



# 19 Virtual Memory (2)

---

Paging Policies, Load control,  
Page Fault Handling, Case studies

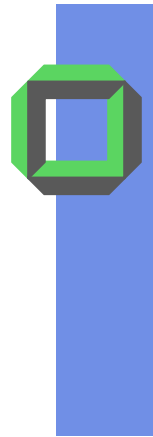
January 21 2009

WT 2008/09



# Roadmap of Today

- Introduction
- Orthogonal Paging Policies
- Page Replacement Policies
- Special Features
  - Page Frame Buffering
  - Segmentation (+ Paging)
  - Resident Set
  - Replacement Scope
- Load Control
- Swap Area
- Page Fault Handling



# VM Management Software

- VM management software depends on whether HW supports paging, segmentation or both
- $\exists$  only few pure segmentation systems (MCP)
- Segments are usually broken into pages
- We'll focus on issues associated with **pure paging**

## Goal of each paging policy:

- Performance, i.e. we must achieve a "low page-fault rate"
- Each page fault slows down an application

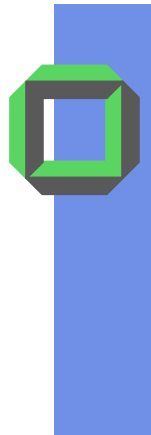


# Orthogonal Policies



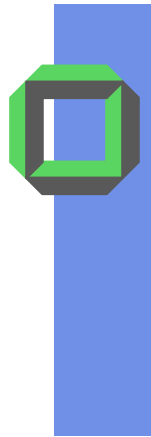
# Policies around Paging

- Fetch policy
  - When to transfer a page from disk to RAM (**page\_in**)
- Placement policy
  - Where to map a page (especially parts of **superpages**)
- Replacement policy
  - Which pair **<page/page frame>** to evict to be used for replacement in order to map a new page
- Clean policy
  - When to transfer a dirty page back to disk (**page\_out**)



## Fetch Policy

- 2 policies:
  - **Demand paging** transfers a page to RAM iff a reference to that page has raised a page fault
    - CON: "Many" initial page faults when a task starts
    - PRO: You only transfer what you really need
  - **Pre-Paging** transfers more pages from disk to RAM additionally to the demanded page
    - CON: Pre-paging is highly **speculative**
      - improves disk I/O throughput by reading chunks
      - wastes I/O bandwidth if page will never be used
      - can destroy the working set of another task in case of page stealing



# Placement Policy

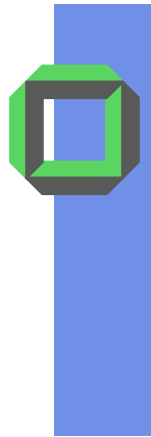
- For *pure segmentation* systems:
  - first-fit, next fit... are possible choices
  - (real issue, this can be quite complicated)
  
- For paging systems:
  - frame location often irrelevant
    - as long as all page frames are equivalent
    - except for some reserved or pinned memory locations for special purpose, interrupt vector, DMA etc.
  
  - frame location is relevant for variable sized pages
    - see transparent super pages



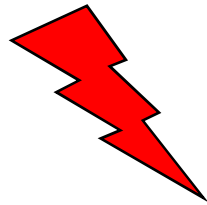
# Replacement Policy

- If there is no free page frame, select most fitting pair **<page/page frame>** for replacement
- Occurs whenever
  - the memory is completely exhausted in case of global replacement
  - the reserved memory per AS is exhausted in case of local replacement

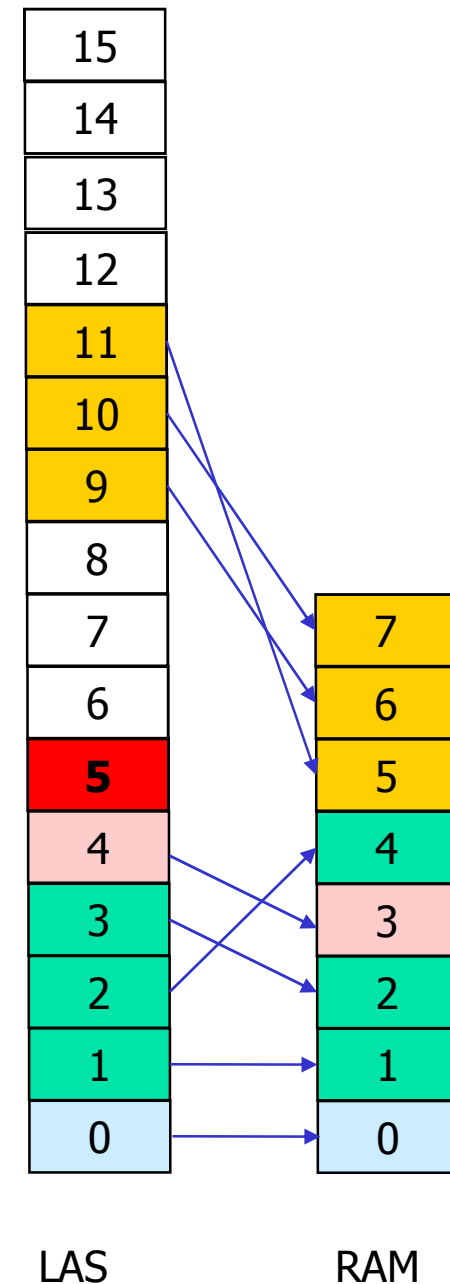




# Replacement Policy



- Page fault on page 5, but physical memory is completely allocated
- Which pair  $\langle \text{page/page frame} \rangle$  should we evict?





# Replacement Policy

- Not all page frames in memory can be replaced
  - Some pages are pinned to specific page frames:
    - Most of the kernel is resident, i.e. pinned
    - some DMA can only access physical addresses, i.e. their buffers must be pinned, too
    - A real-time task might have to pin some/all of its pages (**otherwise no one can guarantee its deadline**)
- OS might decide that set of pages considered for next replacement should be:
  - Limited to frames of the task having initiated page fault  
⇒ **local page replacement**
  - Unlimited, i.e. also frames belonging to other tasks  
⇒ **global page replacement**



## Replacement Policy (2)

- Decision for set of pages to be considered for replacement is related to the resident set policy:
  - Number of page frames allocated to one task is either
    - limited (easier performance guarantees) or
    - unlimited (adaptable to strange reference patterns)
  - No matter which set of page frames is considered for replacement, its policy tries to choose the
    - **most promising pair <page, page frame>**
  - *However, what is the most promising pair?*

⇒ Replacement algorithms



# Cleaning Policy

*When should we page-out a "dirty" page?*

## ■ Demand Cleaning

- a page is transferred to disk only when its hosting page frame has been selected for replacement by the replacement policy
- ⇒ page faulting activity must wait for 2 page transfers (out and in)

## ■ Pre-Cleaning

- dirty pages are transferred to disk before their page frames are needed
- ⇒ transferring large clusters can improve disk throughput, but it makes few sense to transfer pages to disk if most of them will be modified again before they will be replaced



## Cleaning Policy

- Good compromise achieved with page buffering
  - Recall that pages chosen for replacement are maintained either in a free (unmodified) list or in a modified list
  - Pages of the modified list can be transferred to disk periodically
  - $\Rightarrow$  A good compromise since:
    - not all dirty pages are transferred to disk, only those that have been chosen for next replacement
    - transferring pages is done in batch (improving disk I/O)



# Replacement Algorithms

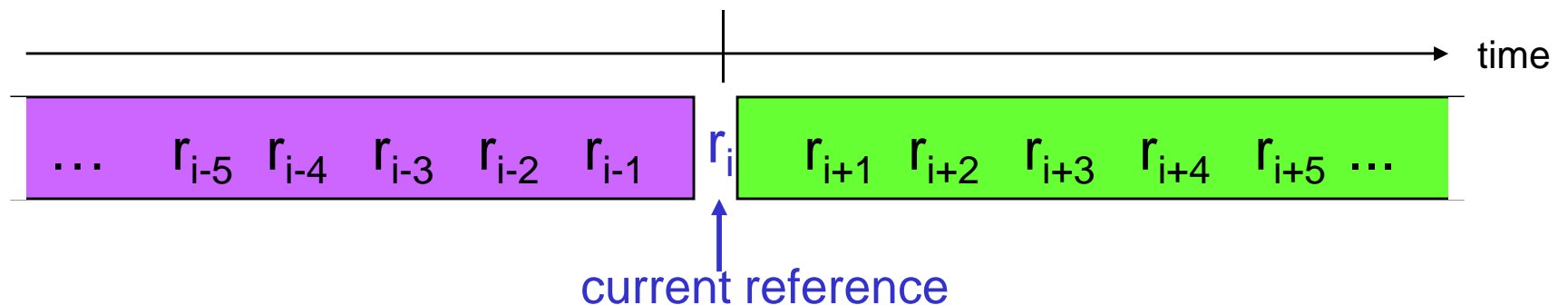
---

Theoretical Policies  
Practical Policies

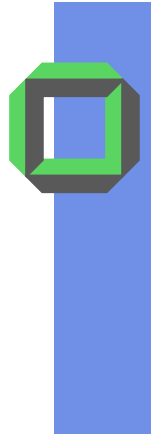


# Replacement Algorithms

- Observing the past may help to predict the future
- Observe the reference string and try to predict:
- “Very recently referenced pages will also be referenced in the near future”



Problem: 1. Which system-behavior can we observe easily?  
2. Which information can we maintain efficiently?



