take a frame from another (more flexible)
  - but a process can't control its own page fault rate as it depends on the paging behaviour of other processes in the system
    - for this reason process execution time can vary greatly
  - **thrashing** might become more likely
- local replacement: each process selects from only its own set of allocated frames
  - more consistent per-process performance, but memory can be under utilized

Implementing global page-replacement policy

- all memory requests are satisfied from the free-frame list
- page replacement is triggered when the list falls below a certain threshold to ensure that there is always sufficient free frames available



Thrashing

- **Definition**: a process is thrashing when it spends more time paging than executing

- Why? the memory is over-committed and pages are continuously tossed out (by a global page-replacement algorithm) while they are still in use
  - effective memory access time approaches disk access time since many memory references cause page fault
  - results in very noticeable loss of performance (throughput)

Thrashing in a multiprogrammed environment

OS monitors CPU utilization

Thrashing in a multiprogrammed environment

OS monitors CPU utilization

$\downarrow$

CPU utilization is too low

Thrashing in a multiprogrammed environment

OS monitors CPU utilization

↓

CPU utilization is too low → OS introduces a new process to the system

the degree of multiprogramming

Thrashing in a multiprogrammed environment

OS monitors CPU utilization

CPU utilization is too low

the degree of multiprogramming

OS introduces a new process
to the system

an active process enters a new phase
of its execution

it needs more frames
so starts faulting

Thrashing in a multiprogrammed environment

OS monitors CPU utilization

