操作系统原理

第十章：虚拟内存

洪明坚

重庆大学软件学院

February 19, 2016

目录

Outline

Background(1/2)

Background(1/2)

- IBM's poster explaining virtual memory in 1978:

Background(1/2)

- IBM's poster explaining virtual memory in 1978:
  - If it's there and you can see it, it's real;

Background(1/2)

- IBM's poster explaining virtual memory in 1978:
  - If it's there and you can see it, it's real;
  - If it's there and you can't see it,

Background(1/2)

- IBM's poster explaining virtual memory in 1978:
  - If it's there and you can see it, it's real;
  - If it's there and you can't see it, it's transparent;

Background(1/2)

- IBM's poster explaining virtual memory in 1978:
  - If it's there and you can see it, it's real;
  - If it's there and you can't see it, it's transparent;
  - If it's not there and you can see it,

Background(1/2)

- IBM's poster explaining virtual memory in 1978:
  - If it's there and you can see it, it's real;
  - If it's there and you can't see it, it's transparent;
  - If it's not there and you can see it, it's virtual;

Background(1/2)

- IBM's poster explaining virtual memory in 1978:
  - If it's there and you can see it, it's real;
  - If it's there and you can't see it, it's transparent;
  - If it's not there and you can see it, it's virtual;
  - If it's not there and you can't see it,

Background(1/2)

- IBM's poster explaining virtual memory in 1978:
  - If it's there and you can see it, it's real;
  - If it's there and you can't see it, it's transparent;
  - If it's not there and you can see it, it's virtual;
  - If it's not there and you can't see it, you erased it!

Background(2/2)

Background(2/2)

- 自从有了计算机以来，我们就希望能够方便、高效地使用（今天仍然）十分有限的内存：

Background(2/2)

- 自从有了计算机以来，我们就希望能够方便、高效地使用（今天仍然）十分有限的内存：
  - 用户总是可以认为自己计算机的内存大到足够运行任何程序；

Background(2/2)

- 自从有了计算机以来，我们就希望能够方便、高效地使用（今天仍然）十分有限的内存：
  - 用户总是可以认为自己计算机的内存大到足够运行任何程序；
  - 程序员可以认为自己的程序将运行在巨大、连续的内存中；

Background(2/2)

- 自从有了计算机以来，我们就希望能够方便、高效地使用（今天仍然）十分有限的内存：
  - 用户总是可以认为自己计算机的内存大到足够运行任何程序；
  - 程序员可以认为自己的程序将运行在巨大、连续的内存中；
    - 不需要用overlay或swap等技术来自己管理内存。

Background(2/2)

- 自从有了计算机以来，我们就希望能够方便、高效地使用（今天仍然）十分有限的内存：
  - 用户总是可以认为自己计算机的内存大到足够运行任何程序；
  - 程序员可以认为自己的程序将运行在巨大、连续的内存中；
    - 不需要用overlay或swap等技术来自己管理内存。
  - 进程在运行过程中不能访问（包括读写）其他进程的数据；

Background(2/2)

- 自从有了计算机以来，我们就希望能够方便、高效地使用（今天仍然）十分有限的内存：
  - 用户总是可以认为自己计算机的内存大到足够运行任何程序；
  - 程序员可以认为自己的程序将运行在巨大、连续的内存中；
    - 不需要用overlay或swap等技术来自己管理内存。
  - 进程在运行过程中不能访问（包括读写）其他进程的数据；
    - 更不能访问属于操作系统的数据；

Background(2/2)

- 自从有了计算机以来，我们就希望能够方便、高效地使用（今天仍然）十分有限的内存：
  - 用户总是可以认为自己计算机的内存大到足够运行任何程序；
  - 程序员可以认为自己的程序将运行在巨大、连续的内存中；
    - 不需要用overlay或swap等技术来自己管理内存。
  - 进程在运行过程中不能访问（包括读写）其他进程的数据；
    - 更不能访问属于操作系统的数据；
    - 而且某一个进程引起的问题不会波及到操作系统或其他进程。

Background(2/2)

- 自从有了计算机以来，我们就希望能够方便、高效地使用（今天仍然）十分有限的内存：
  - 用户总是可以认为自己计算机的内存大到足够运行任何程序；
  - 程序员可以认为自己的程序将运行在巨大、连续的内存中；
    - 不需要用overlay或swap等技术来自己管理内存。
  - 进程在运行过程中不能访问（包括读写）其他进程的数据；
    - 更不能访问属于操作系统的数据；
    - 而且某一个进程引起的问题不会波及到操作系统或其他进程。
  - 由于内存有限，应该尽可能地共享一些公用的代码和数据。

Background(2/2)

- 自从有了计算机以来，我们就希望能够方便、高效地使用（今天仍然）十分有限的内存：
  - 用户总是可以认为自己计算机的内存大到足够运行任何程序；
  - 程序员可以认为自己的程序将运行在巨大、连续的内存中；
    - 不需要用overlay或swap等技术来自己管理内存。
  - 进程在运行过程中不能访问（包括读写）其他进程的数据；
    - 更不能访问属于操作系统的数据；
    - 而且某一个进程引起的问题不会波及到操作系统或其他进程。
  - 由于内存有限，应该尽可能地共享一些公用的代码和数据。
- 前面介绍的各种技术解决了上述问题的某一些方面，但没有提出一个整体的解决方案。

Background(2/2)

- 自从有了计算机以来，我们就希望能够方便、高效地使用（今天仍然）十分有限的内存：
  - 用户总是可以认为自己计算机的内存大到足够运行任何程序；
  - 程序员可以认为自己的程序将运行在巨大、连续的内存中；
    - 不需要用overlay或swap等技术来自己管理内存。
  - 进程在运行过程中不能访问（包括读写）其他进程的数据；
    - 更不能访问属于操作系统的数据；
    - 而且某一个进程引起的问题不会波及到操作系统或其他进程。
  - 由于内存有限，应该尽可能地共享一些公用的代码和数据。
- 前面介绍的各种技术解决了上述问题的某一些方面，但没有提出一个整体的解决方案。
  - 虚拟内存技术基于已有的各种技术给出了一个完整的解决方案，是20世纪计算机技术最重要的发明之一。

What's virtual memory?(1/2)

What's virtual memory?(1/2)

- **Virtual memory** is the separation of user logical memory from physical memory.

What's virtual memory?(1/2)

- **Virtual memory** is the separation of user logical memory from physical memory.
  - Provides a huge, continuous and private logical memory, which may be much larger than the physical memory, to **each** process.

What's virtual memory?(1/2)

- **Virtual memory** is the separation of user logical memory from physical memory.
  - Provides a huge, continuous and private logical memory, which may be much larger than the physical memory, to **each** process.
    - Only part of the program needs to be in memory for execution.

What's virtual memory?(1/2)

- **Virtual memory** is the separation of user logical memory from physical memory.
  - Provides a huge, continuous and private logical memory, which may be much larger than the physical memory, to **each** process.
    - Only part of the program needs to be in memory for execution.
  - Allows address spaces to be shared by several processes.

What's virtual memory?(1/2)

- **Virtual memory** is the separation of user logical memory from physical memory.
  - Provides a huge, continuous and private logical memory, which may be much larger than the physical memory, to **each** process.
    - Only part of the program needs to be in memory for execution.
  - Allows address spaces to be shared by several processes.
    - More efficient process creation.

What's virtual memory?(1/2)

- **Virtual memory** is the separation of user logical memory from physical memory.
  - Provides a huge, continuous and private logical memory, which may be much larger than the physical memory, to **each** process.
    - Only part of the program needs to be in memory for execution.
  - Allows address spaces to be shared by several processes.
    - More efficient process creation.
- Virtual memory can be implemented via:

What's virtual memory?(1/2)

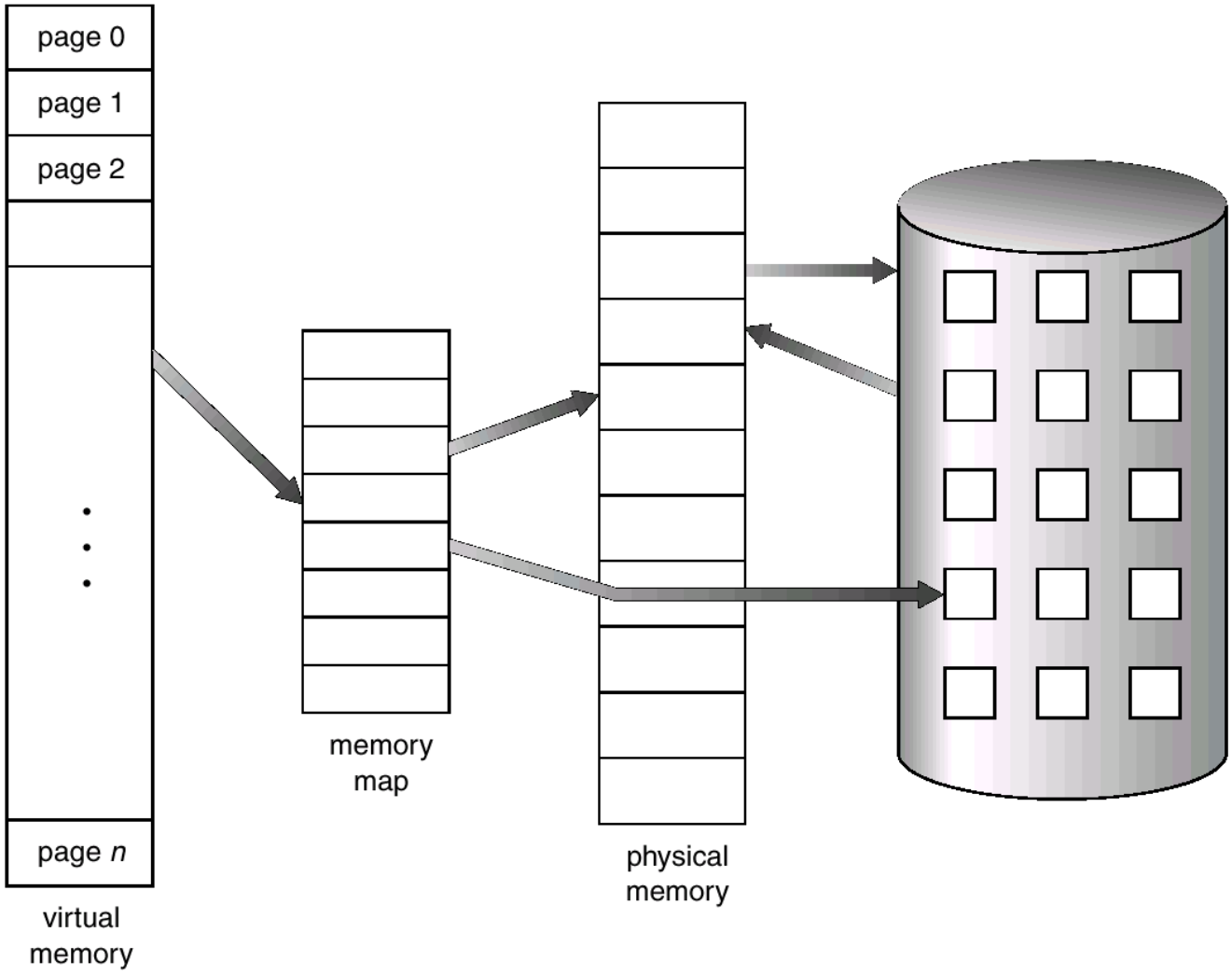
- **Virtual memory** is the separation of user logical memory from physical memory.
  - Provides a huge, continuous and private logical memory, which may be much larger than the physical memory, to **each** process.
    - Only part of the program needs to be in memory for execution.
  - Allows address spaces to be shared by several processes.
    - More efficient process creation.
- Virtual memory can be implemented via:
  - Demand paging

What's virtual memory?(1/2)

- **Virtual memory** is the separation of user logical memory from physical memory.
  - Provides a huge, continuous and private logical memory, which may be much larger than the physical memory, to **each** process.
    - Only part of the program needs to be in memory for execution.
  - Allows address spaces to be shared by several processes.
    - More efficient process creation.
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

What's virtual memory(2/2)

What's virtual memory(2/2)



Outline

Demand paging

Demand paging

- Bring a page into memory only when it is needed, i.e., **lazy swapper**.