## Observable Data for Replacement

- “Nice to have” the following data for an appropriate decision:
  - time when the page has been transferred from disk to RAM
  - time of latest reference to the page
  - number of references to that page
    - Absolute number
    - Weighted number with aging





# Replacement Criteria (1)

- Time stamp when the page has been transferred to RAM
  - This info ~ the **age of a page**, very easy to implement.
- FIFO replacement policy is based on the age of a page.

⇒ *Replacement problem solved?*

No, this policy offers some **anomalies** (see exercises)



## Replacement Criteria (2)

- Latest reference time of the page
- A page that has not been referenced for a long period of time is a good candidate for replacement.
- LRU\*-based policies & NRU policy use this criteria.
- However, you won't implement an exact LRU-based algorithm, because overwriting the reference date with every reference is far **too much overhead**.

\*LRU = least recently used



## Replacement Criteria (3)

- Number of references to that page
- Idea: Pages that have been referenced a lot should remain in main memory.
- Thus the least frequently used page is the candidate for the next replacement.
- As well as the fact that you need a reference counter per page to be incremented with every reference, this replacement policy tends to some anomalous behavior **without** a proper aging mechanism.



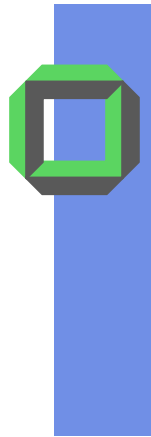
# Overview: Replacement Algorithms

- Belady-Algorithm (optimal algorithm)
  - Random (low end performance)
  - Not recently used (NRU)
  - **First-in, first-out (FIFO)**
  - (enhanced) FIFO with second chance
  - Least recently used (LRU)
  - **Clock**
  - Aging: Recently Not Used (RNU)
  - Least frequently used (LFU)
  - **Working Set**
  - WSClock
- To compare with



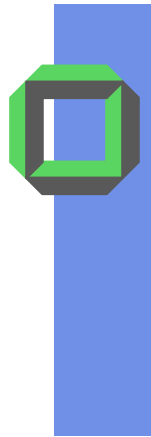
# Belady Algorithm

- Select the pair <page/page frame> for which the time till the next reference is maximal, i.e. that won't be used for the longest time (if ever)
- **Pros**
  - Minimal number of page faults
  - Off-line algorithm for comparison analysis
- **Cons**
  - impossible (or very hard) to implement
    - need to know the future (or measure reference string in a test environment)
    - only clairvoyants believe to see future events



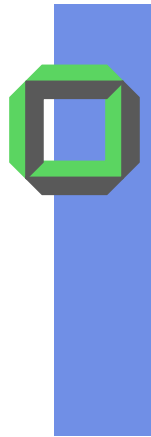
## Not Recently used (NRU)

- Randomly select a <page/page frame> from the following list (in this order)
  - Not referenced and not modified
  - Not referenced and modified
  - Referenced and not modified
  - Referenced and modified
- **Pros**
  - Easy to implement?
- **Cons**
  - Far from optimal
  - **Huge overhead** for maintaining the above lists



## FIFO Policy

- Manage mapped page frames as a circular buffer
  - When buffer is full, the oldest page is replaced
  - Hence: first-in, first-out, although ...
    - old pages have been referenced in the very past
    - old pages have been referenced very frequently
- Comment: Easy to implement
  - requires only a pointer that circles through the page frames of the task (local replacement)
  - requires only a pointer circling through the page frames of all tasks (global replacement)



# FIFO Policy with Second Chance\*

## ■ Algorithm

- Check R-Bit of oldest page/page frame
- If it is 0, replace it
- If it is 1, reset R-Bit, put page/page frame to the end of the FIFO list, and continue searching

## ■ Pros

- Relatively fast and do not replace heavily used page

## ■ Cons

- The worst case may take a long time, i.e. search the complete FIFO-list

\*FIFO with second chance is similar to the clock algorithm



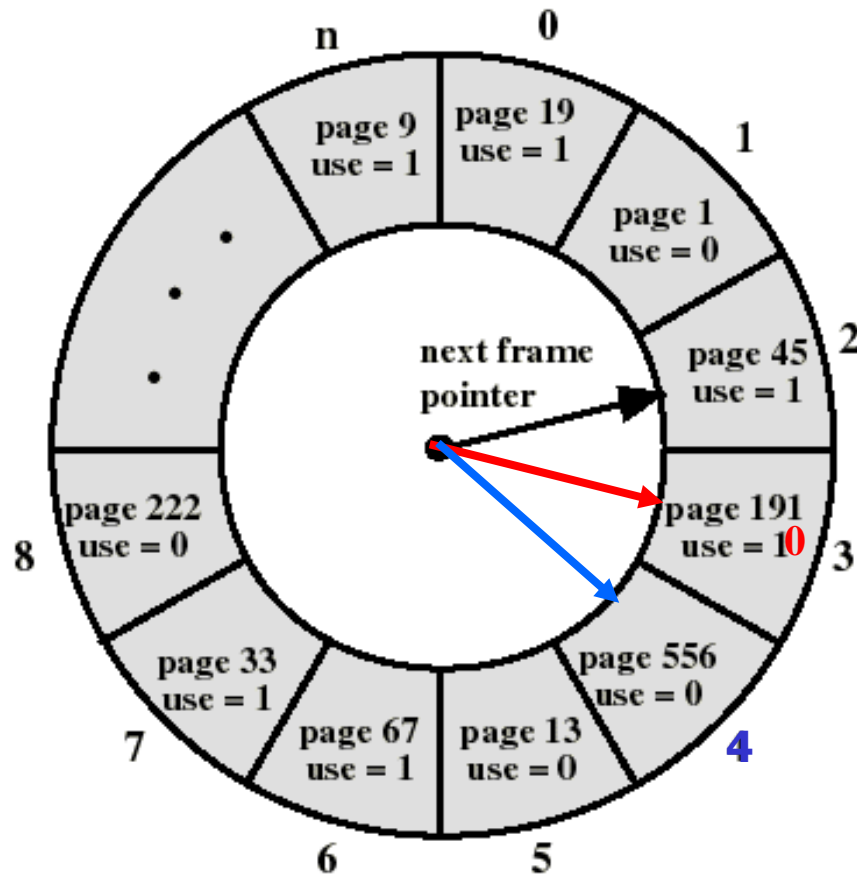


## Clock Policy

- Set of page frames to be replaced is considered as a circular buffer
- R(eference)-bit for each frame is set to 1 whenever the corresponding page is referenced (also upon loading in case of demand paging)
- During search for replacement, in all controlled page frames the R-bit with value 1 is changed to 0, indicating that this page gets a second chance to be referenced again before the next replacement step
- **Replace first frame that has the R-bit = 0**
- Whenever a page frame has been replaced, a pointer is set to point to the next page frame in the buffer