CPU utilization is too low

the degree of multiprogramming

OS introduces a new process
to the system

an active process enters a new phase
of its execution

it needs more frames
so starts faulting

a global page-replacement policy takes
frames away from other processes

Thrashing in a multiprogrammed environment

OS monitors CPU utilization

CPU utilization is too low

the degree of multiprogramming

OS introduces a new process to the system

an active process enters a new phase of its execution

it needs more frames so starts faulting

a global page-replacement policy takes frames away from other processes

these processes needed those frames

they fault as well taking frames from other processes

Thrashing in a multiprogrammed environment

OS monitors CPU utilization

CPU utilization is too low

the degree of multiprogramming

OS introduces a new process to the system

an active process enters a new phase of its execution

it needs more frames so starts faulting

a global page-replacement policy takes frames away from other processes

these processes needed those frames

they fault as well taking frames from other processes

all faulting processes queue up for the paging device

average service time

throughput

Thrashing in a multiprogrammed environment

OS monitors CPU utilization

CPU utilization is too low

the degree of multiprogramming

OS introduces a new process to the system

an active process enters a new phase of its execution

it needs more frames so starts faulting

a global page-replacement policy takes frames away from other processes

these processes needed those frames

they fault as well taking frames from other processes

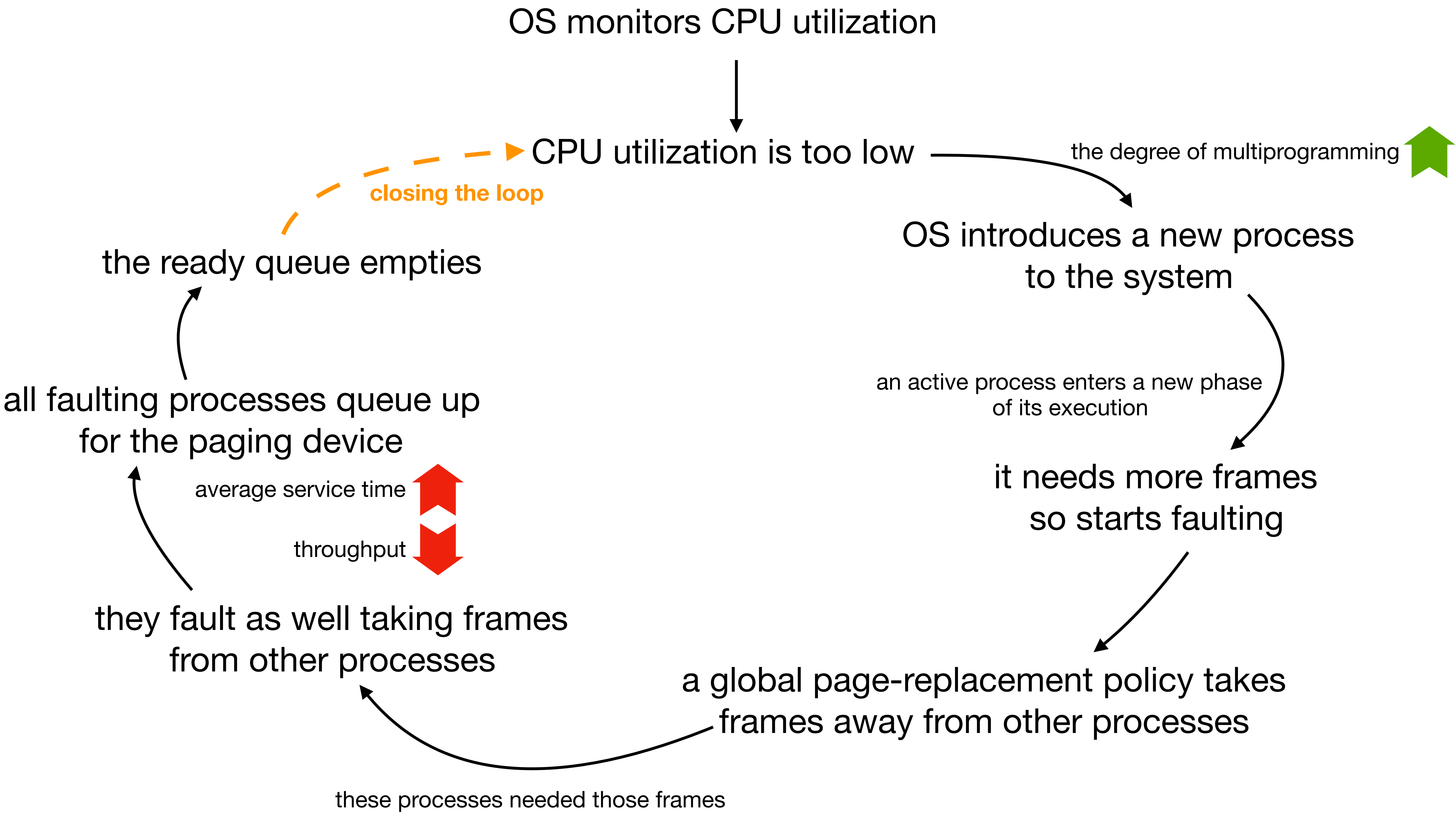
average service time

throughput

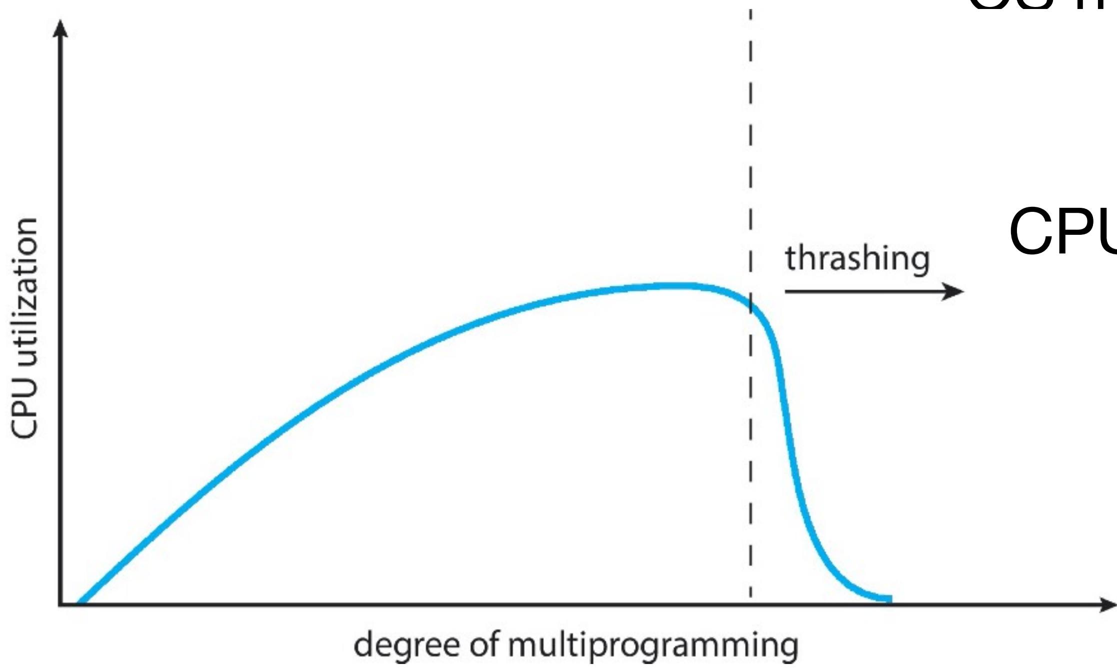
all faulting processes queue up for the paging device

