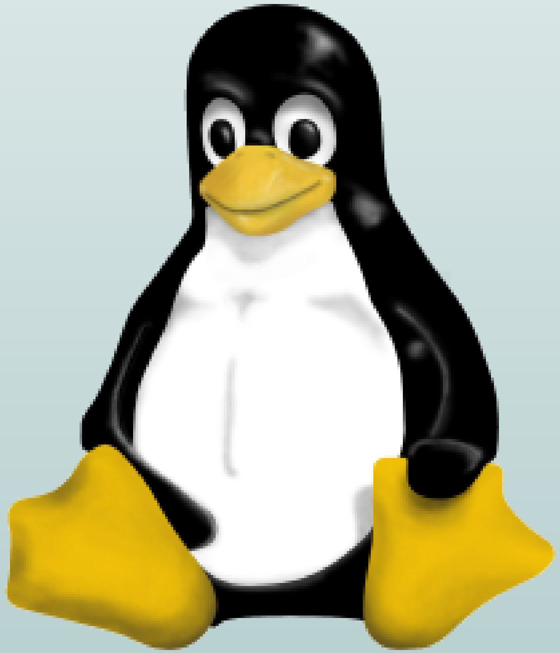
Linux Kernel Hacking Free Course, 3rd edition

D.P. Bovet
University of Rome “Tor Vergata”

Heuristic programming in Linux 2.6



March 1, 2006

Purpose of the talk

Purpose of the talk

- While trying to describe some important Linux kernel functions, I came to the conclusion that some of them are much more difficult to analyze than others

Purpose of the talk

- While trying to describe some important Linux kernel functions, I came to the conclusion that some of them are much more difficult to analyze than others
- Linux programmers are among the best in the world, poor coding is not a valid explanation for the existence of difficult functions

Purpose of the talk

- While trying to describe some important Linux kernel functions, I came to the conclusion that some of them are much more difficult to analyze than others
- Linux programmers are among the best in the world, poor coding is not a valid explanation for the existence of difficult functions
- I'll try to convince you that difficult functions do exist only when kernel designers rely on the **dark side of programming**

Outline of the talk

Outline of the talk

- Heuristic programming

Outline of the talk

- Heuristic programming
- Heuristic programming in Linux

Outline of the talk

- Heuristic programming
- Heuristic programming in Linux
- Page frame reclaiming

Heuristic programming

Problem solving techniques

Problem solving techniques

- Two main ways to solve a problem using a computer

Problem solving techniques

- Two main ways to solve a problem using a computer

- Algorithmic programming: we use a single algorithm to solve our problem

Problem solving techniques

- Two main ways to solve a problem using a computer
- Algorithmic programming: we use a single algorithm to solve our problem
- Heuristic programming: we solve a problem by a method of trial and error, in which the success of each attempt at getting the “best” solution is assessed and used to improve the subsequent attempts

Other definitions of heuristic programming

Other definitions of heuristic programming

- A branch of artificial intelligence which uses common-sense rules drawn from experience to solve problems

Other definitions of heuristic programming

- A branch of artificial intelligence which uses common-sense rules drawn from experience to solve problems
- Self learning programs that get better with experience