# The Clock Policy

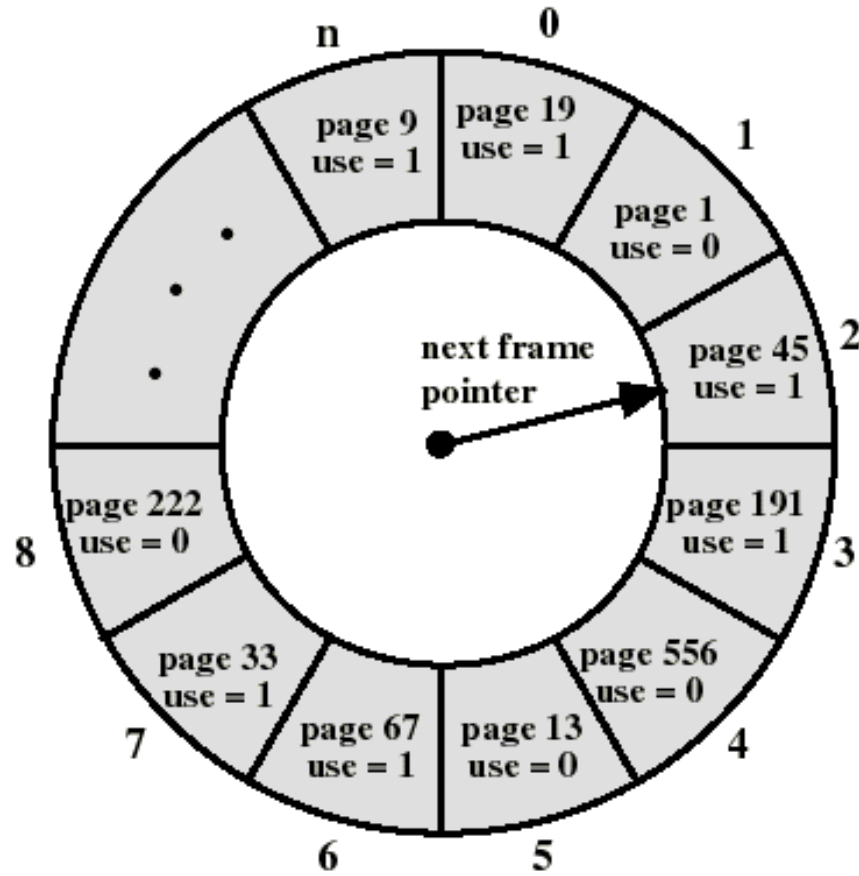


(a) State of buffer just prior to a page replacement

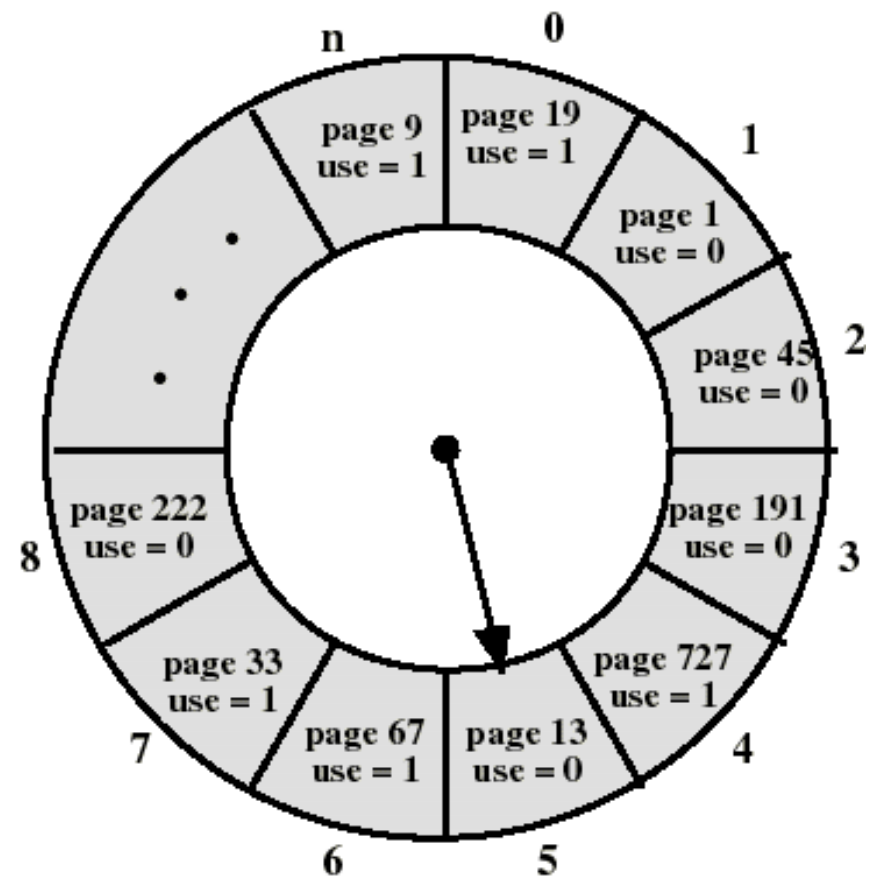
1. A page fault occurs, because page 727 is currently not in main memory.
2. Look for a candidate to be replaced
3. In a previous replacement step page 45 on page frame 2 had been replaced
4. Starting from next frame pointer we are looking through the following page frames, resetting their reference bits.
5. Continue this resetting until we find the first page frame with reference bit = 0. Its page frame number is 4, containing page 556 which has to be replaced, because in the past it was not referenced any more: it didn't use its second chance.



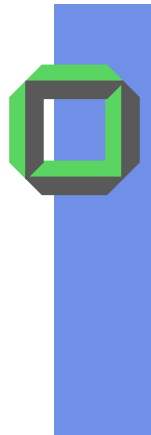
# The Clock Policy



(a) State of buffer just prior to a page replacement

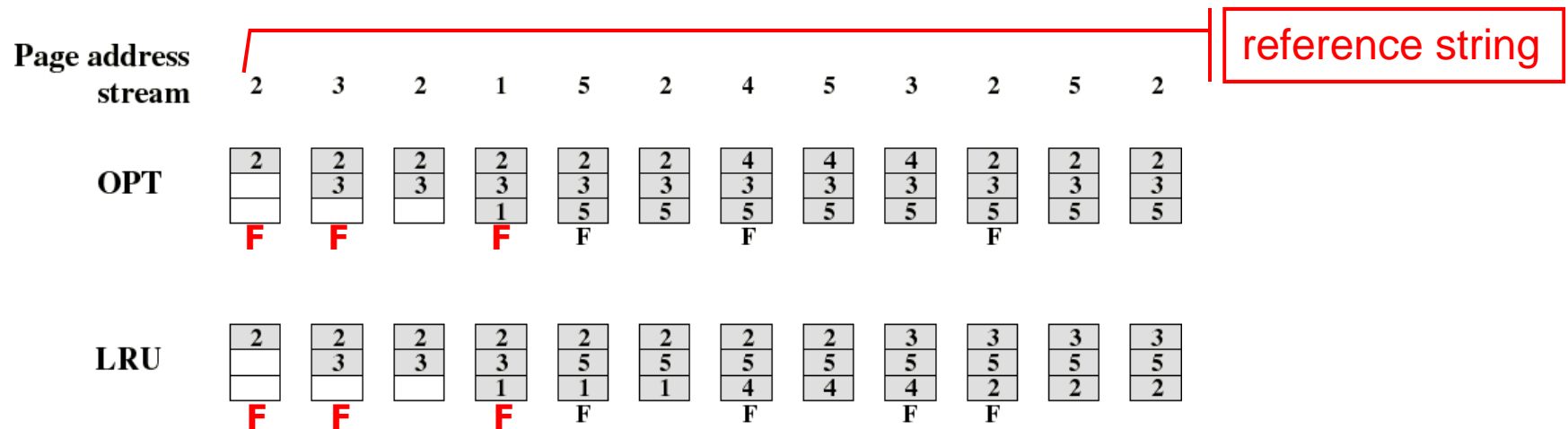


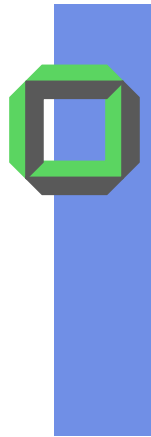
(b) State of buffer just after the next page replacement



# LRU Policy

- Replaces page that has not been referenced for the longest period of time
  - By the principle of locality, this should be the page least likely to be referenced in the near future
  - performs “nearly as well as the OPT”





# LRU Policy

- Algorithm
  - Replace page/page frame that hasn't been used for the longest time
- How to implement?
  - Order the pages/page frames by time of reference
  - Need timestamp (per page or) per page frame
- Pros
  - Quite good to approximate the optimal policy, because it follows the idea of locality
- Cons
  - Difficult to implement



## Implementation of LRU Policy

- Each page table entry could be enhanced with a time stamp of the current memory reference.
- LRU page is the page with the minimum time value
- Expensive hardware is needed, implying a great deal of avoidable overhead.