the ready queue empties

Thrashing in a multiprogrammed environment



Thrashing in a multiprogrammed environment



OS monitors CPU utilization

CPU utilization is too low

the degree of multiprogramming

OS introduces a new process to the system

an active process enters a new phase of its execution

it needs more frames so starts faulting

a global page-replacement policy takes frames away from other processes

these processes needed those frames

they fault as well taking frames from other processes

all faulting processes queue up for the paging device

average service time

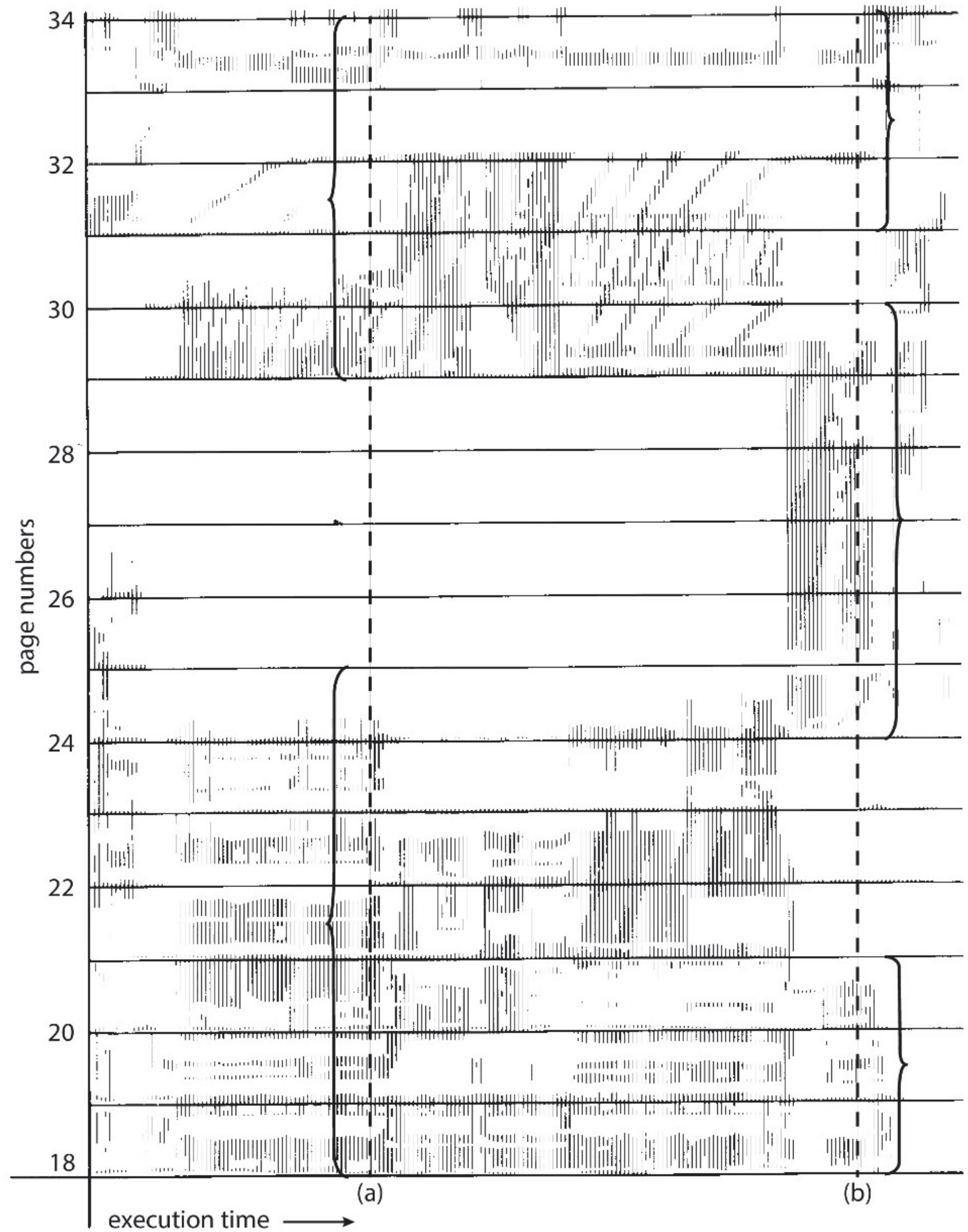
throughput

What can be done in a multiprogrammed environment to limit thrashing?