Other definitions of heuristic programming

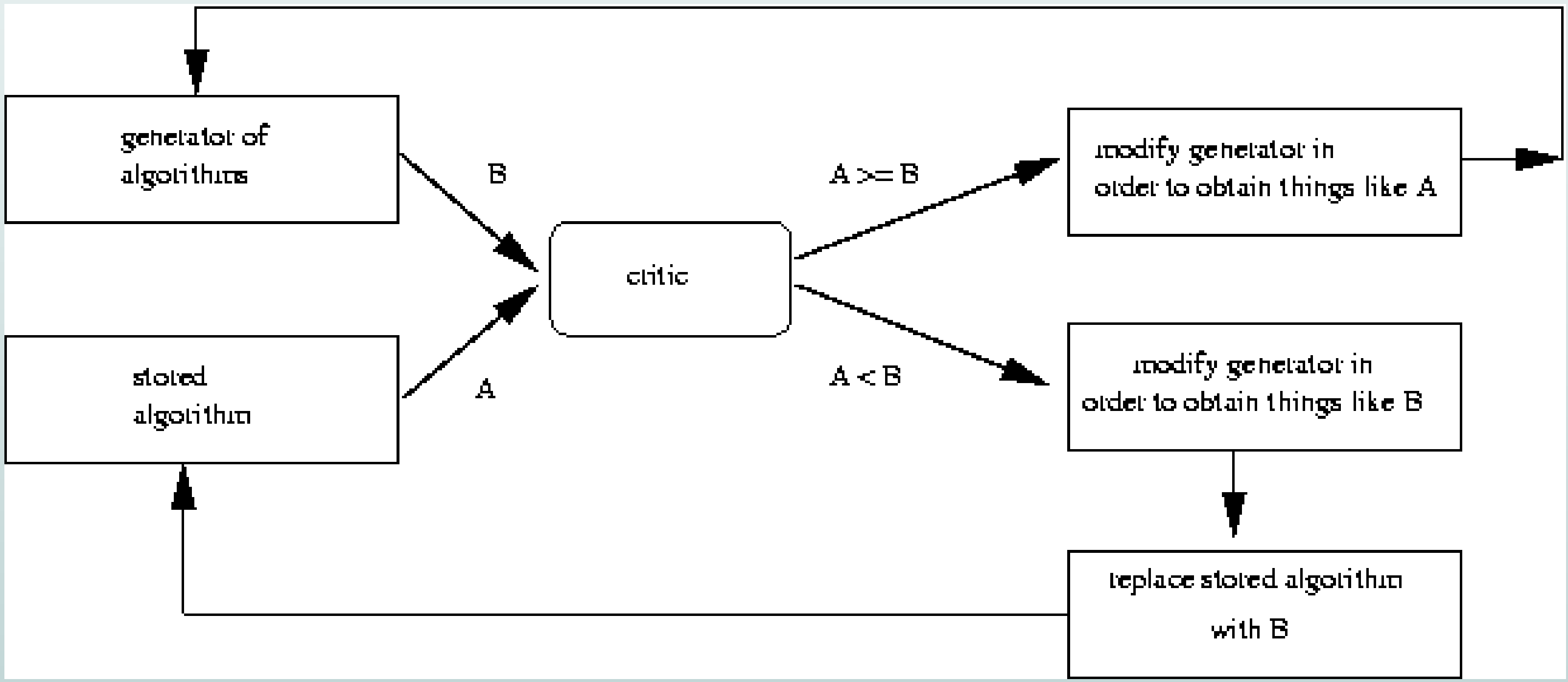
- A branch of artificial intelligence which uses common-sense rules drawn from experience to solve problems

- Self learning programs that get better with experience

- Programs that include ill-defined parameters to be tuned up (my own irreverent definition of heuristic programming)

Heuristic programming according to Minsky

Heuristic programming according to Minsky



How to evaluate a heuristic program

How to evaluate a heuristic program

- We can't deduce from the program structure any information about its running time

How to evaluate a heuristic program

- We can't deduce from the program structure any information about its running time

- We have to rely on a set of benchmarks to evaluate the relative ”goodness” of different heuristic programs that perform the same task

How to evaluate a heuristic program

- We can't deduce from the program structure any information about its running time

- We have to rely on a set of benchmarks to evaluate the relative ”goodness” of different heuristic programs that perform the same task

- Before using benchmarks, we must specify the criteria according to which different heuristic programs will be compared

Heuristic programming in Linux

Algorithmic programming in Linux

Algorithmic programming in Linux

- Linux algorithmic programs make a wide use of hash tables, lists, and trees to handle resource descriptors

Algorithmic programming in Linux

- Linux algorithmic programs make a wide use of hash tables, lists, and trees to handle resource descriptors
- In some cases, sophisticated search trees are used to get better performances:

Algorithmic programming in Linux

- Linux algorithmic programs make a wide use of hash tables, lists, and trees to handle resource descriptors

- In some cases, sophisticated search trees are used to get better performances:
  - Red-black trees are used to sort the memory region descriptors of a process

Algorithmic programming in Linux

- Linux algorithmic programs make a wide use of hash tables, lists, and trees to handle resource descriptors

- In some cases, sophisticated search trees are used to get better performances:
  - Red-black trees are used to sort the memory region descriptors of a process
  - Priority search trees (based on the radix priority search tree proposed by Edward M. McCreight in SIAM Journal of Computing, vol. 14, no.2, pages 257-276, May 1985) are used to locate quickly all the memory regions that refer to the same page frame

Examples of tasks fullfilled by algorithmic programs

Examples of tasks fullfilled by algorithmic programs

- Page fault handling: checking whether a logical address issued by a process belongs to the process's address space