**Never implemented that way!**



## Implementation of LRU Policy

- Upon a reference, each referenced page is deleted from LRU stack and then pushed on the LRU stack.
- The LRU page is the page at the LRU stack bottom.
- Expensive hardware, significant overhead.

**Not implemented per reference**

- “Approximate LRU algorithms” are used instead



# Approximation of LRU

- Use CPU ticks
  - For each memory reference, store the ticks in its PTE
  - Replace the page/page frame with the minimal tick value
- Use a smaller counter (e.g. 8 bit)
- Divide the set of page /page frames according to their behavior since the last page fault



Pages reference since  
the last page fault



Pages not reference since  
the last page fault

- Replace a page/page frame from the list of not referenced pages since the  $(n-1)$  last page faults,  $n > 2$





# Aging: Recently Not Used (RNU)\*

- Algorithm
  - Shift reference bits into counters (*when?*)
  - Select the page/page frame with the smallest counter
- *Main difference between LRU and NFU?*
  - NFU has a shorter history (only counter length)
  - How many bits are enough?
    - In practice 8 bits are sufficient
  - Pros:
    - Requires only a reference bit
  - Cons
    - Requires looking at all counters

pf <sub>0</sub>	0000 0000	0000 0000	1000 0000	0100 0000	1010 0000
pf <sub>1</sub>	0000 0000	1000 0000	0100 0000	1010 0000	0101 0000
pf <sub>2</sub>	1000 0000	1100 0000	1110 0000	0111 0000	0011 1000
pf <sub>3</sub>	0000 0000	0000 0000	0000 0000	1000 0000	0100 0000

\*some claim this NFU (*evaluate this notion*)



# Working Set Page Replacement

- On a page fault, scan through all page/page frames of the faulting task, i.e. local replacement
- If R-Bit is set, record current time  $t$  for the page/page frame (and reset R-Bit)
- If the R-Bit is not set, check time of latest use
  - If this page has not been referenced within  $\Delta$  time units, it no longer belongs to the WS and is a candidate for page-replacement
  - Replace the oldest non WS page
- Add the faulting page to the working set  $W(t, \Delta)$



## Details of WS Replacement

- If R-Bit was not set, check time of last reference
  - If page has been referenced within  $\Delta$ , go to the next
  - If page has not been referenced within  $\Delta$  and if dirty-bit is set, schedule page for page out and go to next page/frame
  - If page has not been referenced within  $\Delta$  and dirty-bit is not set, page is a candidate for replacement



# Special Features

---

Page Frame Buffering

Segmentation

Resident Set

Replacement Scope



# Page Frame Buffering (1)

- Pages to be replaced soon are kept in main memory
- Two lists are maintained to support replacement
  - a free page list for frames having not been modified since they have been brought in (no need to transfer to disk)
  - a modified page list for frames having been modified (need to transfer to disk before replacing)
- Page selected for lazy replacement is inserted into one list and it is unmapped (i.e. its valid bit is reset )

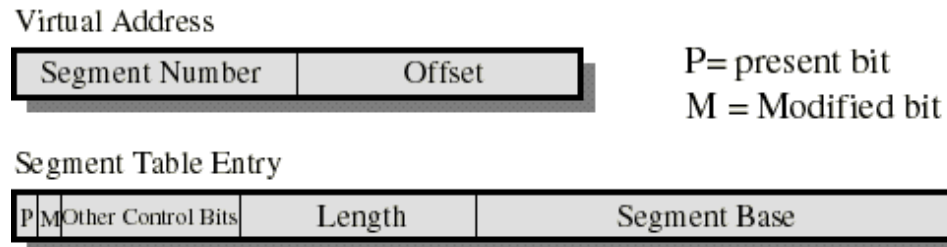


## Page Frame Buffering (2)

- If a page of these 2 lists is referenced again ( $\Rightarrow$  light page fault), page only has to be removed from list and has to be remapped
- No expensive transfer from disk or even additional transfer to disk is necessary
- The list of modified pages enables that dirty pages are transferred together, i.e. in a cluster



# Virtual Segmentation



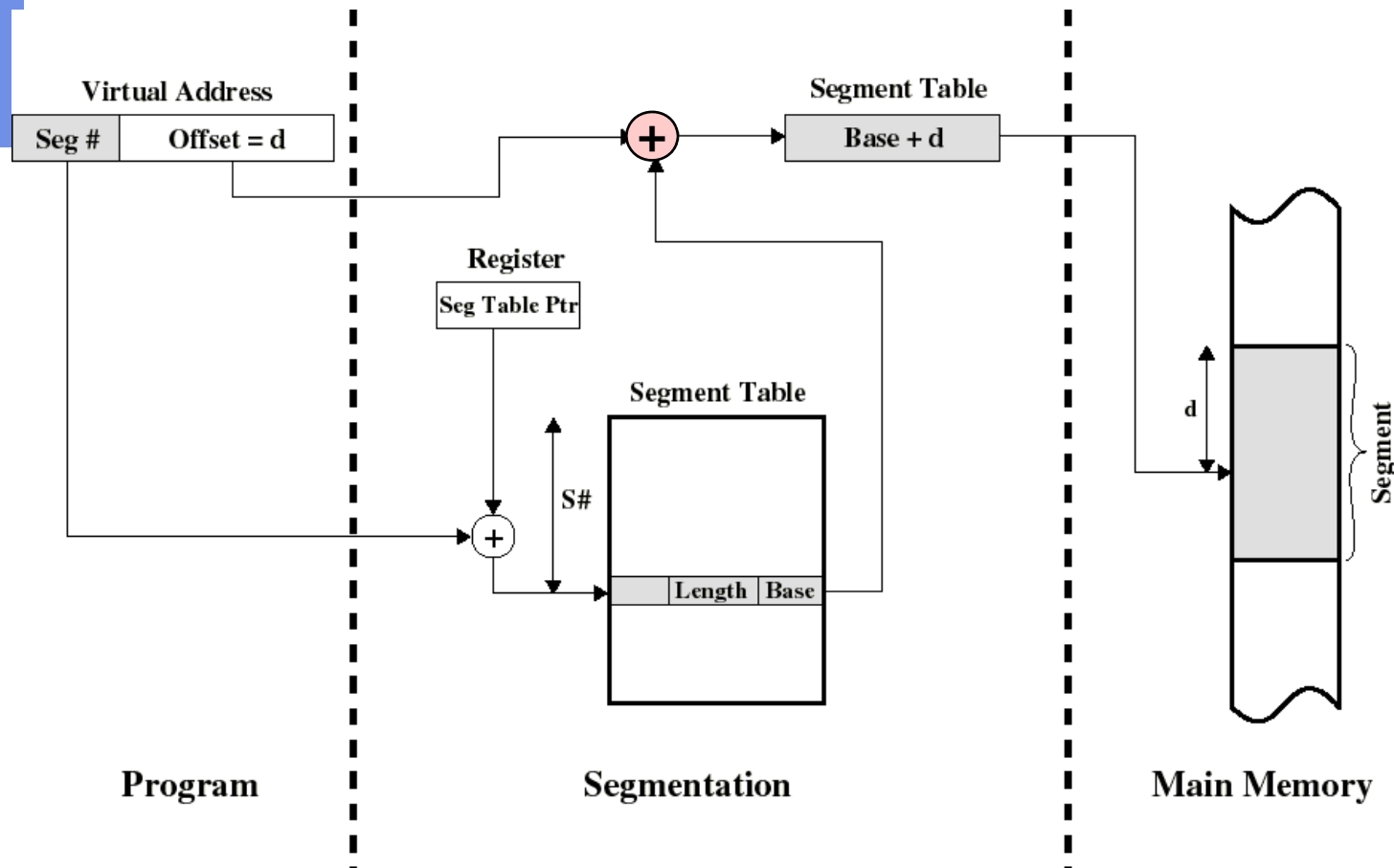
Similarly to paging, a segment table entry STE contains a present bit and a modified bit

If the segment is in main memory, the entry contains the starting address and the length of the segment

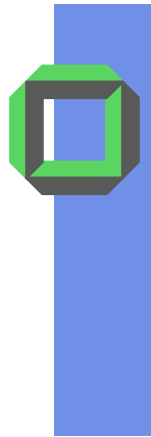
Other control bits may be present if protection and sharing is managed at segment level

Logical to physical address translation ~ to paging except that the offset has to be added to the starting address.