Demand paging

- Bring a page into memory only when it is needed, i.e., **lazy swapper**.
  - Less I/O needed;

Demand paging

- Bring a page into memory only when it is needed, i.e., **lazy swapper**.
  - Less I/O needed;
  - Less memory needed;

Demand paging

- Bring a page into memory only when it is needed, i.e., **lazy swapper**.
  - Less I/O needed;
  - Less memory needed;
  - Faster response;

Demand paging

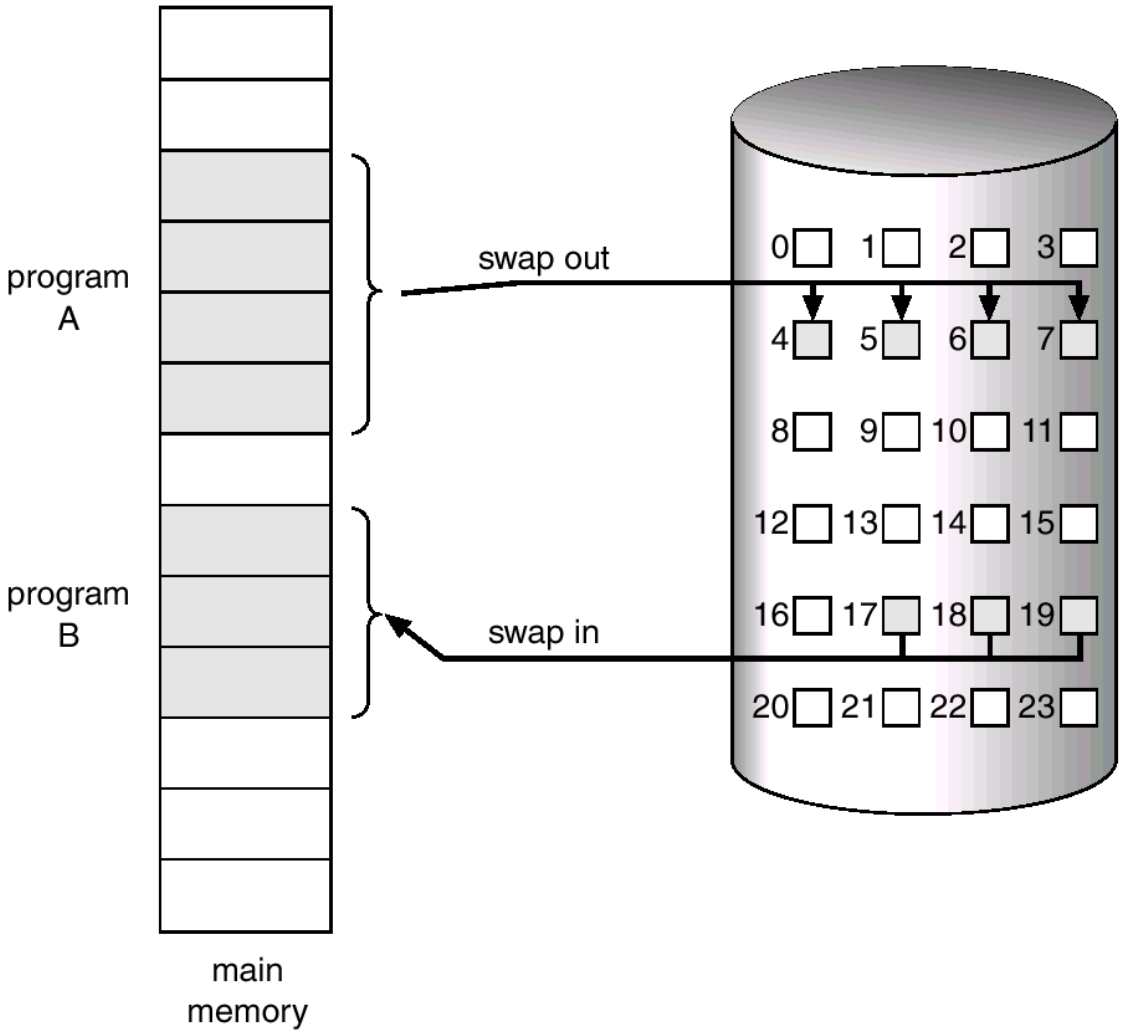
- Bring a page into memory only when it is needed, i.e., **lazy swapper**.
  - Less I/O needed;
  - Less memory needed;
  - Faster response;
  - More processes;

Demand paging

- Bring a page into memory only when it is needed, i.e., **lazy swapper**.
  - Less I/O needed;
  - Less memory needed;
  - Faster response;
  - More processes;
- 一般情况下，我们用swapper表示整个进程的交换；而用pager来表示对页进行交换的lazy swapper。

Pager

Pager


program
A
program
B
swap out
swap in
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
main
memory

When to bring in a page?

When to bring in a page?

- 当操作系统调度某个进程运行时，如何判断该进程的页面是否已经被映射到内存中？

When to bring in a page?

- 当操作系统调度某个进程运行时，如何判断该进程的页面是否已经被映射到内存中？
- Remember: With each page table entry (PTE) a valid-invalid bit is associated:

When to bring in a page?

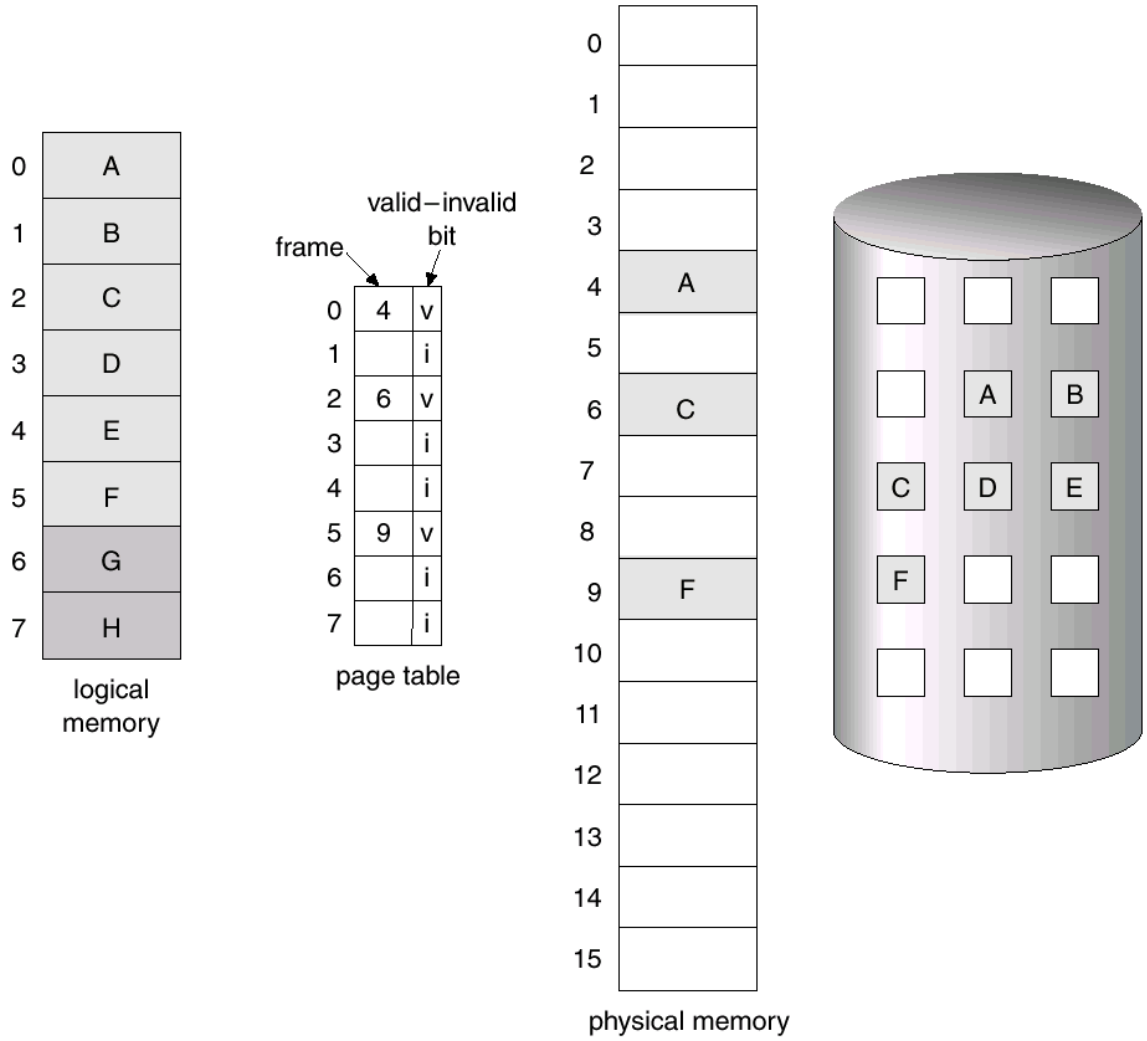
- 当操作系统调度某个进程运行时，如何判断该进程的页面是否已经被映射到内存中？
- Remember: With each page table entry (PTE) a valid-invalid bit is associated:
  - If the bit is set to "valid", it indicates that the associated page is both legal and in memory;

When to bring in a page?

- 当操作系统调度某个进程运行时，如何判断该进程的页面是否已经被映射到内存中？
- Remember: With each page table entry (PTE) a valid-invalid bit is associated:
  - If the bit is set to "valid", it indicates that the associated page is both legal and in memory;
  - Otherwise, it indicates the page is not legal (i.e., not in the logical address space of the process), or is legal but currently not in memory.

Valid-Invalid bit

Valid-Invalid bit



What happens if not in memory?