Examples of tasks fullfilled by algorithmic programs

- Page fault handling: checking whether a logical address issued by a process belongs to the process's address space
- Managing free memory areas: buddy system algorithm (page frames), slab allocator (memory areas of arbitrary sizes)

Examples of tasks fullfilled by algorithmic programs

- Page fault handling: checking whether a logical address issued by a process belongs to the process's address space
- Managing free memory areas: buddy system algorithm (page frames), slab allocator (memory areas of arbitrary sizes)
- Pathname lookup: a cache is used to retrieve quickly the inode corresponding to a file's pathname

Examples of tasks fullfilled by algorithmic programs

- Page fault handling: checking whether a logical address issued by a process belongs to the process's address space
- Managing free memory areas: buddy system algorithm (page frames), slab allocator (memory areas of arbitrary sizes)
- Pathname lookup: a cache is used to retrieve quickly the inode corresponding to a file's pathname
- Checking for expired dynamic timers: a clever data structure allows timers to percolate among up to 5 different lists

Examples of tasks fullfilled by heuristic programs

Examples of tasks fullfilled by heuristic programs

- CPU scheduling: selecting the best process to run on a CPU

Examples of tasks fullfilled by heuristic programs

- CPU scheduling: selecting the best process to run on a CPU
- Multiprocessor scheduling: selecting the best CPU that should run a given process

Examples of tasks fullfilled by heuristic programs

- CPU scheduling: selecting the best process to run on a CPU
- Multiprocessor scheduling: selecting the best CPU that should run a given process
- I/O scheduling (elevators): inserting a new block device request in a dispatch queue in the best position

Examples of tasks fullfilled by heuristic programs

- CPU scheduling: selecting the best process to run on a CPU
- Multiprocessor scheduling: selecting the best CPU that should run a given process
- I/O scheduling (elevators): inserting a new block device request in a dispatch queue in the best position
- IRQ scheduling: selecting the best CPU that should handle an I/O interrupt

Examples of tasks fullfilled by heuristic programs

- CPU scheduling: selecting the best process to run on a CPU
- Multiprocessor scheduling: selecting the best CPU that should run a given process
- I/O scheduling (elevators): inserting a new block device request in a dispatch queue in the best position
- IRQ scheduling: selecting the best CPU that should handle an I/O interrupt
- Read-ahead: deciding how many pages of a sequentially processed file should be read in advance

Examples of tasks fullfilled by heuristic programs

- CPU scheduling: selecting the best process to run on a CPU
- Multiprocessor scheduling: selecting the best CPU that should run a given process
- I/O scheduling (elevators): inserting a new block device request in a dispatch queue in the best position
- IRQ scheduling: selecting the best CPU that should handle an I/O interrupt
- Read-ahead: deciding how many pages of a sequentially processed file should be read in advance
- Page frame reclaiming: selecting the most suitable page frame to be freed

Heuristic programming is crucial

Heuristic programming is crucial

- Even if some of us don't like heuristic programming, we have no alternative solutions

Heuristic programming is crucial

- Even if some of us don't like heuristic programming, we have no alternative solutions
- It is impossible to implement an efficient multitasking kernel without making use of heuristic programming

Heuristic programming is crucial

- Even if some of us don't like heuristic programming, we have no alternative solutions
- It is impossible to implement an efficient multitasking kernel without making use of heuristic programming
- No matter how accurate are the benchmarks, they cannot cover all possible user requirements

Heuristic programming is crucial

- Even if some of us don't like heuristic programming, we have no alternative solutions
- It is impossible to implement an efficient multitasking kernel without making use of heuristic programming
- No matter how accurate are the benchmarks, they cannot cover all possible user requirements
- This leads to strange situations such as: I switched from the older version X of a kernel to the newer version Y of the same kernel but my application is running slower on the newer kernel

Page frame reclaiming

The need for page frame reclaiming (1)

The need for page frame reclaiming (1)

- Linux, like all modern kernels, uses a page cache containing pages of data read from disk (this is done to reduce the number of disk accesses)

The need for page frame reclaiming (1)

- Linux, like all modern kernels, uses a page cache containing pages of data read from disk (this is done to reduce the number of disk accesses)

- A page of data is kept in the cache, even when no process is using it, simply because there is a chance it might be re-used by some other process in the future

The need for page frame reclaiming (1)

- Linux, like all modern kernels, uses a page cache containing pages of data read from disk (this is done to reduce the number of disk accesses)

- A page of data is kept in the cache, even when no process is using it, simply because there is a chance it might be re-used by some other process in the future