# Address Translation with Segmentation







# Comments on Segmentation

- Each STE contains starting address and length of the segment ⇒
  - segment can thus dynamically grow or shrink as needed
  - address validity easily checked with length field
  - partial overlapping of segments is easily done
- Segments may lead to external fragmentation, and swapping segments in and out is more complicated
- Protection and sharing at segment level is visible to programmers (pages are artificial entities)

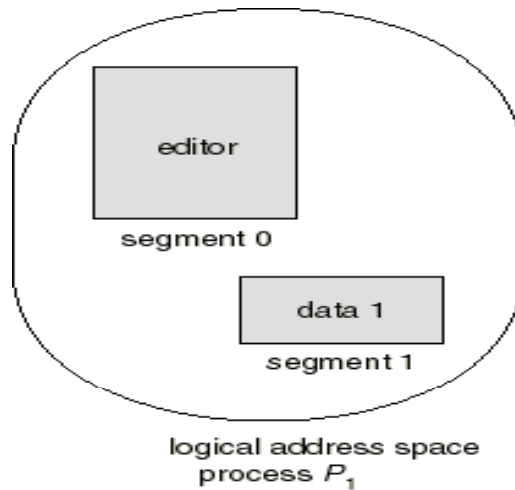


# Sharing and Segmentation

- Segments are as easily shared as pages are
- Example: code of a text editor shared by many users
  - only one copy is kept in main memory
  - but each task needs to have its own private data segment
- Remark:  
Some system architects also call the “superior pages” of a two-level paging system “segments.”
- Review:  
Burroughs’ MCP (written in Espol) supported only segments

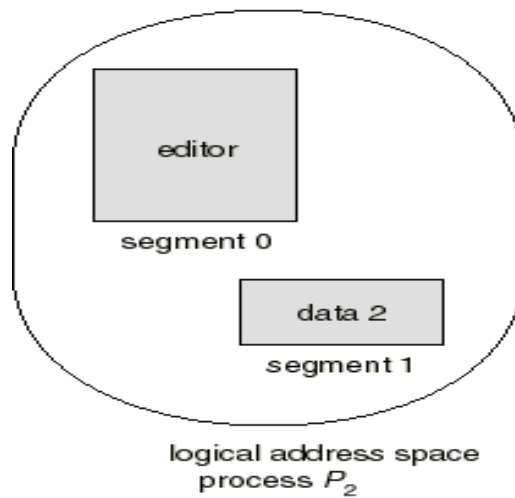


# Sharing of Segments



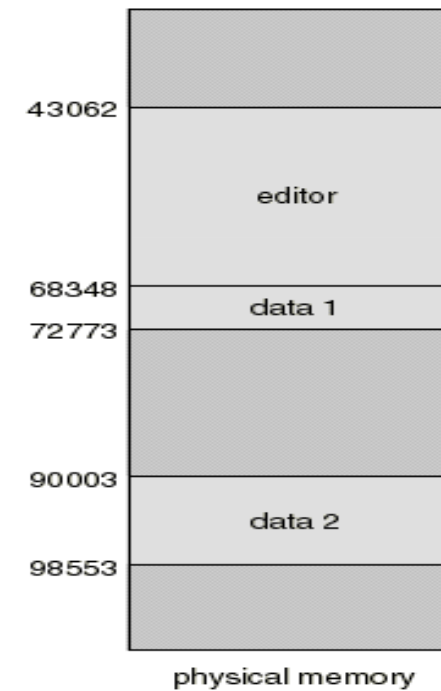
	limit	base
0	25286	43062
1	4425	68348

segment table  
process  $P_1$



	limit	base
0	25286	43062
1	8850	90003

segment table  
process  $P_2$



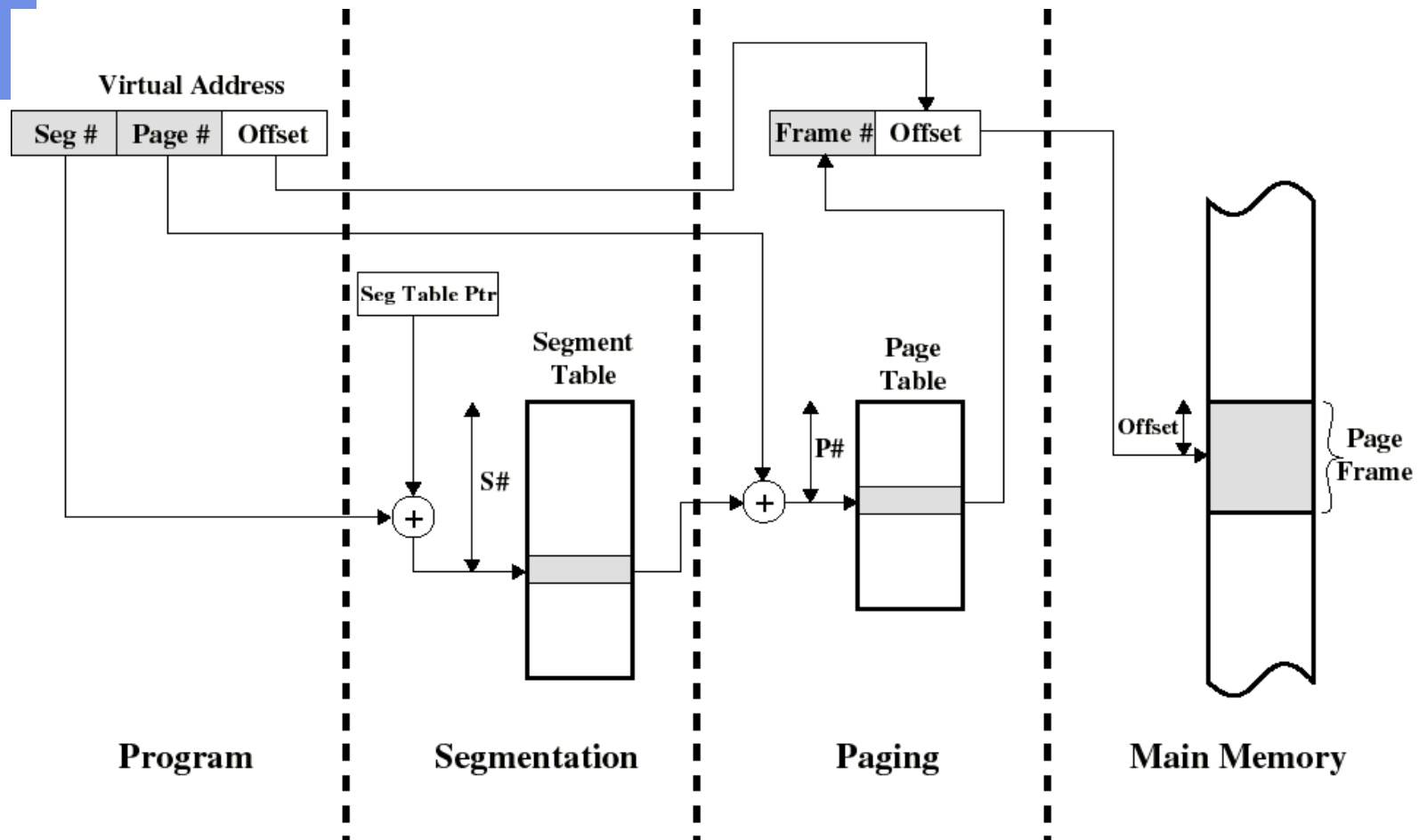


# Combined Segmentation and Paging

- Some OSes offer a combination of both address-translation methods: segmentation and paging
- Each task has:
  - one segment table
  - several page tables: one (variable-sized) page table per segment
- Virtual address consist of:
  - Segment number: used to index the segment table who's entry gives the starting address of the page table for that segment
  - Page number: used to index that page table to obtain the corresponding frame number
  - An offset or displacement  $d$ : used to locate the cell in the frame



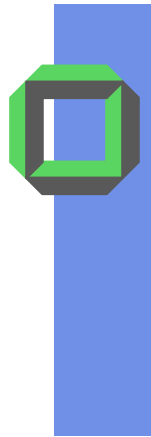
# Address Translation in a Segmentation/Paging System