What happens if not in memory?

- During address translation in MMU, if valid-invalid bit in PTE is "invalid",

What happens if not in memory?

- During address translation in MMU, if valid-invalid bit in PTE is "invalid",
  - CPU triggers a **page-fault trap** into the OS,

What happens if not in memory?

- During address translation in MMU, if valid-invalid bit in PTE is "invalid",
  - CPU triggers a **page-fault trap** into the OS,
  - then OS looks at an internal table (usually kept in the PCB) to decide if it's

What happens if not in memory?

- During address translation in MMU, if valid-invalid bit in PTE is "invalid",
  - CPU triggers a **page-fault trap** into the OS,
  - then OS looks at an internal table (usually kept in the PCB) to decide if it's
    - an illegal reference, OS terminates the process;

What happens if not in memory?

- During address translation in MMU, if valid-invalid bit in PTE is "invalid",
  - CPU triggers a **page-fault trap** into the OS,
  - then OS looks at an internal table (usually kept in the PCB) to decide if it's
    - an illegal reference, OS terminates the process;
    - legal but not in memory, OS will bring it in.

Page-fault service routine(1/2)

Page-fault service routine(1/2)

1. We check an internal table for this process to determine whether the reference was a legal or illegal memory access;

Page-fault service routine(1/2)

1. We check an internal table for this process to determine whether the reference was a legal or illegal memory access;
2. If the reference was illegal, we terminate the process. If it was legal, but we have not yet brought in that page, we now page it in;

Page-fault service routine(1/2)

1. We check an internal table for this process to determine whether the reference was a legal or illegal memory access;
2. If the reference was illegal, we terminate the process. If it was legal, but we have not yet brought in that page, we now page it in;
3. We find a free frame;

Page-fault service routine(1/2)

1. We check an internal table for this process to determine whether the reference was a legal or illegal memory access;
2. If the reference was illegal, we terminate the process. If it was legal, but we have not yet brought in that page, we now page it in;
3. We find a free frame;
4. We schedule a disk operation to read the desired page into the newly allocated frame

Page-fault service routine(1/2)

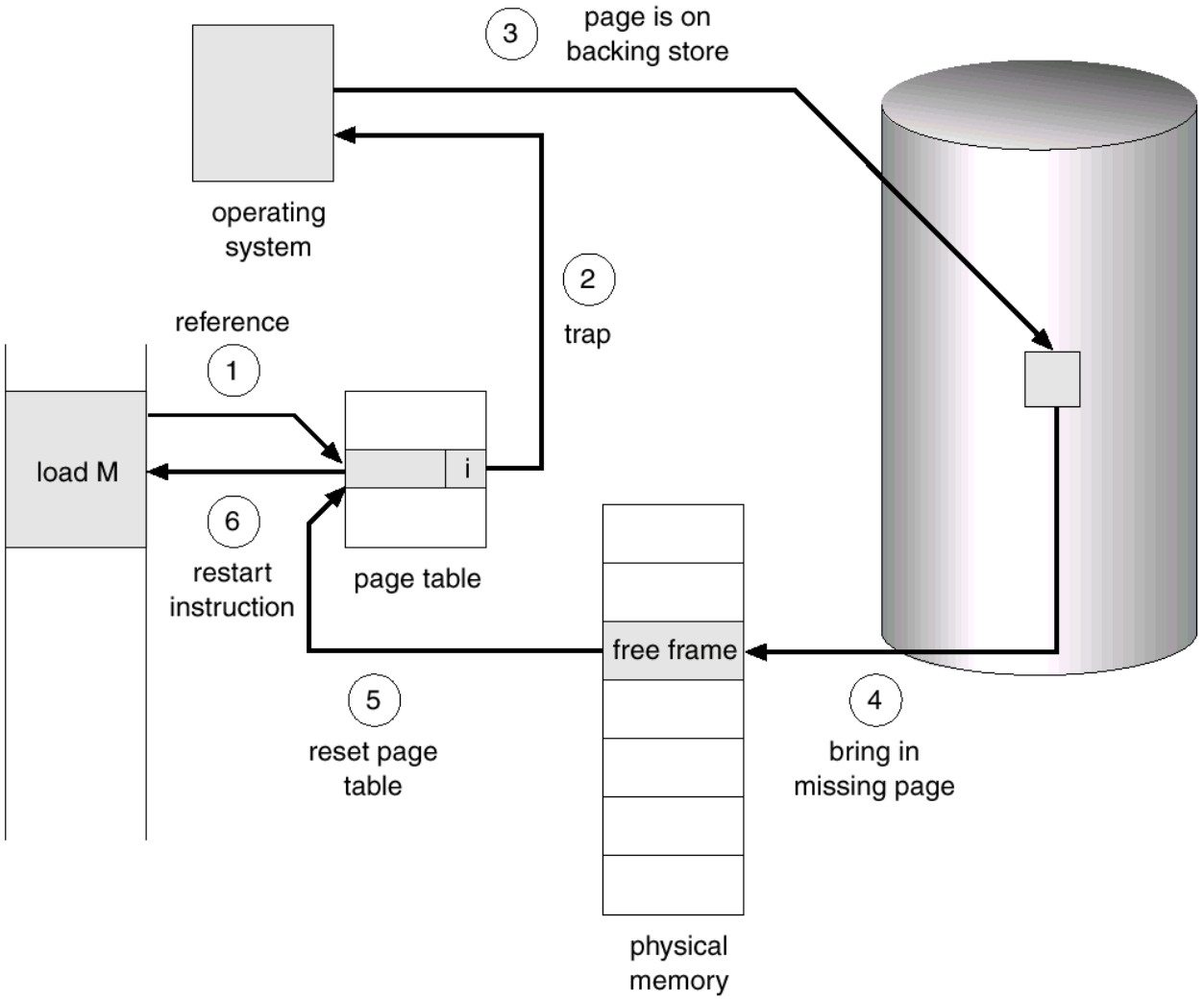
1. We check an internal table for this process to determine whether the reference was a legal or illegal memory access;
2. If the reference was illegal, we terminate the process. If it was legal, but we have not yet brought in that page, we now page it in;
3. We find a free frame;
4. We schedule a disk operation to read the desired page into the newly allocated frame
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory;

Page-fault service routine(1/2)

1. We check an internal table for this process to determine whether the reference was a legal or illegal memory access;
2. If the reference was illegal, we terminate the process. If it was legal, but we have not yet brought in that page, we now page it in;
3. We find a free frame;
4. We schedule a disk operation to read the desired page into the newly allocated frame
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory;
6. We restart the instruction that was interrupted by the page-fault trap. The process can now access the page as if it had always been in memory.

Page-fault service routine(2/2)

Page-fault service routine(2/2)



Architecture requirements

Architecture requirements

1. Page Table: This table has the ability to mark an entry invalid through a valid-invalid bit;

Architecture requirements

1. Page Table: This table has the ability to mark an entry invalid through a valid-invalid bit;
2. Secondary memory: This memory holds those pages that are not present in main memory. It's usually called **swap space**;

Architecture requirements

1. Page Table: This table has the ability to mark an entry invalid through a valid-invalid bit;
2. Secondary memory: This memory holds those pages that are not present in main memory. It's usually called **swap space**;
3. Capability to restart any instruction **exactly** after a page-fault.

Architecture requirements

1. Page Table: This table has the ability to mark an entry invalid through a valid-invalid bit;
2. Secondary memory: This memory holds those pages that are not present in main memory. It's usually called **swap space**;
3. Capability to restart any instruction **exactly** after a page-fault.
   - This is **NOT** easy sometime.

Questions

- Any questions?

What happens if no free frame?

What happens if no free frame?

- While the process executes and accesses pages that are in memory, execution proceeds normally.

What happens if no free frame?

- While the process executes and accesses pages that are in memory, execution proceeds normally.
- By increasing the degree of multiprogramming, we will **over-allocate** the physical memory. When page fault occurs, no free frames available. How to proceed?

What happens if no free frame?

- While the process executes and accesses pages that are in memory, execution proceeds normally.
- By increasing the degree of multiprogramming, we will **over-allocate** the physical memory. When page fault occurs, no free frames available. How to proceed?
- Find a frame in memory, but not actively in use, page it out.

What happens if no free frame?

- While the process executes and accesses pages that are in memory, execution proceeds normally.
- By increasing the degree of multiprogramming, we will **over-allocate** the physical memory. When page fault occurs, no free frames available. How to proceed?
- Find a frame in memory, but not actively in use, page it out.
- Here comes the **Page replacement**

What happens if no free frame?

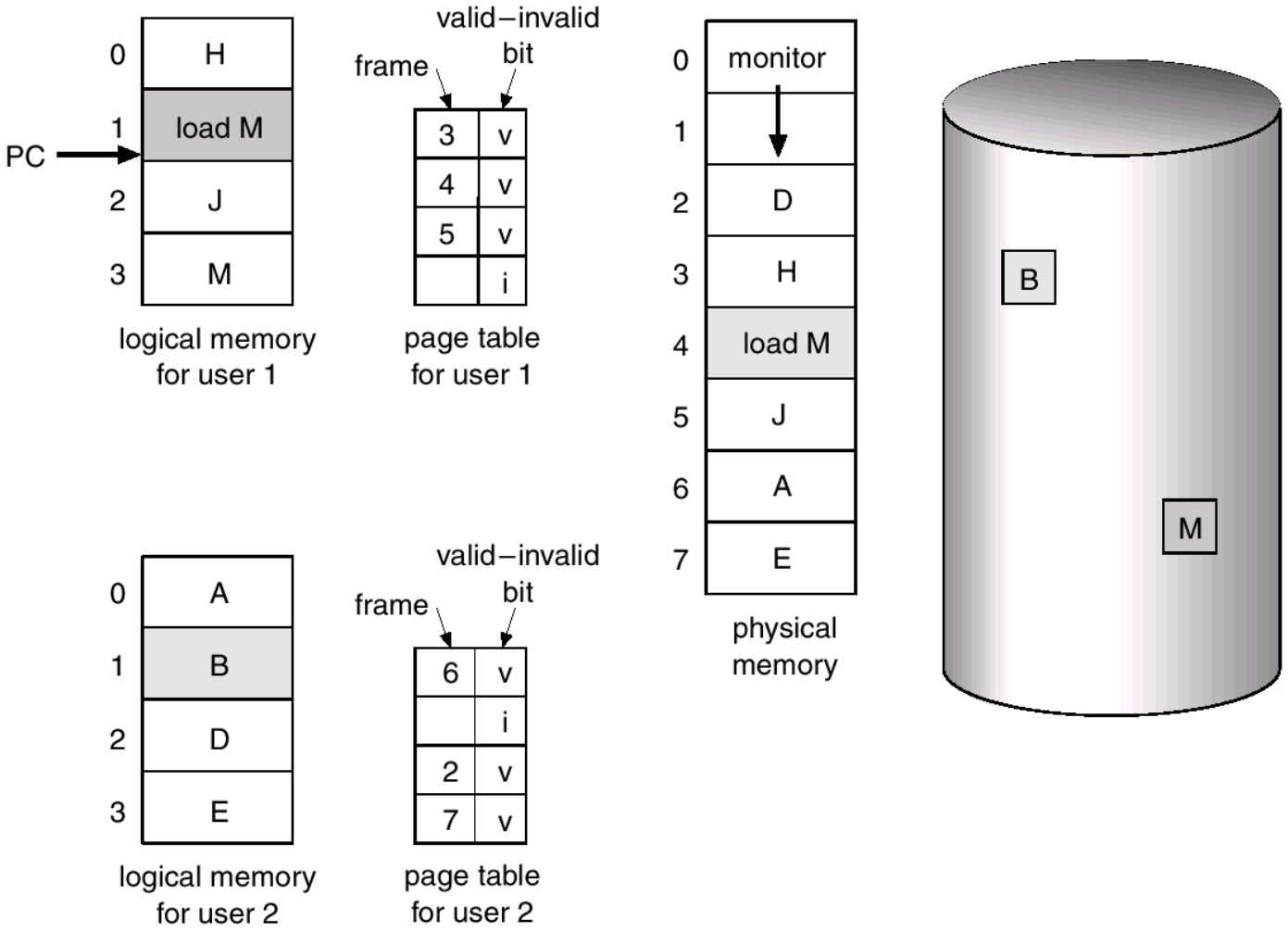
- While the process executes and accesses pages that are in memory, execution proceeds normally.
- By increasing the degree of multiprogramming, we will **over-allocate** the physical memory. When page fault occurs, no free frames available. How to proceed?
- Find a frame in memory, but not actively in use, page it out.
- Here comes the **Page replacement**
  - Prevent over-allocation of memory by modifying page-fault service routine to include page replacement.