- per-process replacement
  - each process has its own pool of pages which is large enough (i.e., contains as many page frames as it needs)
  - run only groups of processes that fit in memory and suspend the rest
- how to determine how many pages a process actually needs?
  - use the **working-set model** which approximates the program locality (i.e., set of pages that are actively used together)
  - **definition:** working set is the set of all pages that a process referenced in the past Δ time units
    - hence, a page that is in active use will be in the working set

Process's locality changes over time

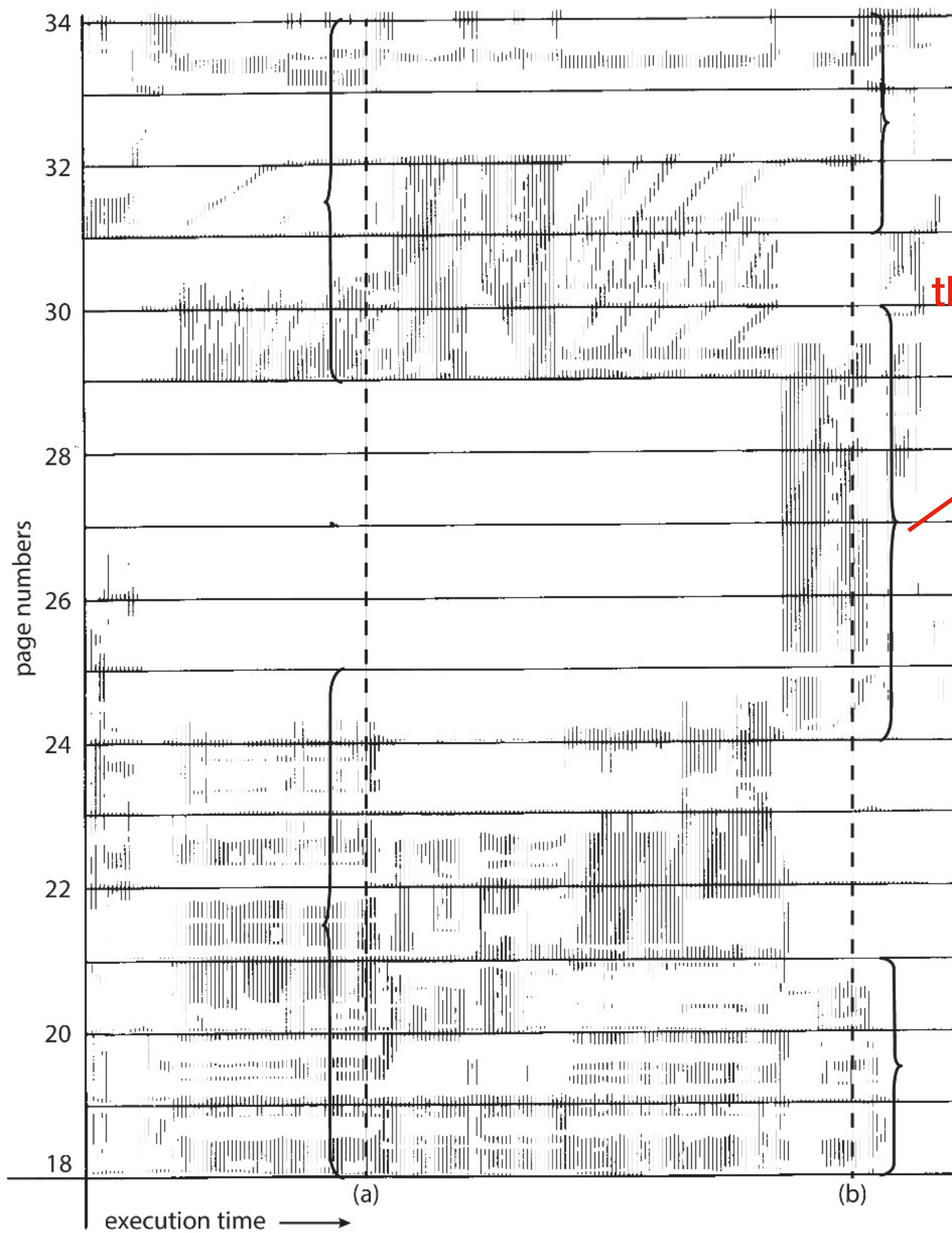
process's localities depends on its program structure and data structure