- As a further optimization, pages in the page cache that are “Dirty” are written to disk a few seconds after they have been modified (deferred writings)

The need for page frame reclaiming (2)

The need for page frame reclaiming (2)

- Running processes acquire more and more page frames to store pages of data and code (demand paging)

The need for page frame reclaiming (2)

- Running processes acquire more and more page frames to store pages of data and code (demand paging)

- A page requested by a process is first read from disk and stored in the page cache, and then copied from Kernel space to the process address space

The need for page frame reclaiming (3)

The need for page frame reclaiming (3)

- There are two main consumers of free page frames:

The need for page frame reclaiming (3)

- There are two main consumers of free page frames:
  - The page cache system

The need for page frame reclaiming (3)

- There are two main consumers of free page frames:
  - The page cache system
  - The running processes

The need for page frame reclaiming (3)

- There are two main consumers of free page frames:
  - The page cache system
  - The running processes

- The page cache system never releases page frames

The need for page frame reclaiming (3)

- There are two main consumers of free page frames:
  - The page cache system
  - The running processes

- The page cache system never releases page frames

- Some processes release page frames (e.g. `execve()`), others keep all their page frames until they terminate

The need for page frame reclaiming (3)

- There are two main consumers of free page frames:
  - The page cache system
  - The running processes

- The page cache system never releases page frames

- Some processes release page frames (e.g. `execve()`), others keep all their page frames until they terminate

- Sooner or later, all the available RAM is assigned to the page cache and to the processes and page frame reclaiming must be performed

When is page frame reclaiming performed?

When is page frame reclaiming performed?

- When a kernel function requires new page frames and there is no free memory left: direct reclaiming

When is page frame reclaiming performed?

- When a kernel function requires new page frames and there is no free memory left: direct reclaiming
- When some kernel thread is periodically activated (once every few seconds): indirect reclaiming

Classifying pages

Classifying pages

- Unreclaimable: free pages, reserved pages, pages dynamically allocated to the kernel, temporarily locked pages, memory locked pages

Classifying pages

- Unreclaimable: free pages, reserved pages, pages dynamically allocated to the kernel, temporarily locked pages, memory locked pages
- Swappable: anonymous pages in User Mode address spaces, mapped pages of IPC shared memory

Classifying pages

- Unreclaimable: free pages, reserved pages, pages dynamically allocated to the kernel, temporarily locked pages, memory locked pages
- Swappable: anonymous pages in User Mode address spaces, mapped pages of IPC shared memory
- Syncable: pages in the page cache containing data of disk files, pages in the page cache containing copies of disk data blocks

Classifying pages

- Unreclaimable: free pages, reserved pages, pages dynamically allocated to the kernel, temporarily locked pages, memory locked pages
- Swappable: anonymous pages in User Mode address spaces, mapped pages of IPC shared memory
- Syncable: pages in the page cache containing data of disk files, pages in the page cache containing copies of disk data blocks
- Discardable: unused pages in the slab cache or in some other memory kernel cache

Page frame reclaiming: easy case

Page frame reclaiming: easy case

- Suppose we have a light memory load

Page frame reclaiming: easy case

- Suppose we have a light memory load

- In this case, page frame reclaiming consists simply of cleaning up the page cache (and other caches) from time to time

Page frame reclaiming: hard case

Page frame reclaiming: hard case

- Suppose we have a heavy memory load

Page frame reclaiming: hard case

- Suppose we have a heavy memory load
- If page frame reclaiming is done unproperly, some running processes may start to do disk thrashing: the PFRA steals pages from them and they have to re-acquire them

Page frame reclaiming: hard case

- Suppose we have a heavy memory load
- If page frame reclaiming is done unproperly, some running processes may start to do disk thrashing: the PFRA steals pages from them and they have to re-acquire them
- Critical problem for enterprise systems with heavy peak loads

Page frame reclaiming (PFRA) in Linux 2.6 (1)

Page frame reclaiming (PFRA) in Linux 2.6 (1)

- Free the easy pages first: “Uptodate” pages are better candidates than “Dirty” pages because they don’t have to be written to disk

Page frame reclaiming (PFRA) in Linux 2.6 (1)

- Free the easy pages first: “Uptodate” pages are better candidates than “Dirty” pages because they don’t have to be written to disk
- Make all pages of a User Mode process reclaimable

Page frame reclaiming (PFRA) in Linux 2.6 (1)