What happens if no free frame?

- While the process executes and accesses pages that are in memory, execution proceeds normally.
- By increasing the degree of multiprogramming, we will **over-allocate** the physical memory. When page fault occurs, no free frames available. How to proceed?
- Find a frame in memory, but not actively in use, page it out.
- Here comes the **Page replacement**
  - Prevent over-allocation of memory by modifying page-fault service routine to include page replacement.
  - **Page replacement completes that large virtual memory can be provided on a smaller physical memory.**

Page replacement

Page replacement



Page-fault service routing including page replacement(1/2)

Page-fault service routing including page replacement(1/2)

- We should modify the page-fault service routine to include page replacement.

Page-fault service routing including page replacement(1/2)

- We should modify the page-fault service routine to include page replacement.
  1. Find the location of the desired page on the disk;

Page-fault service routing including page replacement(1/2)

- We should modify the page-fault service routine to include page replacement.
  1. Find the location of the desired page on the disk;
  2. Find a free frame:

Page-fault service routing including page replacement(1/2)

- We should modify the page-fault service routine to include page replacement.
  1. Find the location of the desired page on the disk;
  2. Find a free frame:
     1. If there is a free frame, use it; else

Page-fault service routing including page replacement(1/2)

- We should modify the page-fault service routine to include page replacement.
  1. Find the location of the desired page on the disk;
  2. Find a free frame:
     1. If there is a free frame, use it; else
     2. Use a **page-replacement algorithm** to select a **victim** frame;

Page-fault service routing including page replacement(1/2)

- We should modify the page-fault service routine to include page replacement.
  1. Find the location of the desired page on the disk;
  2. Find a free frame:
     1. If there is a free frame, use it; else
     2. Use a **page-replacement algorithm** to select a **victim** frame;
     3. Write the victim page to the disk; change the PTE accordingly.

Page-fault service routing including page replacement(1/2)

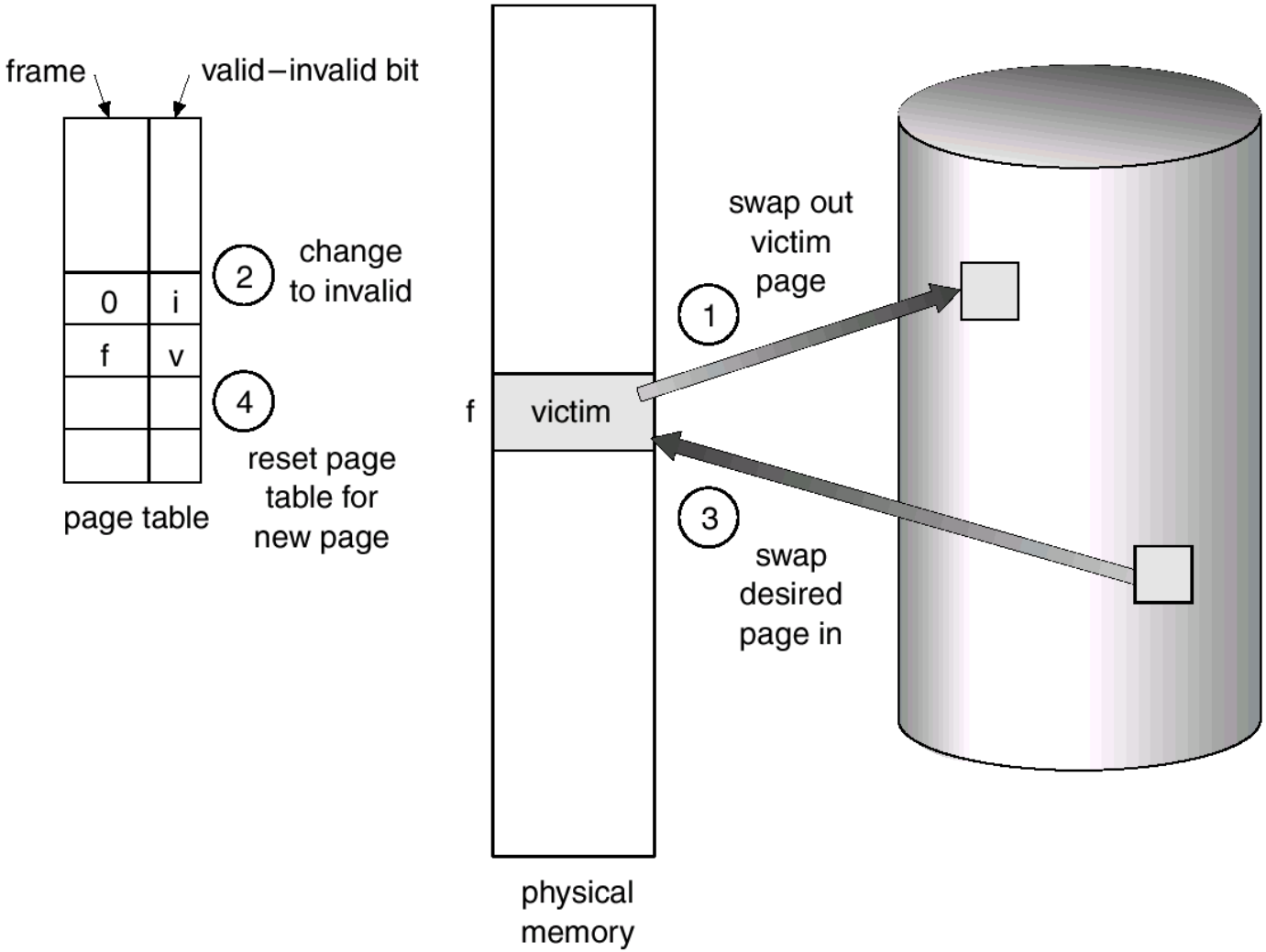
- We should modify the page-fault service routine to include page replacement.
  1. Find the location of the desired page on the disk;
  2. Find a free frame:
     1. If there is a free frame, use it; else
     2. Use a **page-replacement algorithm** to select a **victim** frame;
     3. Write the victim page to the disk; change the PTE accordingly.
  3. Read the desired page into the (newly) free frame; change the PTE;

Page-fault service routing including page replacement(1/2)

- We should modify the page-fault service routine to include page replacement.
  1. Find the location of the desired page on the disk;
  2. Find a free frame:
     1. If there is a free frame, use it; else
     2. Use a **page-replacement algorithm** to select a **victim** frame;
     3. Write the victim page to the disk; change the PTE accordingly.
  3. Read the desired page into the (newly) free frame; change the PTE;
  4. Restart the instruction.

Page-fault service routing including page replacement(2/2)

Page-fault service routing including page replacement(2/2)



Refinement

Refinement

- Notice that, if no frames are free, two page transfers are required:

Refinement

- Notice that, if no frames are free, two page transfers are required:
  - One write and one read

Refinement

- Notice that, if no frames are free, two page transfers are required:
  - One write and one read
- We can reduce this overhead by associating each page a **modify** bit (or **dirty** bit)

Refinement

- Notice that, if no frames are free, two page transfers are required:
  - One write and one read
- We can reduce this overhead by associating each page a **modify** bit (or **dirty** bit)
  - Whenever a page was modified, the hardware will set this bit.

Refinement

- Notice that, if no frames are free, two page transfers are required:
  - One write and one read
- We can reduce this overhead by associating each page a **modify** bit (or **dirty** bit)
  - Whenever a page was modified, the hardware will set this bit.
  - The page-out is only needed if dirty bit is set.