localities at (a) and (b) overlap

Process’s locality changes over time

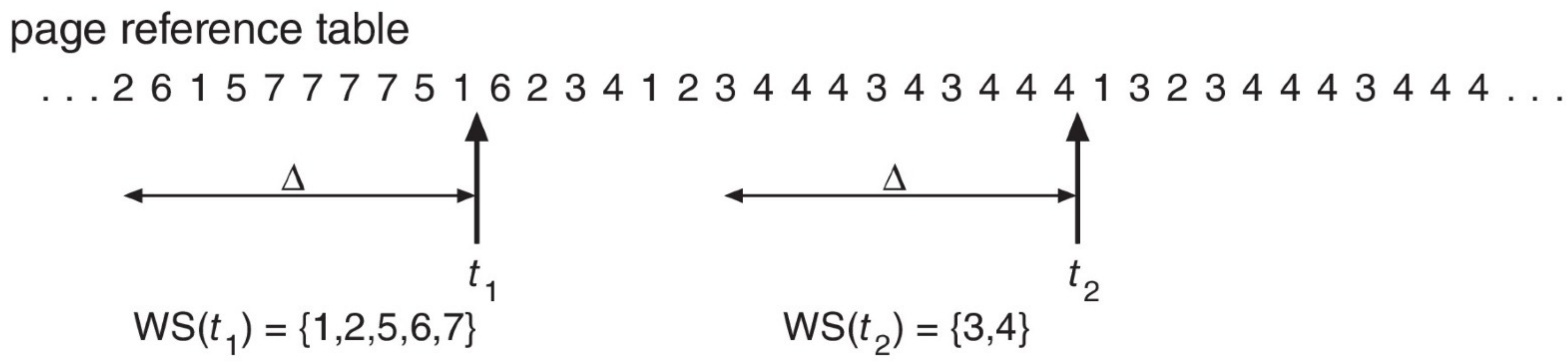
process’s localities depends on its program structure and data structure



they might be pages that belong to a function called before (b)

localities at (a) and (b) overlap

Working set determination



the working set updates dynamically

how to pick Δ?