## Resident Set Size

- OS controls maximum number of page frames that a task or a process might get
  - If a task/process has a high page fault rate, maybe too few frames are currently allocated to it
  - If the system has a high page fault rate, perhaps too many tasks/processes have been activated
  - *Complete the case:*
    - *"If the page fault rate in the system is very low, ..."*



# Resident-Set Size

## ■ **Fixed-allocation policy**

- Fixed number of frames remaining over run time
  - Determined at load time
  - Depending on type of application

## ■ **Variable-allocation policy**

- Number of frames of a task may vary over time
  - May increase if page fault rate is high
  - May decrease if page fault rate is low
- Requires more overhead to observe and to control the behavior of an activated task



# Replacement Scope

- Local replacement policy
  - chooses among only those frames that are allocated to the task having issued the page fault
- Global replacement policy
  - each unlocked frame in the whole system might be a candidate for page replacement
- Analyze the following 3 design alternatives





## Fixed Alloc./Local Replacement Scope

- When a page fault occurs: page frames considered for replacement are local to the page-faulting task
  - Number of allocated frames remains constant
  - Any replacement algorithm can be used, however, restricted to the resident set of task
- Problem: difficult to estimate correct number of page frames
  - if too small  $\Rightarrow$  page fault rate too high
  - if too large  $\Rightarrow$  some page-frames not really used, could decrease multiprogramming level



## Variable Alloc./Global Replac. Scope

- Simple to implement--adopted by some commercial OS (like Unix SVR4)
- List of free frames is maintained
  - When a task issues a page fault, a free frame (from this list) is allocated to it
  - Hence the number of frames allocated to a page-faulting task can increase
  - The choice for the task that will lose a frame is arbitrary: that's why this is far from optimal
- Page buffering alleviates this problem since a page may be reclaimed if it is referenced again in time



## Variable Alloc./Local Replac. Scope

- **“Good combination”** (according to Stallings, e.g. Windows NT)
- At load time, allocate a certain number of frames to a new task, based on the application type or measured in earlier runs
  - use either pre paging or demand paging
  - to fill up the allocation
- When page fault  $\Rightarrow$  select page to be replaced from resident set of the “page-faulting task”
- Periodically reevaluate the allocation provided and increase or decrease it to improve load situation, thus increasing overall performance



# Load Control

---

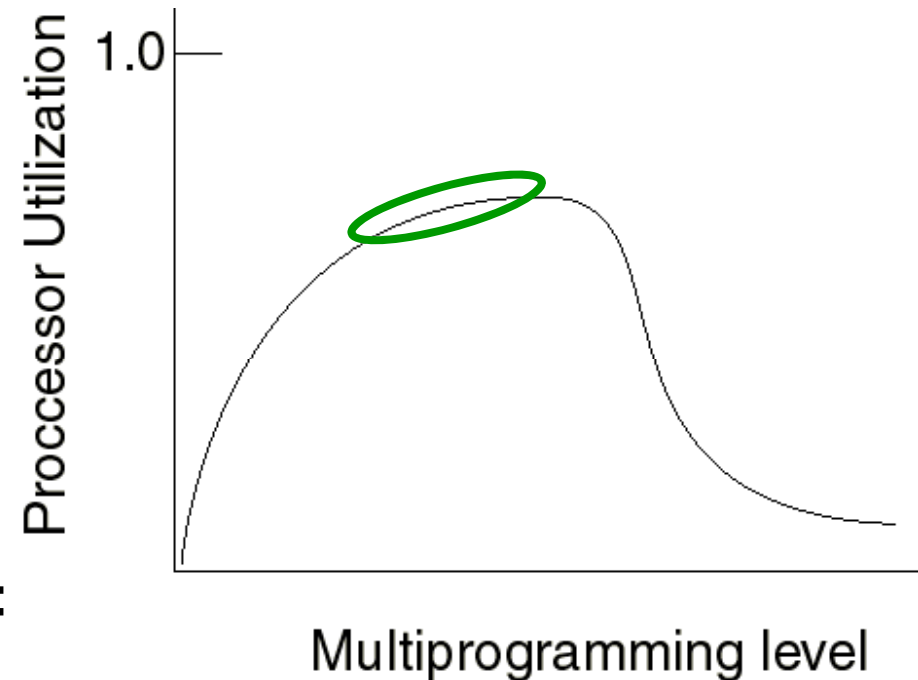
Working-Set Model  
Page Fault Frequency Model

# Load Control

Too few tasks/processes  $\Rightarrow$   
all tasks may be blocked  
and CPU will be idle

**If too many tasks  $\Rightarrow$**   
resident size of each task/  
process will be too small and  
flurries of page faults will result:

# Thrashing





## Load Control (2)

- A working set or page fault frequency algorithm implicitly incorporates load control
  - only those tasks whose “resident sets” are sufficiently large are allowed to execute
- Another approach is to explicitly adjust the multi-programming level so that the mean time between page faults equals the time to handle a page fault
  - performance studies indicate that this is the point where processor usage is at a maximum





# Task Suspension

- Load control  $\Rightarrow$  swap out tasks.

Selection criteria to deactivate a task:

- Page faulting task
  - this task obviously does not have its working set in main memory, thus it will be blocked anyway
- Last task activated
  - this task is least likely to have its working set already resident (at least with pure demand paging)
- Task with smallest resident set
  - this task requires the least future effort to reload
- Task with largest resident set
  - this task will deliver the most free frames



# Load Control with Working Set Policy

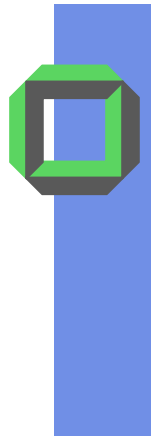
- WSP is a variable-allocation method with local scope based on the principle of locality
- Notion:  
Working set of task  $T$  at time  $t$   $W_T(\Delta, t)$ , is the set of pages that have been referenced in the last  $\Delta$  virtual time units
  - virtual time = time elapsed while the task was in execution (e.g:  $\sim$  number of instructions executed)
  - $\Delta$  is a window of time
  - For any fixed  $t$ ,  $|W(\Delta, t)|$  is a non decreasing function of  $\Delta$
  - $W_T(\Delta, t)$  is an approximation of the program's locality





# The Working Set Policy

- $W_T(\Delta, t)$  first grows when T starts executing and then tends to stabilize due to the principle of locality
- It may grow again when the task enters a new phase (transition period, e.g. procedure) up to a point where working set contains pages from two phases
- It then decreases again, spending some time in the new phase



# Working Set Policy

- Working set concept suggests the following policy to determine size of the resident set:
  - Monitor the working set for each task
  - Periodically remove from the task's resident set those pages that are no longer in its working set
  - When the resident set of a task is smaller than its working set, allocate more frames to it
  - If not enough frames are available, suspend task (until more frames are available), i.e. a task may execute only if its working set fits into the main memory



# Implementing Working Set Policy

- Problems implementing a pure working set policy
  - measurement of the working set for each task is impractical
    - need to time stamp a referenced page per reference
    - need to maintain a time-ordered queue of referenced pages for each task
  - the **optimal value for  $\Delta$**  is unknown and time-varying due to transient and stable execution phases
- Solution: monitor the “page fault rate” instead of the working set