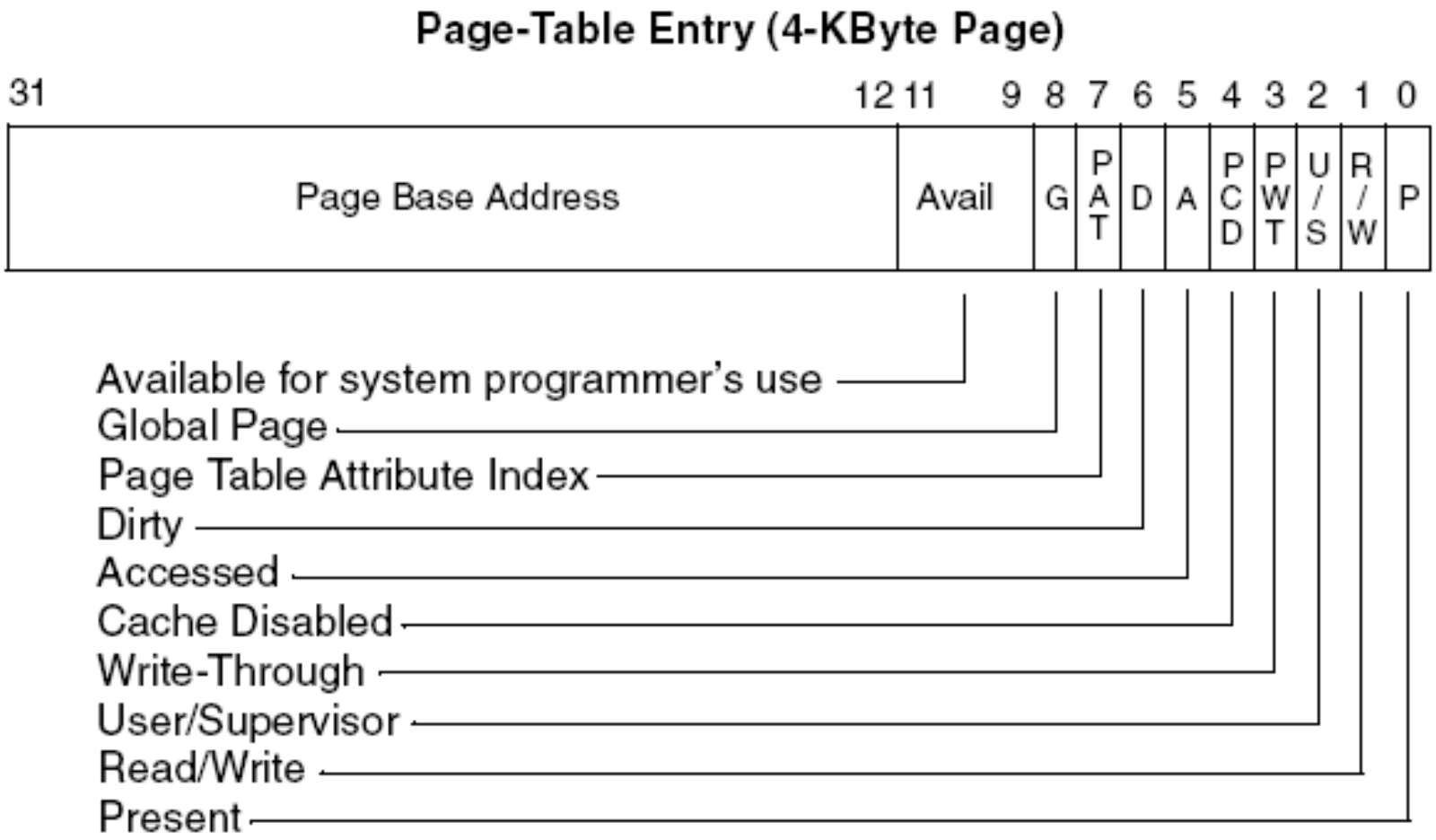
Example dirty bit

Example dirty bit

- PTE of IA-32

Example dirty bit

- PTE of IA-32



Questions

- Any questions?

Page-replacement algorithm

Page-replacement algorithm

- 算法目标：

Page-replacement algorithm

- 算法目标：
  - Lowest page-fault rate

Page-replacement algorithm

- 算法目标：
  - Lowest page-fault rate
- 为了方便研究各种页面置换算法，计算各个算法所产生的page fault，引入如下概念：

Page-replacement algorithm

- 算法目标：
  - Lowest page-fault rate
- 为了方便研究各种页面置换算法，计算各个算法所产生的page fault，引入如下概念：
  - *Reference string*

Page-replacement algorithm

- 算法目标：
  - Lowest page-fault rate
- 为了方便研究各种页面置换算法，计算各个算法所产生的page fault，引入如下概念：
  - *Reference string* - the string of memory references.

Page-replacement algorithm

- 算法目标：
  - Lowest page-fault rate
- 为了方便研究各种页面置换算法，计算各个算法所产生的page fault，引入如下概念：
  - *Reference string* - the string of memory references. 通常以页面为单位。

Page-replacement algorithm

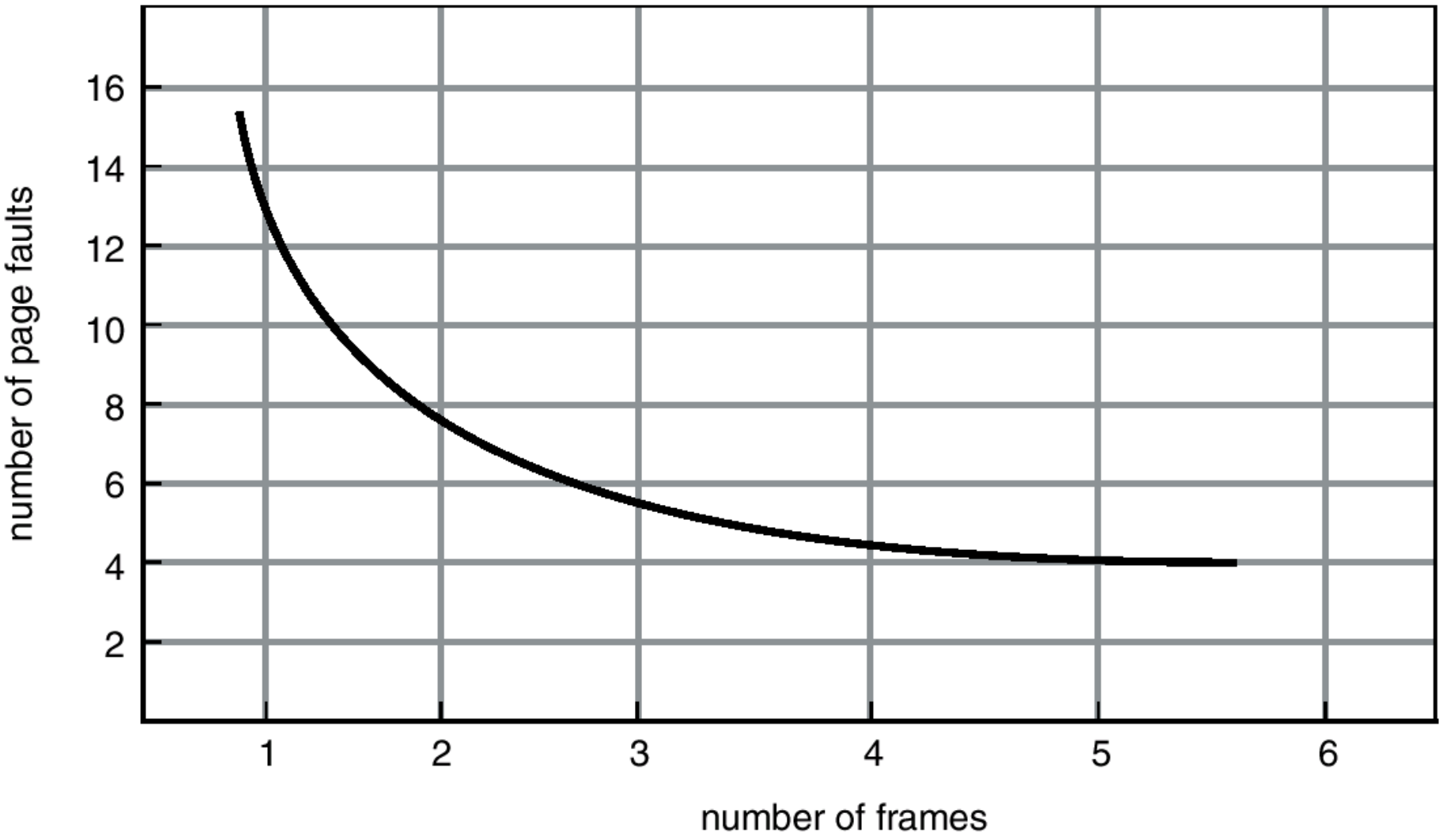
- 算法目标：
  - Lowest page-fault rate
- 为了方便研究各种页面置换算法，计算各个算法所产生的page fault，引入如下概念：
  - *Reference string* - the string of memory references. 通常以页面为单位。
- Evaluate an algorithm by running it on a reference string and computing the number of page faults on that string.

Page-replacement algorithm

- 算法目标：
  - Lowest page-fault rate
- 为了方便研究各种页面置换算法，计算各个算法所产生的page fault，引入如下概念：
  - *Reference string* - the string of memory references. 通常以页面为单位。
- Evaluate an algorithm by running it on a reference string and computing the number of page faults on that string.
  - In all following examples, the reference string is
    7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Page faults versus the number of frames

Page faults versus the number of frames



Questions

- Any questions?

Various page replacement algorithms

Various page replacement algorithms

- FIFO page replacement

Various page replacement algorithms

- FIFO page replacement
- Optimal page replacement

Various page replacement algorithms

- FIFO page replacement
- Optimal page replacement
- LRU page replacement

Various page replacement algorithms

- FIFO page replacement
- Optimal page replacement
- LRU page replacement
- Second-chance page replacement

FIFO page replacement

FIFO page replacement

- Replace the *oldest* page.

FIFO page replacement

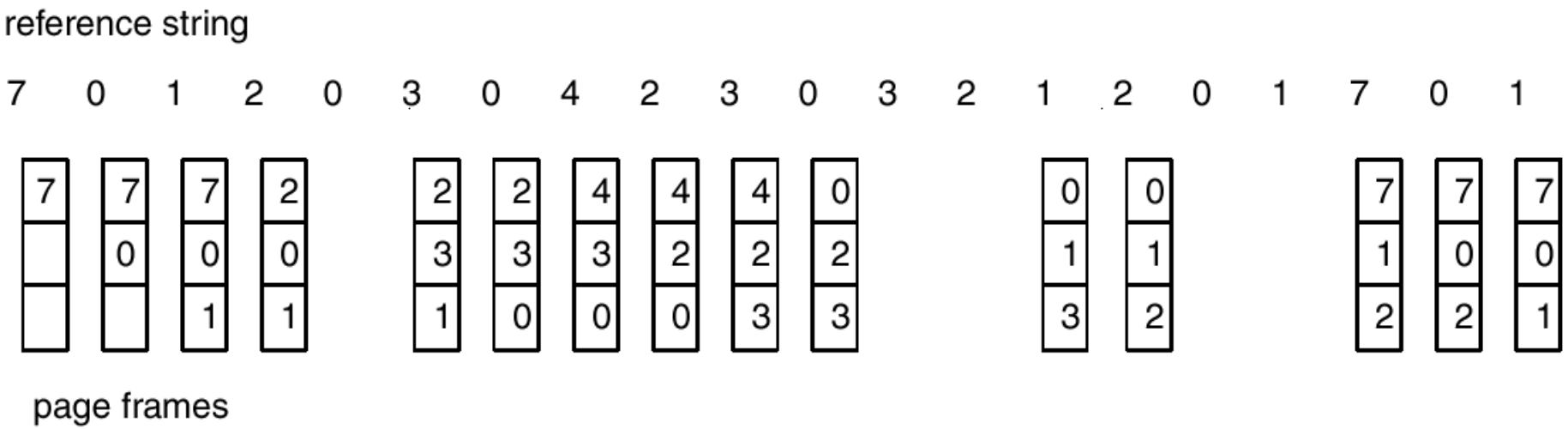
- Replace the *oldest* page.
- Example

FIFO page replacement

- Replace the *oldest* page.
- Example
  - Page faults with 3 frames.

FIFO page replacement

- Replace the *oldest* page.
- Example
  - Page faults with 3 frames.