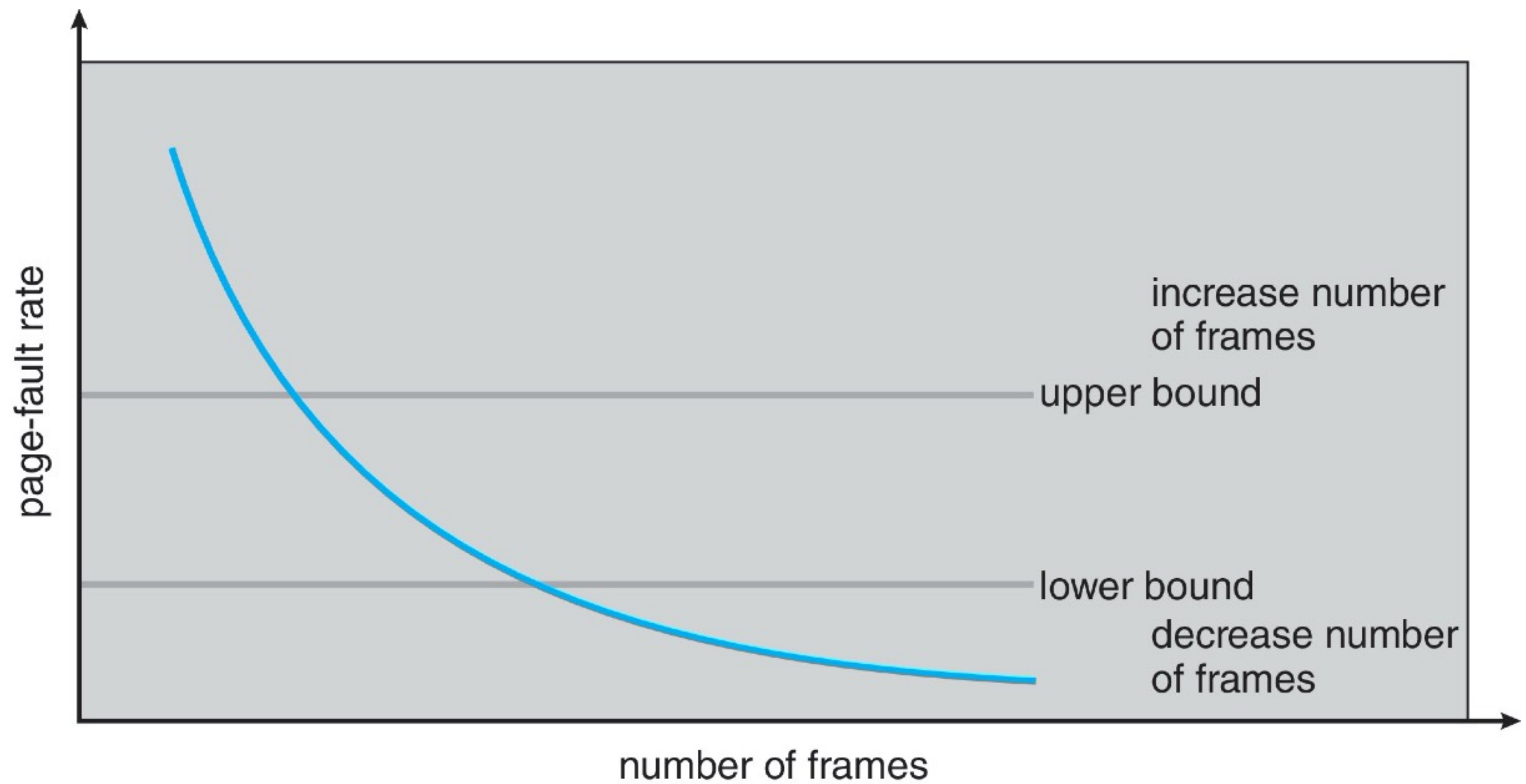
- what if Δ is too small? it does not encompass the entire locality
- what if Δ is too big? it overlaps several localities and may result in thrashing

Per-process replacement

- working sets are expensive to compute!

Per-process replacement

- working sets are expensive to compute!
- an alternative approach is to track page-fault frequency of each process and use it to create a **control signal**
  - give a process more page frames if the page fault frequency > some threshold
    - if no free frame is available we may need to suspend some processes
  - take away some page frames if the page fault frequency < another threshold



Per-process replacement

- working sets are expensive to compute!
- an alternative approach is to track page-fault frequency of each process and use it to create a **control signal**
  - give a process more page frames if the page fault frequency > some threshold
    - if no free frame is available we may need to suspend some processes
  - take away some page frames if the page fault frequency < another threshold
- this approach yields more consistent performance independent of system load

Per-process replacement

- working sets are expensive to compute!
- an alternative approach is to track page-fault frequency of each process and use it to create a **control signal**
  - give a process more page frames if the page fault frequency > some threshold
    - if no free frame is available we may need to suspend some processes
  - take away some page frames if the page fault frequency < another threshold
- this approach yields more consistent performance independent of system load
- **Goal:** the system-wide mean time between page faults should be equal to the time it takes to handle a page fault