- Free the easy pages first: “Uptodate” pages are better candidates than “Dirty” pages because they don’t have to be written to disk
- Make all pages of a User Mode process reclaimable
- Reclaim all pages that refer to the same page frame at once (“reverse mapping” is used to discover page frames shared by several processes)

Page frame reclaiming (PFRA) in Linux 2.6 (1)

- Free the easy pages first: “Uptodate” pages are better candidates than “Dirty” pages because they don’t have to be written to disk
- Make all pages of a User Mode process reclaimable
- Reclaim all pages that refer to the same page frame at once (“reverse mapping” is used to discover page frames shared by several processes)
- Order pages according to age: least recently used pages should be freed before pages accessed recently (Least Recently Used heuristics)

Page frame reclaiming (PFRA) in Linux 2.6 (1)

- Free the easy pages first: “Uptodate” pages are better candidates than “Dirty” pages because they don’t have to be written to disk
- Make all pages of a User Mode process reclaimable
- Reclaim all pages that refer to the same page frame at once (“reverse mapping” is used to discover page frames shared by several processes)
- Order pages according to age: least recently used pages should be freed before pages accessed recently (Least Recently Used heuristics)
- Reduce the danger of disk thrashing by making use of a “swap token” (see later)

Page frame reclaiming in Linux 2.6 (2)

Page frame reclaiming in Linux 2.6 (2)

- A simple heuristics is used to rank all reclaimable pages according to how recently they have been addressed. The pages are inserted in one of the following two lists:

Page frame reclaiming in Linux 2.6 (2)

- A simple heuristics is used to rank all reclaimable pages according to how recently they have been addressed. The pages are inserted in one of the following two lists:
  - Inactive list: pages that have not been accessed for a long time

Page frame reclaiming in Linux 2.6 (2)

- A simple heuristics is used to rank all reclaimable pages according to how recently they have been addressed. The pages are inserted in one of the following two lists:
  - Inactive list: pages that have not been accessed for a long time
  - Active list: pages referenced recently

Page frame reclaiming in Linux 2.6 (2)

- A simple heuristics is used to rank all reclaimable pages according to how recently they have been addressed. The pages are inserted in one of the following two lists:
  - Inactive list: pages that have not been accessed for a long time
  - Active list: pages referenced recently
  - the PFRA moves pages from one list to the other according to their usage and to the amount of free memory

Balancing the two lists

Balancing the two lists

- New pages are added to the head of the inactive list

Balancing the two lists

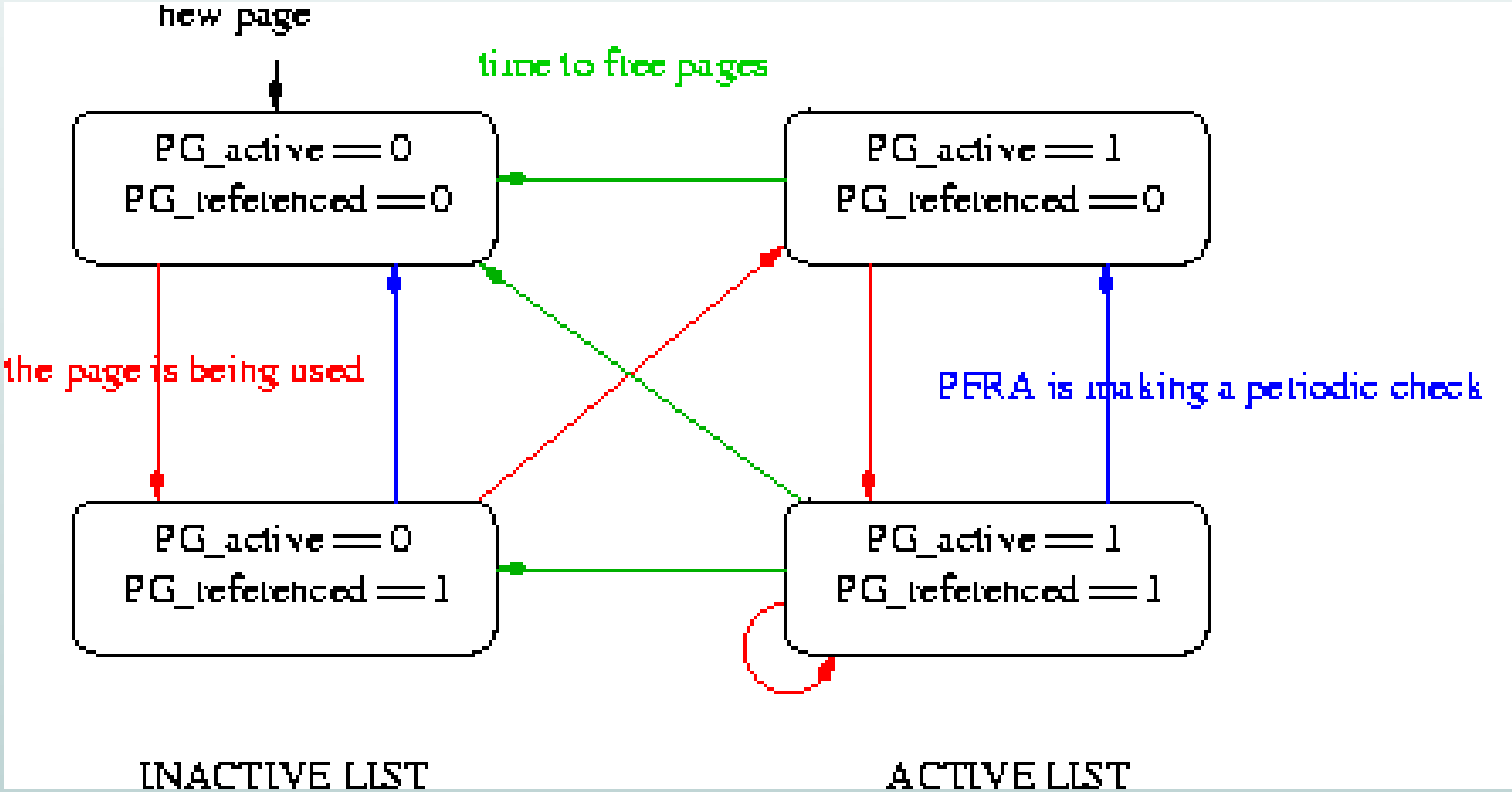
- New pages are added to the head of the inactive list
- Referenced pages are moved from the inactive to the active list