Belady's anomaly

Belady's anomaly

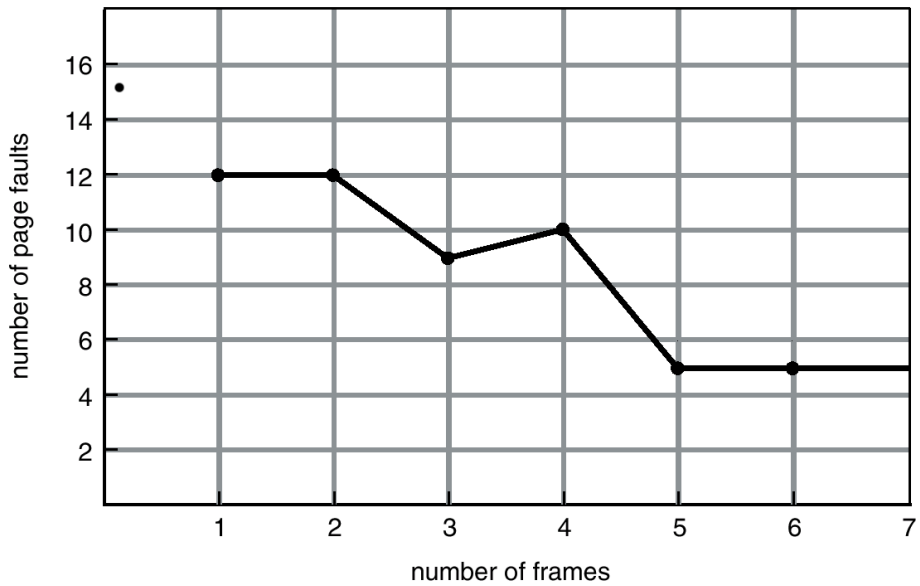
- 一般情况下，page-fault会随着frame的数量增加而减少，但是，如果采用FIFO算法，情况有时并非如此。

Belady's anomaly

- 一般情况下，page-fault会随着frame的数量增加而减少，但是，如果采用FIFO算法，情况有时并非如此。
- 例如，分别在有3和4个frame的系统上引用
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

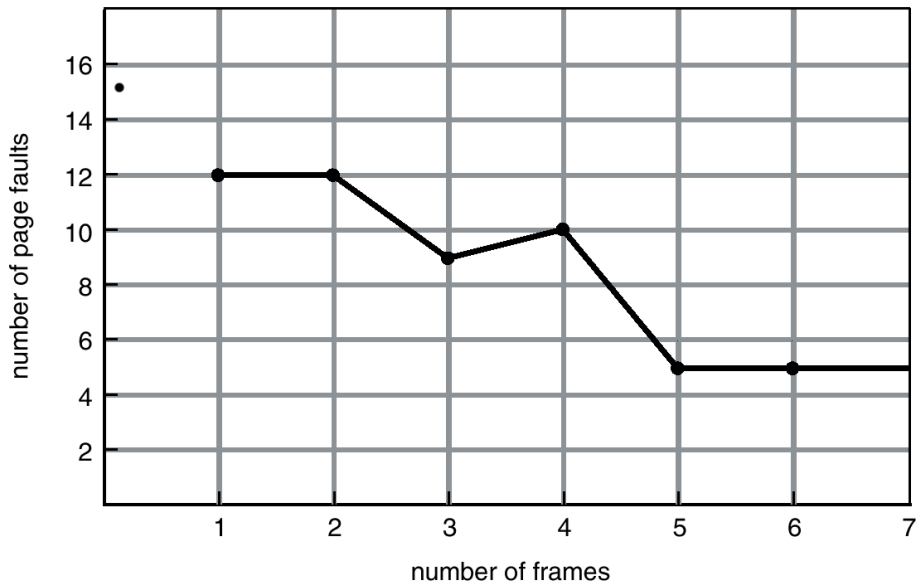
Belady's anomaly

- 一般情况下，page-fault会随着frame的数量增加而减少，但是，如果采用FIFO算法，情况有时并非如此。
- 例如，分别在有3和4个frame的系统上引用
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



Belady's anomaly

- 一般情况下，page-fault会随着frame的数量增加而减少，但是，如果采用FIFO算法，情况有时并非如此。
- 例如，分别在有3和4个frame的系统上引用
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- 因此，算法的设计非常重要。

Questions

- Any questions?

Optimal page replacement

Optimal page replacement

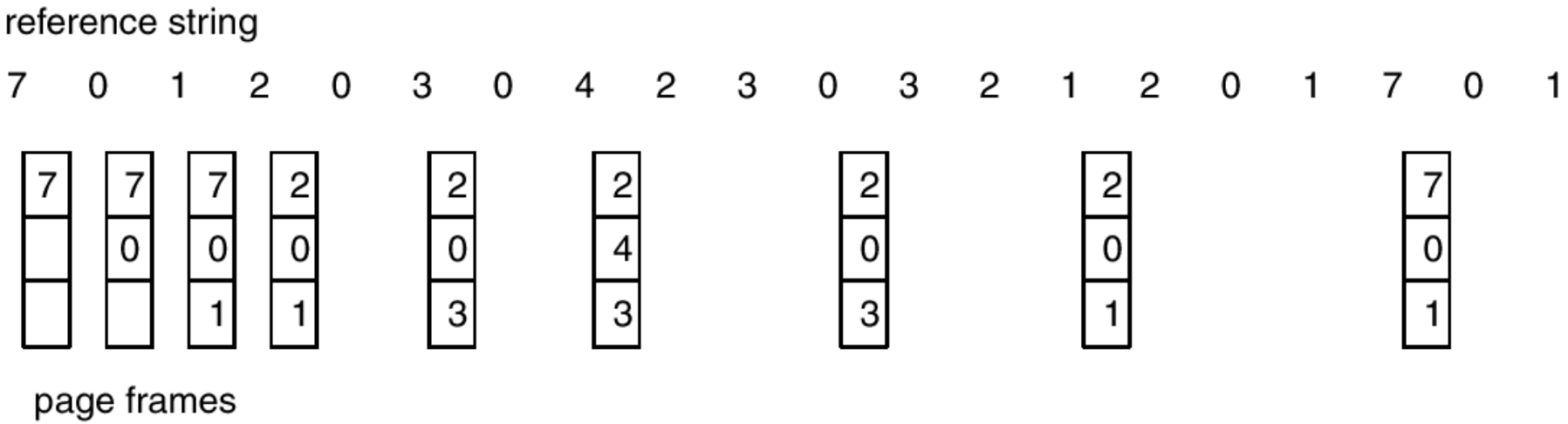
- Replace the page that will not be used for longest period time.

Optimal page replacement

- Replace the page that will not be used for longest period time.
- Example

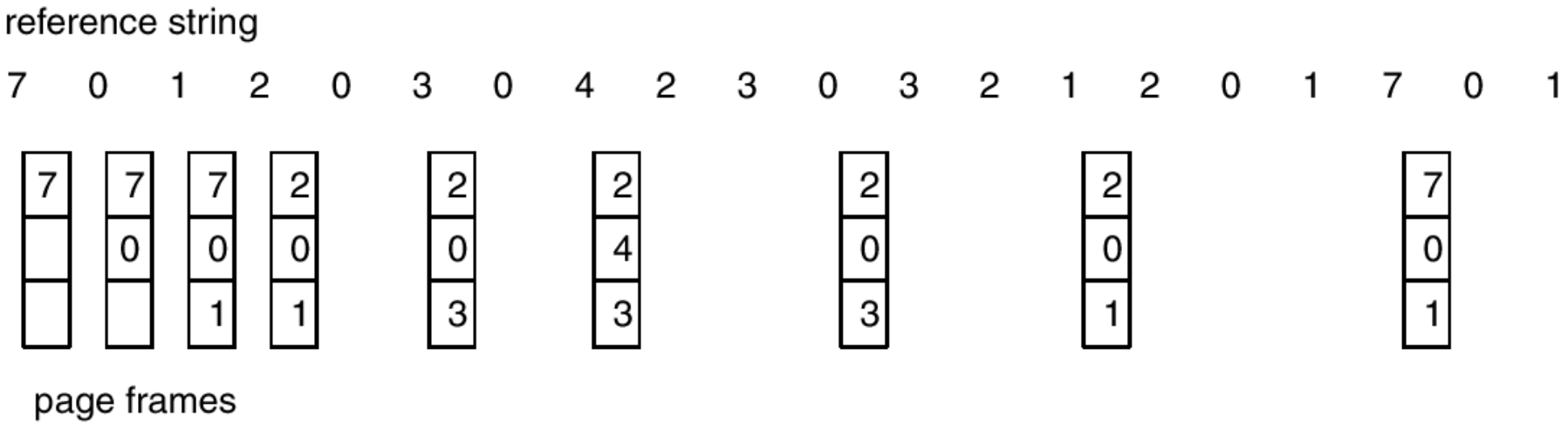
Optimal page replacement

- Replace the page that will not be used for longest period time.
- Example



Optimal page replacement

- Replace the page that will not be used for longest period time.
- Example



- Unfortunately, the optimal page replacement algorithm is difficult to implement because it requires future knowledge of the reference string.

LRU page replacement

LRU page replacement

- Replace the *least-recently-used* page.

LRU page replacement

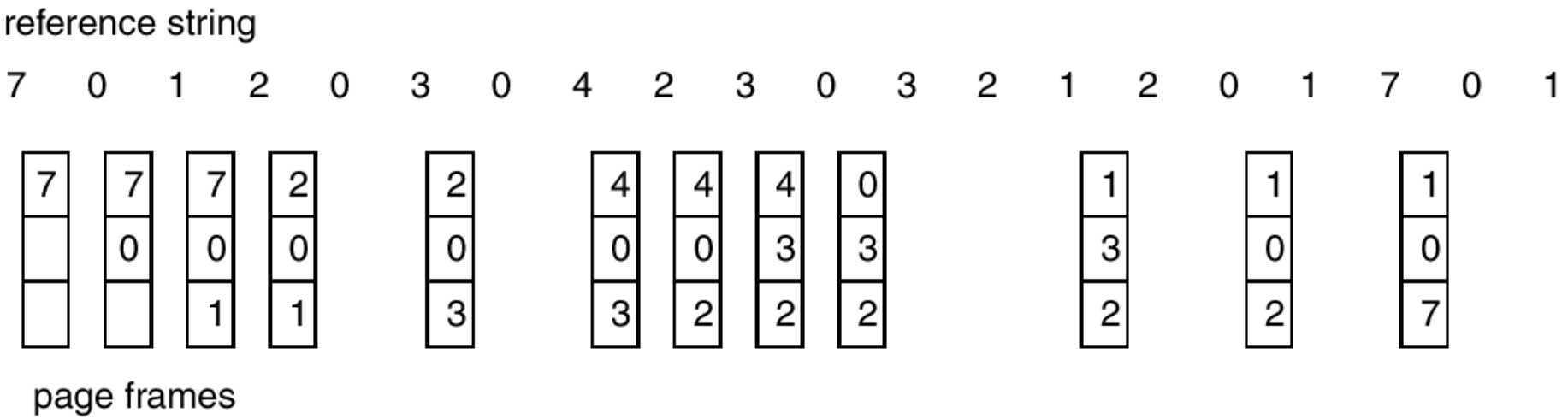
- Replace the *least-recently-used* page.
  - An approximation of the optimal page replacement.