Kernel memory allocator

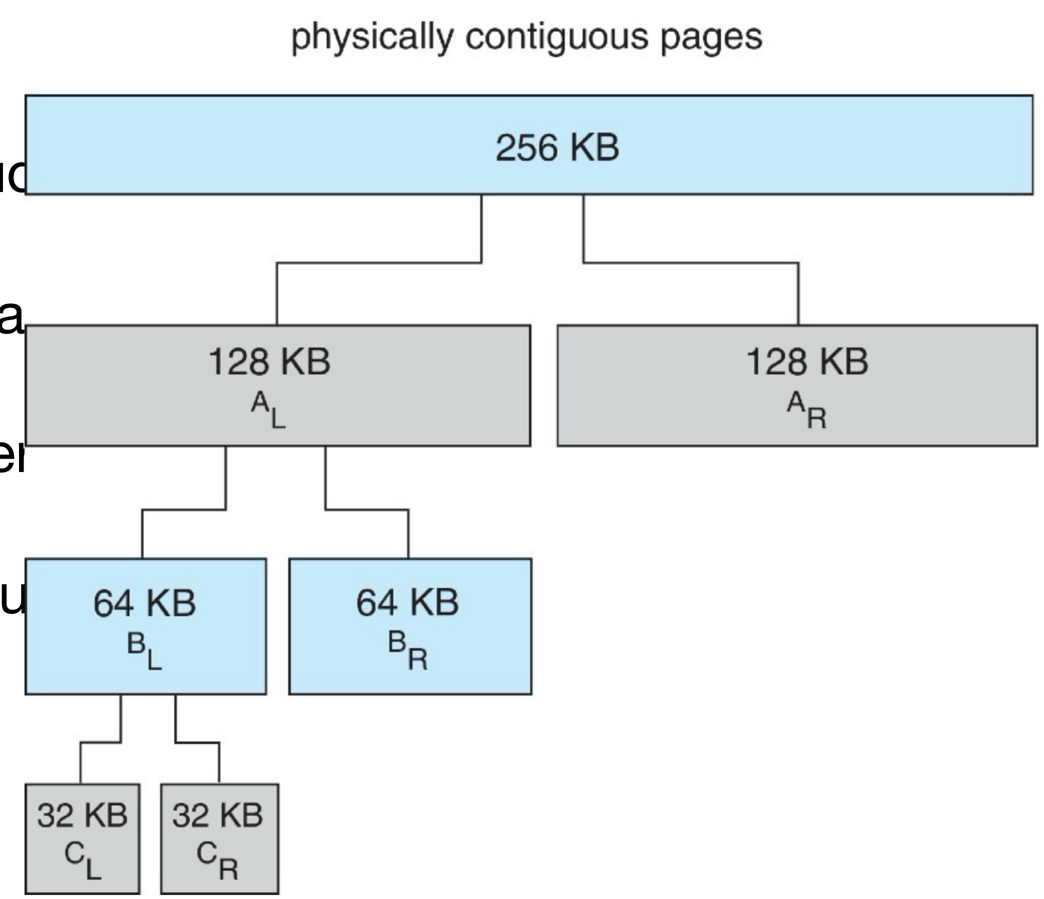
- Kernel memory is often allocated from a free-memory pool which is different from the free-frame list used to allocate frames to user-level processes
  - kernel code or data may not be subject to the paging system
  - kernel code may need to be in contiguous physical memory

Kernel memory allocator

- Kernel memory is often allocated from a free-memory pool which is different from the free-frame list used to allocate frames to user-level processes
  - kernel code or data may not be subject to the paging system
  - kernel code may need to be in contiguous physical memory
- buddy allocator
  - allocate memory from fixed-size segments or buddies (i.e., power-of-2 allocator)
  - for example a request for 11KB is satisfied with a 16KB segment
  - adjacent buddies can be combined to form larger segments
  - can lead to internal fragmentation because of rounding up to the highest power of 2

Kernel memory allocator

- Kernel memory is often allocated from a free-memory pool which is different from the free-frame list used to allocate frames to user-level processes
  - kernel code or data may not be subject to the paging system
  - kernel code may need to be in contiguous physical memory
- buddy allocator
  - allocate memory from fixed-size segments or buc
  - for example a request for 11KB is satisfied with a
  - adjacent buddies can be combined to form larger
  - can lead to internal fragmentation because of rou