Balancing the two lists

- New pages are added to the head of the inactive list
- Referenced pages are moved from the inactive to the active list
- When low on memory, refile pages from the tail of the active list to the head of the inactive list and start freeing pages from the tail of the inactive list

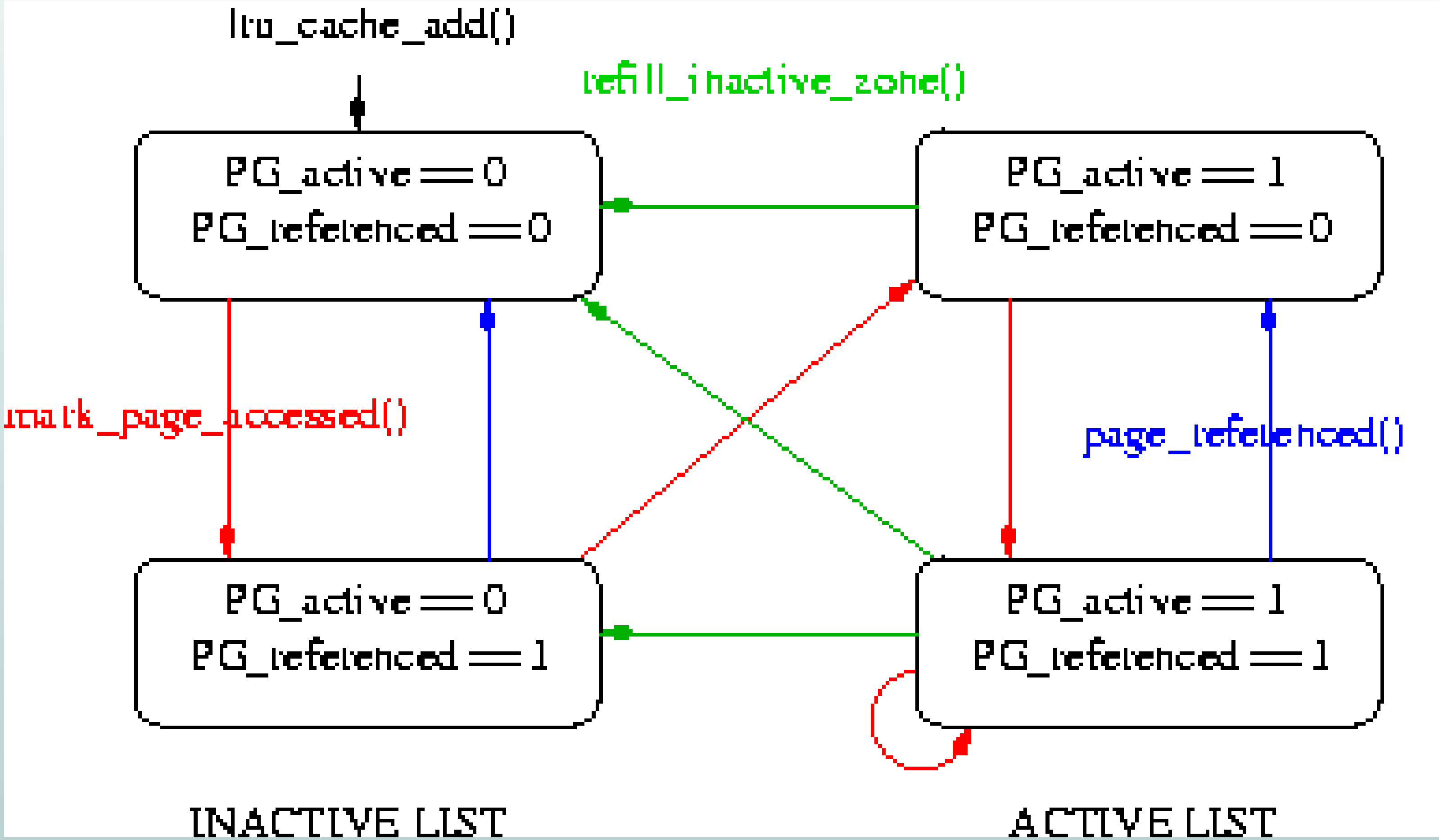
Moving pages across the LRU lists (1)