LRU page replacement

- Replace the *least-recently-used* page.
  - An approximation of the optimal page replacement.
- Example

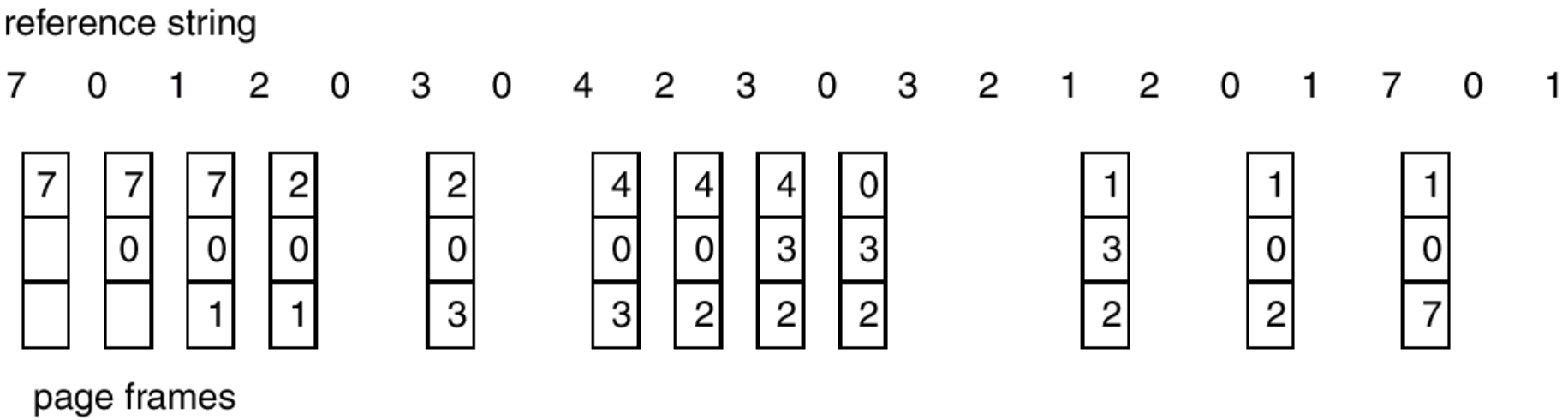
LRU page replacement

- Replace the *least-recently-used* page.
  - An approximation of the optimal page replacement.
- Example



LRU page replacement

- Replace the *least-recently-used* page.
  - An approximation of the optimal page replacement.
- Example



- The LRU replacement may require substantial hardware assistance to determine an order for the frames defined by the time of last use.

Second-chance page replacement (1/4)

Second-chance page replacement (1/4)

- Few computer system provide sufficient hardware support for true LRU algorithm.

Second-chance page replacement (1/4)

- Few computer system provide sufficient hardware support for true LRU algorithm.
- For an approximation, some hardware provide some help in the form of a *reference(R)* or *accessed(A)* bit.

Second-chance page replacement (1/4)

- Few computer system provide sufficient hardware support for true LRU algorithm.
- For an approximation, some hardware provide some help in the form of a *reference(R)* or *accessed(A)* bit.
  - Whenever a page is accessed, the hardware will set the *R* bit associated with the page.

Second-chance page replacement (1/4)

- Few computer system provide sufficient hardware support for true LRU algorithm.
- For an approximation, some hardware provide some help in the form of a *reference(R)* or *accessed(A)* bit.
  - Whenever a page is accessed, the hardware will set the *R* bit associated with the page.
- The algorithm works as following

Second-chance page replacement (1/4)

- Few computer system provide sufficient hardware support for true LRU algorithm.
- For an approximation, some hardware provide some help in the form of a *reference(R)* or *accessed(A)* bit.
  - Whenever a page is accessed, the hardware will set the R bit associated with the page.
- The algorithm works as following
  - When a page has been selected, we inspect its R bit.

Second-chance page replacement (1/4)

- Few computer system provide sufficient hardware support for true LRU algorithm.
- For an approximation, some hardware provide some help in the form of a *reference(R)* or *accessed(A)* bit.
  - Whenever a page is accessed, the hardware will set the R bit associated with the page.
- The algorithm works as following
  - When a page has been selected, we inspect its R bit.
    - If $R = 0$, replace this page;

Second-chance page replacement (1/4)

- Few computer system provide sufficient hardware support for true LRU algorithm.
- For an approximation, some hardware provide some help in the form of a *reference(R)* or *accessed(A)* bit.
  - Whenever a page is accessed, the hardware will set the R bit associated with the page.
- The algorithm works as following
  - When a page has been selected, we inspect its R bit.
    - If $R = 0$, replace this page;
    - Otherwise, we give it a second chance and move on to select next page.

Second-chance page replacement (1/4)

- Few computer system provide sufficient hardware support for true LRU algorithm.
- For an approximation, some hardware provide some help in the form of a *reference(R)* or *accessed(A)* bit.
  - Whenever a page is accessed, the hardware will set the R bit associated with the page.
- The algorithm works as following
  - When a page has been selected, we inspect its R bit.
    - If $R = 0$, replace this page;
    - Otherwise, we give it a second chance and move on to select next page.
  - When a page gets a second chance, its R bit is cleared.

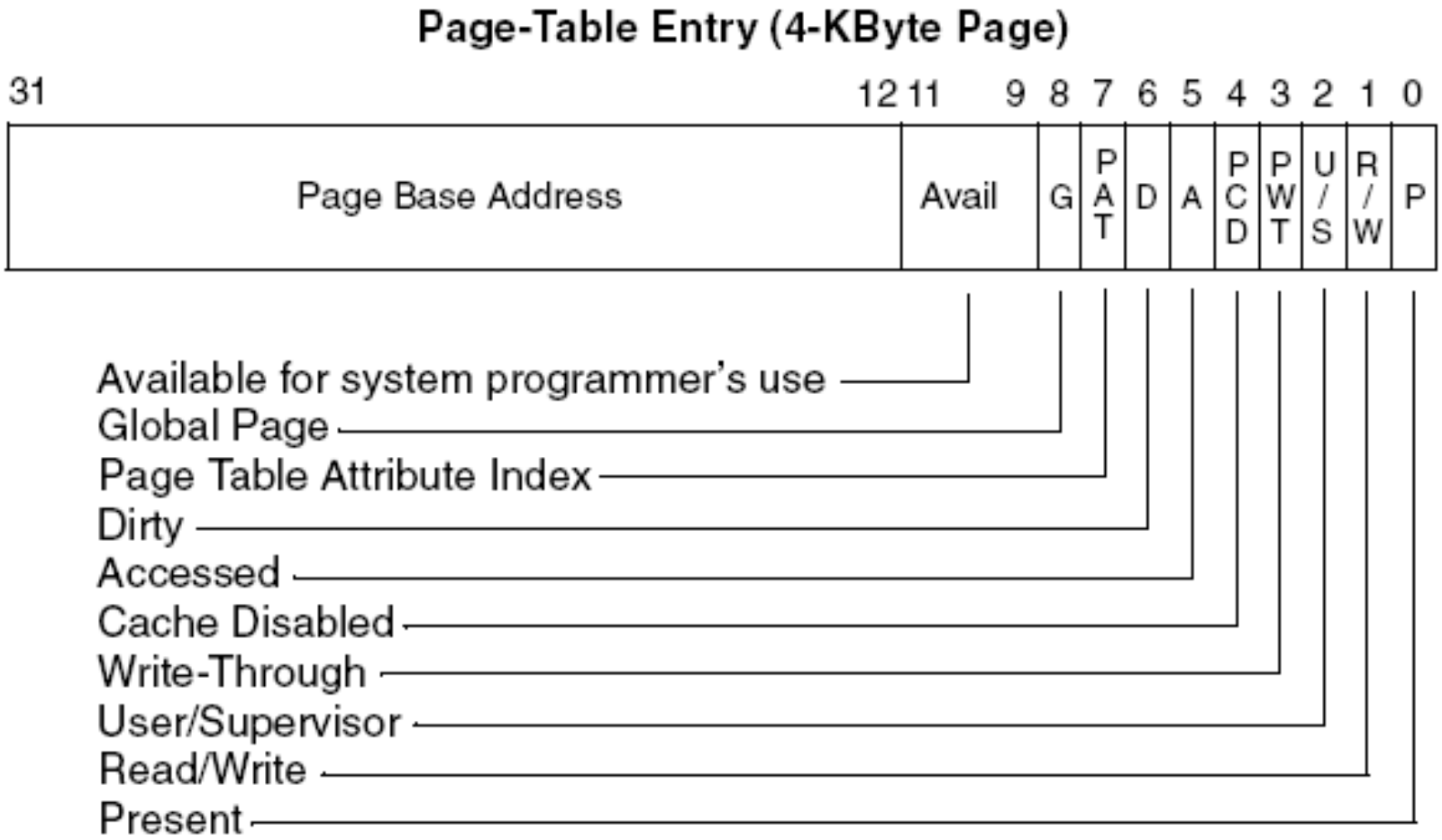
Second-chance page replacement (2/4)

Second-chance page replacement (2/4)