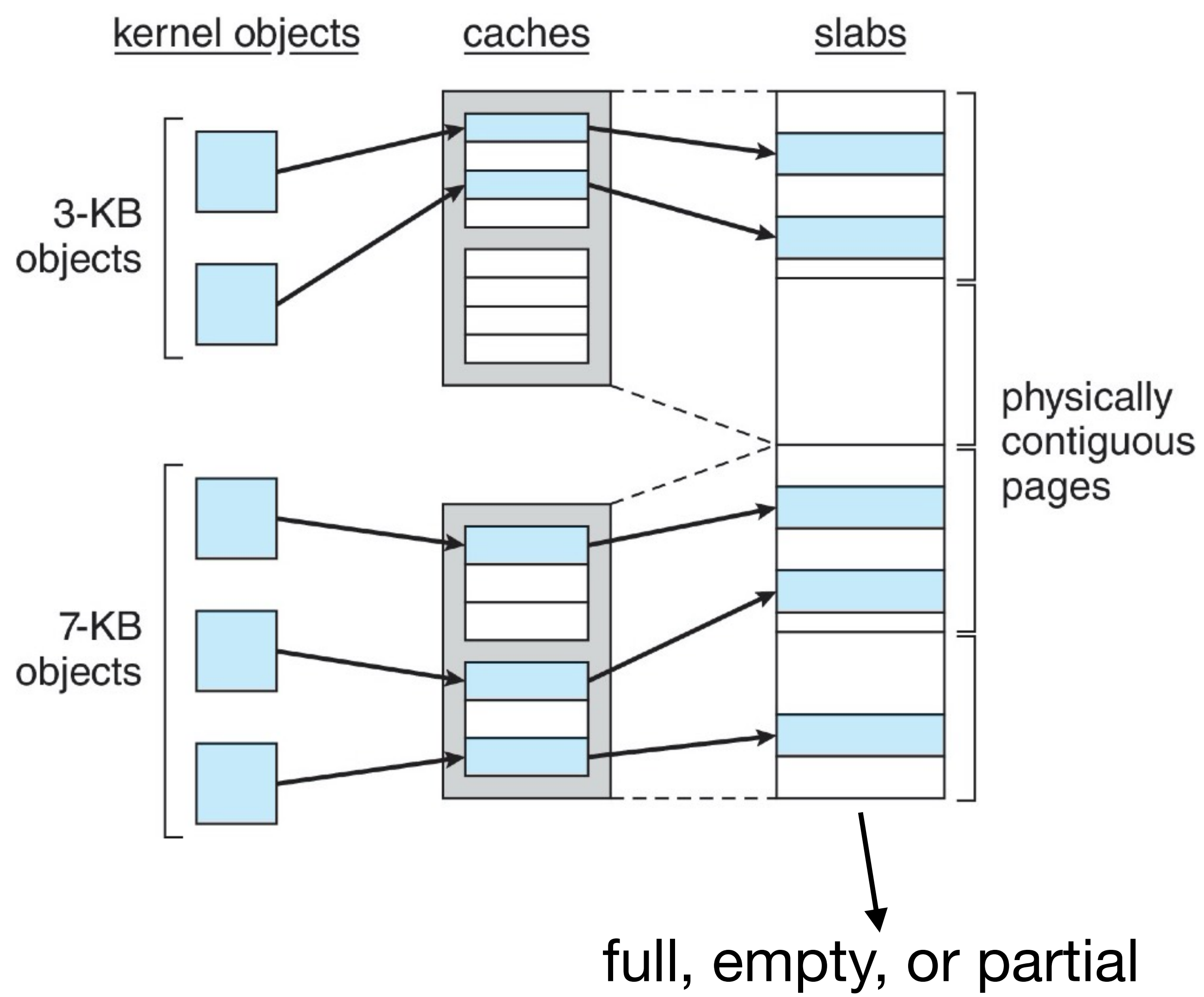
Slab allocator

- group objects of same size in a “slab”; a slab is made up of one or more **physically contiguous pages**; a cache consists of one or more slabs

- there is a separate cache for each kernel data structure (e.g., PCB, semaphores, file tables)
    - a cache contains many preallocated objects that are marked as free initially

- no memory is wasted due to internal fragmentation; memory allocation is faster

- used in Solaris and Linux



Page replacement in practice

UNIX and Linux use variants of the LRU algorithm* (e.g., the clock algorithm), Windows NT uses the clock algorithm (on a uniprocessor system) and random replacement (on a multiprocessor system)

- experiments show that all algorithms do poorly if processes have insufficient physical memory (less than half of their virtual address space)
- all algorithms approach optimal as the physical memory allocated to a process approaches the virtual memory size
- the more processes running concurrently, the less physical memory each process can have
- a critical issue the OS must decide is how many processes may share memory simultaneously and how many frames should be allocated to each process

* **https://www.kernel.org/doc/gorman/html/understand/understand013.html**