Moving pages across the LRU lists (1)



Moving pages across the LRU lists (2)

Moving pages across the LRU lists (2)



The refill_inactive_zone() function

The refill_inactive_zone() function

- This function moves pages from the active to the inactive list and makes them eligible for page frame reclaiming

The refill_inactive_zone() function

- This function moves pages from the active to the inactive list and makes them eligible for page frame reclaiming

- Pages belonging to process address spaces cannot be reclaimed as long as they stay in the active list

The refill_inactive_zone() function

- This function moves pages from the active to the inactive list and makes them eligible for page frame reclaiming
- Pages belonging to process address spaces cannot be reclaimed as long as they stay in the active list
- Moving pages from the active to the inactive list means make them eligible for page frame reclaiming

The refill_inactive_zone() function

- This function moves pages from the active to the inactive list and makes them eligible for page frame reclaiming
- Pages belonging to process address spaces cannot be reclaimed as long as they stay in the active list
- Moving pages from the active to the inactive list means make them eligible for page frame reclaiming
- A crucial heuristic parameter called `swap_tendency` is used to decide how much `refill_inactive_zone()` can shrink the active list (see later)

The mark_page_accessed() function

The mark_page_accessed() function

- This function, which marks a page as accessed, is invoked when:

The mark_page_accessed() function

- This function, which marks a page as accessed, is invoked when:
  - Reading a page of data from a file

The mark_page_accessed() function

- This function, which marks a page as accessed, is invoked when:
  - Reading a page of data from a file
  - Loading an anonymous page for a process

The mark_page_accessed() function

- This function, which marks a page as accessed, is invoked when:
  - Reading a page of data from a file
  - Loading an anonymous page for a process
  - Loading a page of an IPC shared memory region

The mark_page_accessed() function

- This function, which marks a page as accessed, is invoked when:
  - Reading a page of data from a file
  - Loading an anonymous page for a process
  - Loading a page of an IPC shared memory region
  - Performing automatic swap in on a swapped out page

The page_referenced() function

The page_referenced() function

- This function is invoked once for every page scanned by the PFRA

The page_referenced() function

- This function is invoked once for every page scanned by the PFRA

- If the `PG_referenced` flag of a given page is set, `page_referenced()` downgrades the status of that page by clearing the flag (see figure)

The page_referenced() function

- This function is invoked once for every page scanned by the PFRA
- If the `PG_referenced` flag of a given page is set, `page_referenced()` down-grades the status of that page by clearing the flag (see figure)
- In this way, pages that have not beeing accessed for a long time move from the active to the inacive list

The swap_tendency factor (1)

The swap_tendency factor (1)

- The swap tendency heuristic parameter specifies a threshold on the amount of pages that can be stolen from processes (under light load, pages should be stolen only from the caches)