# Page Fault Frequency Model

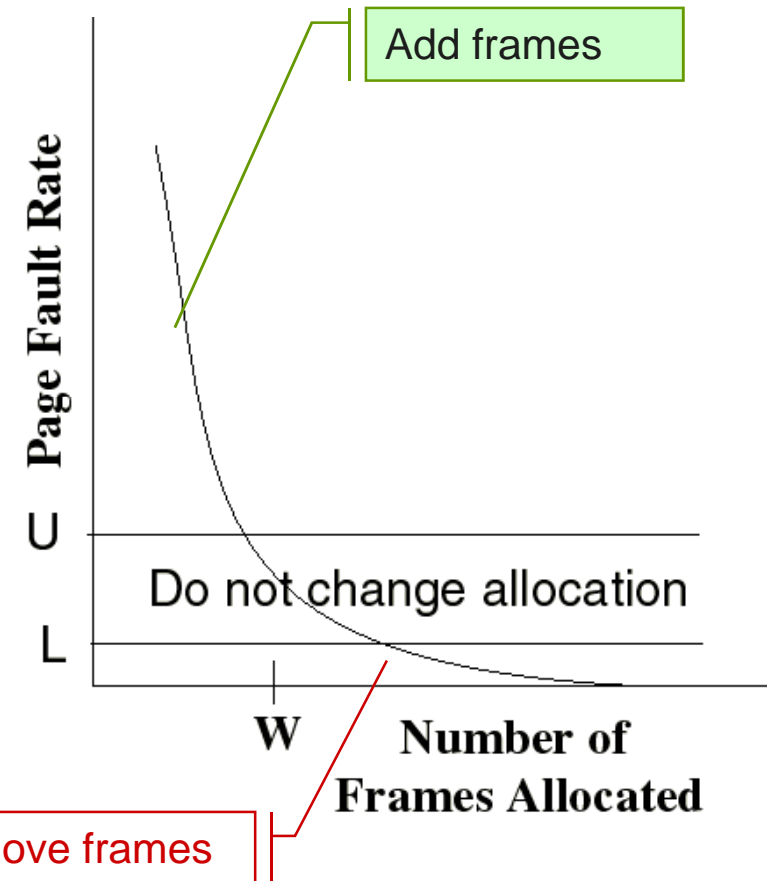
Define an upper bound  $U$  and lower bound  $L$  for page fault rates

Allocate more frames to a task if its fault rate is higher than  $U$

Allocate fewer frames if its fault rate is  $< L$

The resident set size should be close to the working set size  $W$

We suspend the task if  $PFF > U$  and no more free frames are available



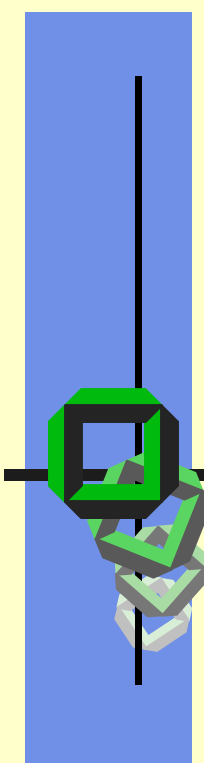
Remark: A variant of this policy is used in Linux



# Swap Area

---

Study of your own



# Page Fault Handling

---



# Organization of a Swap Area

- Study this battle field as a **KIT** student
- Assume  $n \geq 1$  disks to keep the swap area

## Definition:

Swap area is the sum of extra disk blocks used to hold unmapped pages

## Questions:

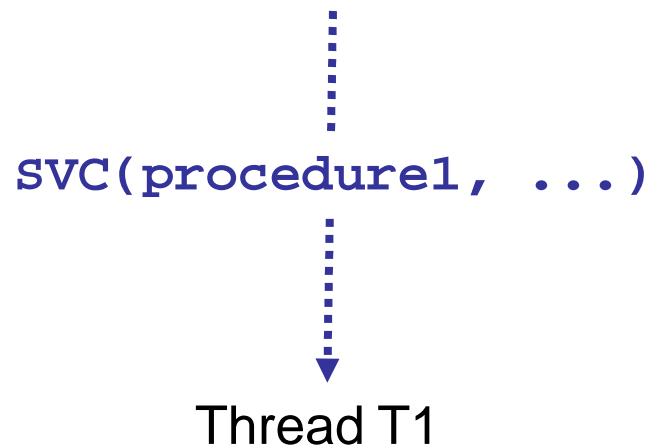
1. *Which disk units to use to install the swap area?*
2. *Which memory management to store AS regions?*
3. *Do we need swap space for code regions?*



# Automatic Replacement

## Assumption:

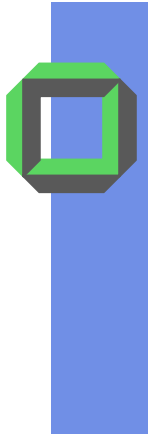
T1 runs & tries to **access** a **data item** and/or **fetch** an **instruction** currently not mapped, but belonging to its address space  $\Rightarrow$  *page fault exception*



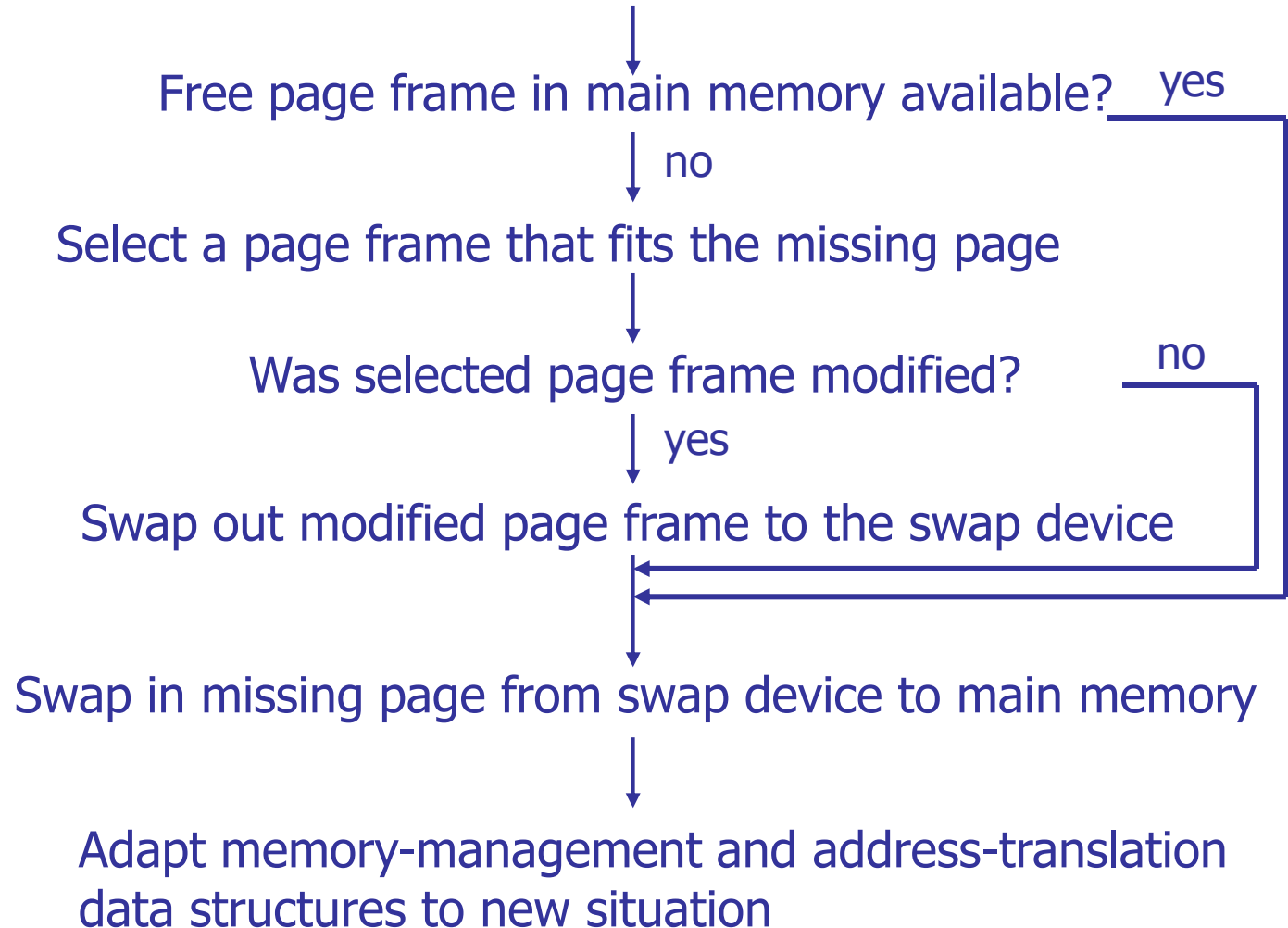
**Page Fault Exception**

?