The swap_tendency factor (1)

- The swap tendency heuristic parameter specifies a threshold on the amount of pages that can be stolen from processes (under light load, pages should be stolen only from the caches)
- It is computed as:

```
swap_tendency = mapped_ratio/2 + distress + vm_swappiness
```

The swap_tendency factor (1)

- The swap tendency heuristic parameter specifies a threshold on the amount of pages that can be stolen from processes (under light load, pages should be stolen only from the caches)
- It is computed as:

```
swap_tendency = mapped_ratio/2 + distress + vm_swappiness
```

- `distress`, which ranges between 0 and 100, is tied to the amount of list scanning needed to reach the target (0= no trouble, 100= great trouble)

The swap_tendency factor (1)

- The swap tendency heuristic parameter specifies a threshold on the amount of pages that can be stolen from processes (under light load, pages should be stolen only from the caches)
- It is computed as:

```
swap_tendency = mapped_ratio/2 + distress + vm_swappiness
```

- `distress`, which ranges between 0 and 100, is tied to the amount of list scanning needed to reach the target (0= no trouble, 100= great trouble)
- `vm_swappiness` is set to 60 when swapping is enabled (at least one swap area is active)

The swap_tendency factor (2)

The swap_tendency factor (2)

- `mapped_ratio` is the percentage of pages belonging to memory mapped files with respect to the total number of pages

The swap_tendency factor (2)

- `mapped_ratio` is the percentage of pages belonging to memory mapped files with respect to the total number of pages

- Pages are reclaimed from the address space of processes only if `swap_tendency` is greater than 99

The swap_tendency factor (2)

- `mapped_ratio` is the percentage of pages belonging to memory mapped files with respect to the total number of pages
- Pages are reclaimed from the address space of processes only if `swap_tendency` is greater than 99
- If we have a light memory load, for instance `distress` = 0, `vm_swappiness` = 60, `swap_tendency` =10, then we'll have:

The swap_tendency factor (2)

- `mapped_ratio` is the percentage of pages belonging to memory mapped files with respect to the total number of pages
- Pages are reclaimed from the address space of processes only if `swap_tendency` is greater than 99
- If we have a light memory load, for instance `distress` = 0, `vm_swappiness` = 60, `swap_tendency` =10, then we'll have:
- `swap_tendency` = 65 and pages will continue to be reclaimed only from the caches

The swap token (1)

The swap token (1)

- New heuristic added recently to the PFRA

The swap token (1)

- New heuristic added recently to the PFRA

- We want to avoid stealing pages to processes that are obtaining new pages at a very fast rate

The swap token (1)

- New heuristic added recently to the PFRA
- We want to avoid stealing pages to processes that are obtaining new pages at a very fast rate
- Applicable to one process at a time, the holder of the swap token

The swap token (1)

- New heuristic added recently to the PFRA
- We want to avoid stealing pages to processes that are obtaining new pages at a very fast rate
- Applicable to one process at a time, the holder of the swap token
- The page acquisition rate is measured by the number of page faults issued by the process

The swap token (2)

The swap token (2)

- A process tries to get the swap token when it does a page fault while looking for a page stored in a file or in a swap area

The swap token (2)

- A process tries to get the swap token when it does a page fault while looking for a page stored in a file or in a swap area
- It effectively gets the swap token if the current holder did not make page faults for some time, or if it held the token for too long

The swap token (2)

- A process tries to get the swap token when it does a page fault while looking for a page stored in a file or in a swap area
- It effectively gets the swap token if the current holder did not make page faults for some time, or if it held the token for too long
- This recent change to the PFRA introduces two additional heuristic constants!

Conclusions?

Conclusions?

- Heuristic programming is easy: every reasonably skilful programmer can invent a new heuristic algorithm and claim it performs well in some cases

Conclusions?

- Heuristic programming is easy: every reasonably skilful programmer can invent a new heuristic algorithm and claim it performs well in some cases

- The amount of heuristic programming included in the kernel can be reduced only by developing suitable models of kernel interactions (e.g. active/inactive lists for identifying LRU pages)

Conclusions?

- Heuristic programming is easy: every reasonably skilful programmer can invent a new heuristic algorithm and claim it performs well in some cases

- The amount of heuristic programming included in the kernel can be reduced only by developing suitable models of kernel interactions (e.g. active/inactive lists for identifying LRU pages)

- Homework: invent your own read-ahead heuristic algorithm and compare it with the official one used by Linux