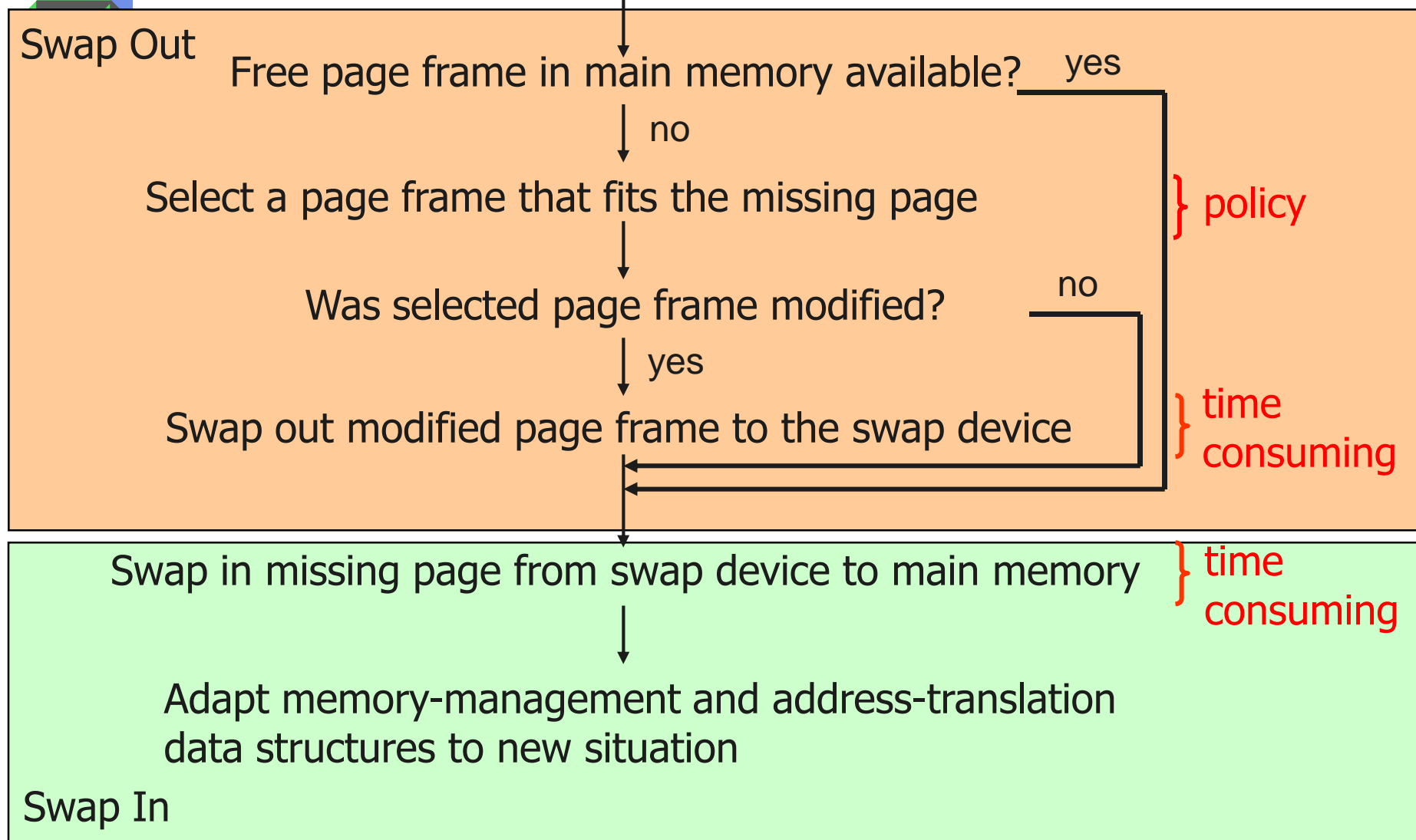




## Page Fault Exception



## Page Fault Exception



**Remark:** You cannot handle all above activities in one exception handler!  
(Contradicts main characteristics of short exception handling!)



## Conclusion

Cutoff between short page fault interrupt handling routine and long-term page fault handling thread(s).

page fault exception

send("my pager", <pno, my\_TID>)  
receive(from my page, "page frame")

### Remark:

We can further enhance page fault handling, if we can assure that there are always some free page frames.

We need another periodic thread (see Unix page daemon) freeing page frames in advance (see exercises).

pager

receive("anonymous", <pno, TID>)

decide(page frame)

swap\_out(page frame)

swap\_in(pno)

send(TID, "page frame")



## Further Enhancement

- You can use the producer/consumer template for further enhancement:
- Periodically a *page daemon* produces *free page frames* by managing the two lists:
  - Free list containing unmodified or clean pages
  - Modified list containing modified or dirty pages



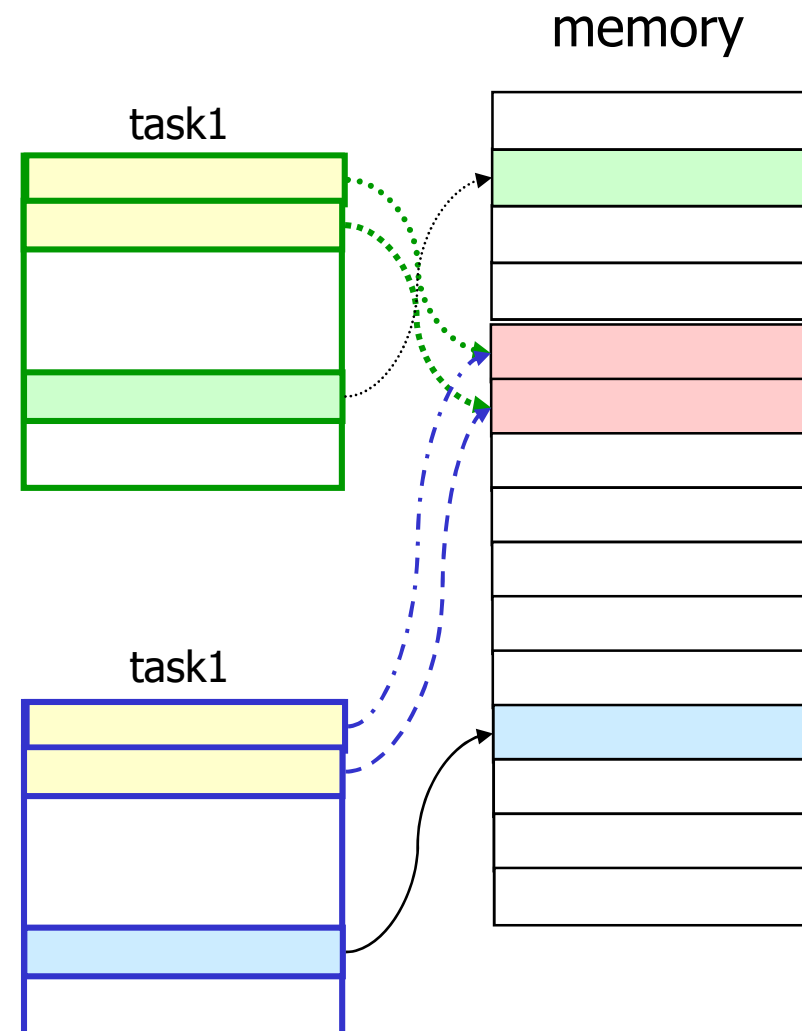
# Pinning

- *When do we need pinning?*
  - When *DMA is active*, we don't want to page out pages that DMA is using directly
  - *How to implement pinning?*
    - Data structure to remember all pinned pages
    - Page replacement only replaces other page/page frames
    - Special calls for pinning and unpinning
- *If entire kernel is in physical memory, do we still need pinning?*



# Shared Pages (1)

- Two or more PTEs from different tasks share the same physical page frame
- Special virtual memory calls to implement shared pages
- *How to delete an address space that shares pages?*
- *How to page in and out shared pages?*
- *How to pin and unpin shared pages?*
- *How to calculate working sets for shared pages?*





## Shared Pages (2)

- Replacement algorithm wants to replace a *shared page, always possible?*
  - No, if otherwise used then this decision might be wrong
    - With local replacement we do not know the replacement characteristic of the other task(s) concerning this shared page
      - May be another task is heavily using this page
      - Don't replace shared pages  $\Rightarrow$  all frames contain shared pages?
    - With global replacement we are able to collect all necessary information around that shared page  $\Rightarrow$  we need an extra data structure containing info on shared pages (task counter)
      - LFU replacement without aging will favor shared pages
      - You can age the usage counter of a shared page differently



# Copy-On-Write

- Child uses same mapping as parent
- Make all pages read-only
- Make child ready
- On a read nothing happens
- On a write, generate an access fault
  - Map to a new page frame, if page was previously a read/write (otherwise we have a normal access fault)
  - Make a copy
  - Restart instruction