- Example reference (or accessed) bit

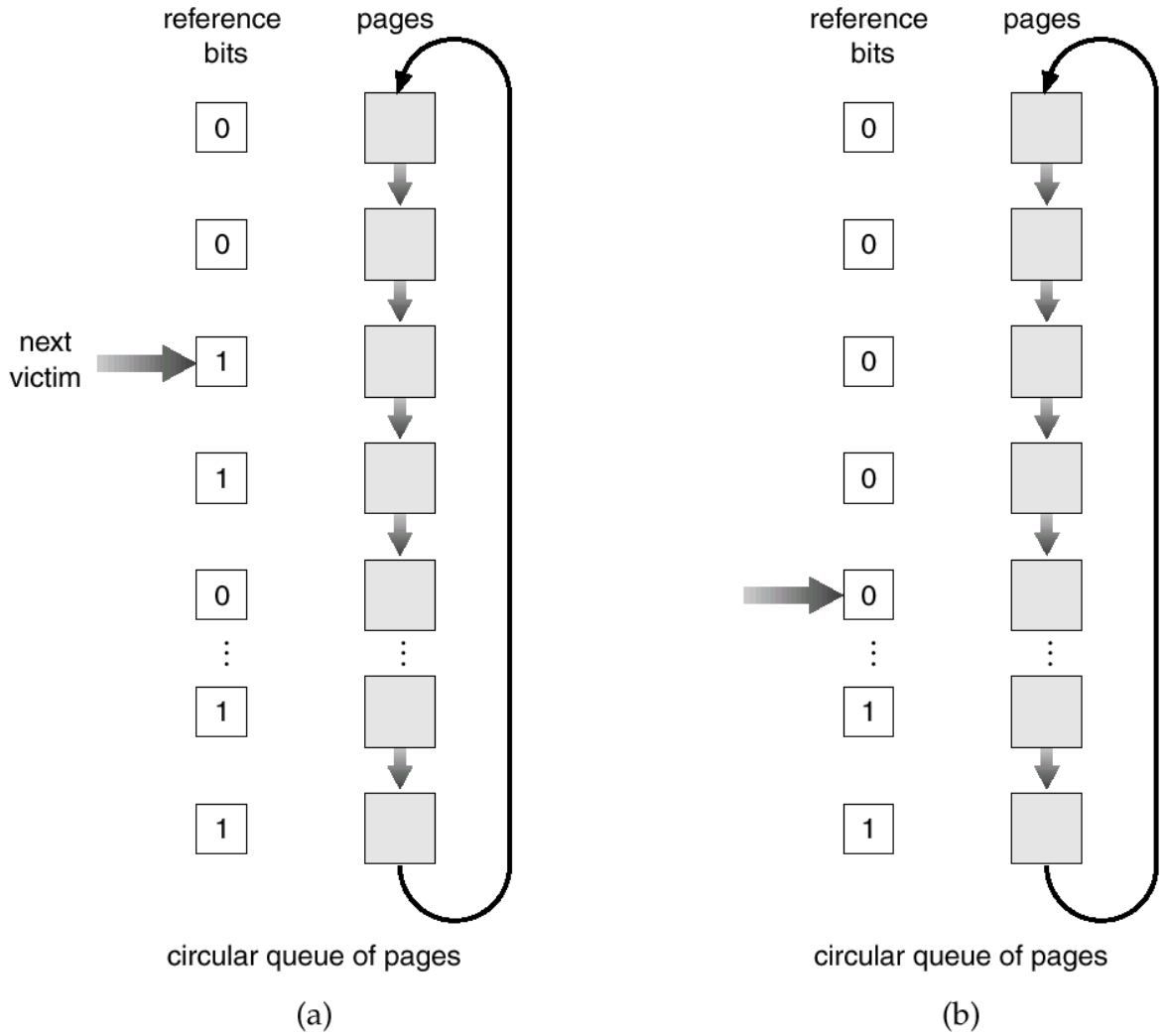
Second-chance page replacement (2/4)

- Example reference (or accessed) bit



Second-chance page replacement (3/4)

Second-chance page replacement (3/4)



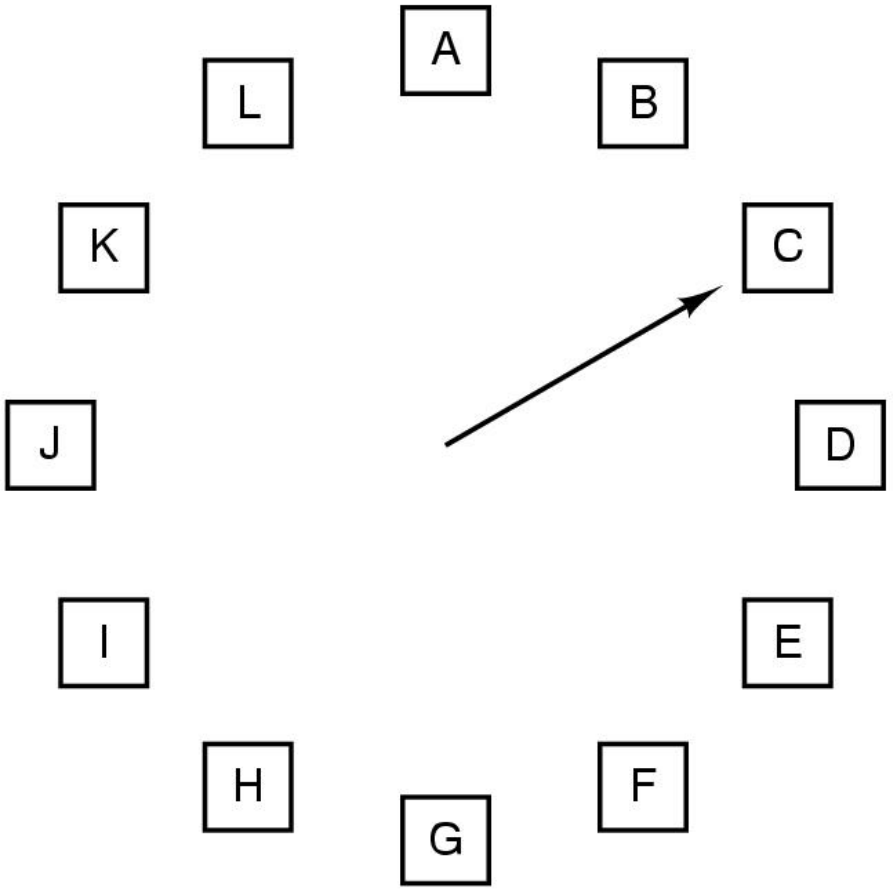
Second-chance page replacement (4/4)

Second-chance page replacement (4/4)

- Also known as *clock* page replacement

Second-chance page replacement (4/4)

- Also known as *clock* page replacement



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

Questions

- Any questions?

Performance

Performance

- *Page-fault rate*: $0 \leq p \leq 1.0$

Performance

- *Page-fault rate*: $0 \leq p \leq 1.0$
  - if $p = 0$, no page faults;

Performance

- *Page-fault rate*: $0 \leq p \leq 1.0$
  - if $p = 0$, no page faults;
  - if $p = 1$, every reference causes a fault.

Performance

- *Page-fault rate*: $0 \leq p \leq 1.0$
  - if $p = 0$, no page faults;
  - if $p = 1$, every reference causes a fault.
- Effective access time (EAT)
  $EAT = (1 - p) \times (\text{memory access time}) + p \times (\text{page fault time})$

Performance

- *Page-fault rate*: $0 \leq p \leq 1.0$
  - if $p = 0$, no page faults;
  - if $p = 1$, every reference causes a fault.
- Effective access time (EAT)
  $EAT = (1 - p) \times (\text{memory access time}) + p \times (\text{page fault time})$
- In general, memory access time is far less than the time to handle the page fault.

Performance

- *Page-fault rate*: $0 \leq p \leq 1.0$
  - if $p = 0$, no page faults;
  - if $p = 1$, every reference causes a fault.
- Effective access time (EAT)
  $EAT = (1 - p) \times (\text{memory access time}) + p \times (\text{page fault time})$
- In general, memory access time is far less than the time to handle the page fault.
  - So, we can assume that EAT is proportional to p, i.e., the page-fault rate.

Why does virtual memory works well?

Why does virtual memory works well?

- The principle behind the scene

Why does virtual memory works well?

- The principle behind the scene
  - *Locality model*

Why does virtual memory works well?

- The principle behind the scene
  - *Locality model*
    - *Locality* is defined as a set of pages that are actively used in the system.

Why does virtual memory works well?

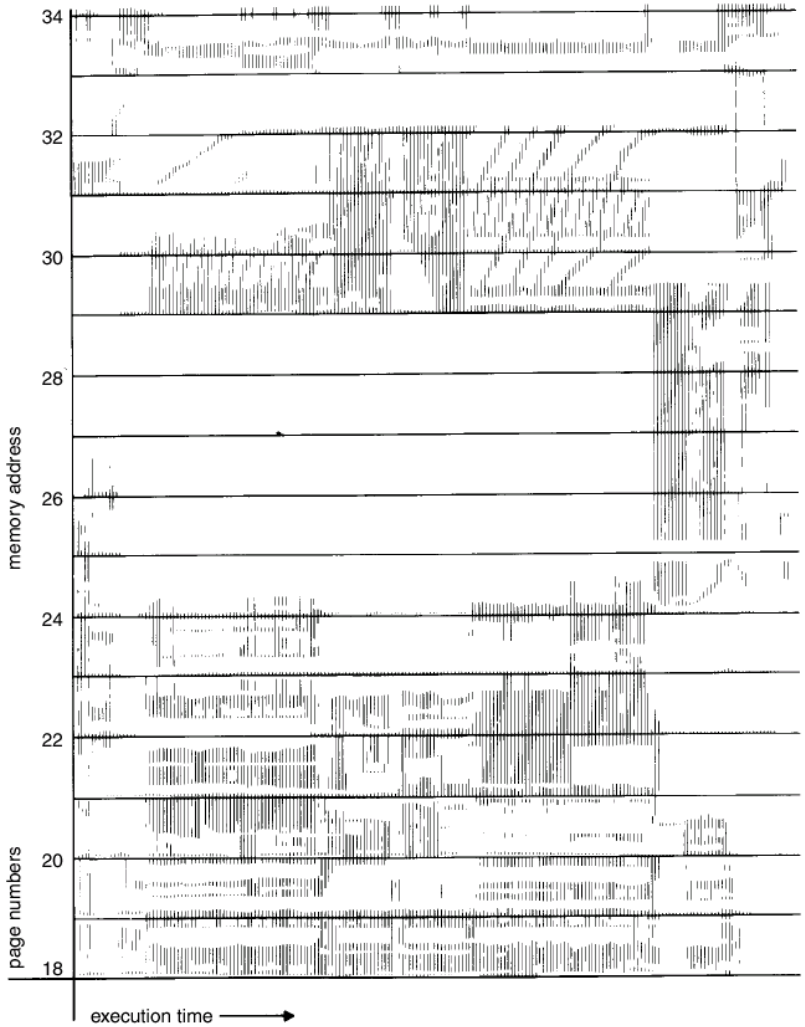
- The principle behind the scene
  - *Locality model*
    - *Locality* is defined as a set of pages that are actively used in the system.
    - *Process migrates from one locality to another*.

Why does virtual memory works well?

- The principle behind the scene
  - *Locality model*
    - *Locality* is defined as a set of pages that are actively used in the system.
    - *Process migrates from one locality to another*.
- Locality model is also the reason why the caches work well.

Locality model

Locality model



Thrashing (1/2)

Thrashing (1/2)

- If a system does not have "enough" frames, the page-fault rate is very high. This leads to:

Thrashing (1/2)

- If a system does not have “enough” frames, the page-fault rate is very high. This leads to:
  - Low CPU utilization.

Thrashing (1/2)

- If a system does not have "enough" frames, the page-fault rate is very high. This leads to:
  - Low CPU utilization.
  - The long-term scheduler thinks that it needs to increase the degree of multiprogramming.

Thrashing (1/2)

- If a system does not have "enough" frames, the page-fault rate is very high. This leads to:
  - Low CPU utilization.
  - The long-term scheduler thinks that it needs to increase the degree of multiprogramming.
  - More processes are added to the system.

Thrashing (1/2)

- If a system does not have "enough" frames, the page-fault rate is very high. This leads to:
  - Low CPU utilization.
  - The long-term scheduler thinks that it needs to increase the degree of multiprogramming.
  - More processes are added to the system.
- *Thrashing* is a situation where a system is busy bringing pages in and out and does nothing useful.

Thrashing (1/2)

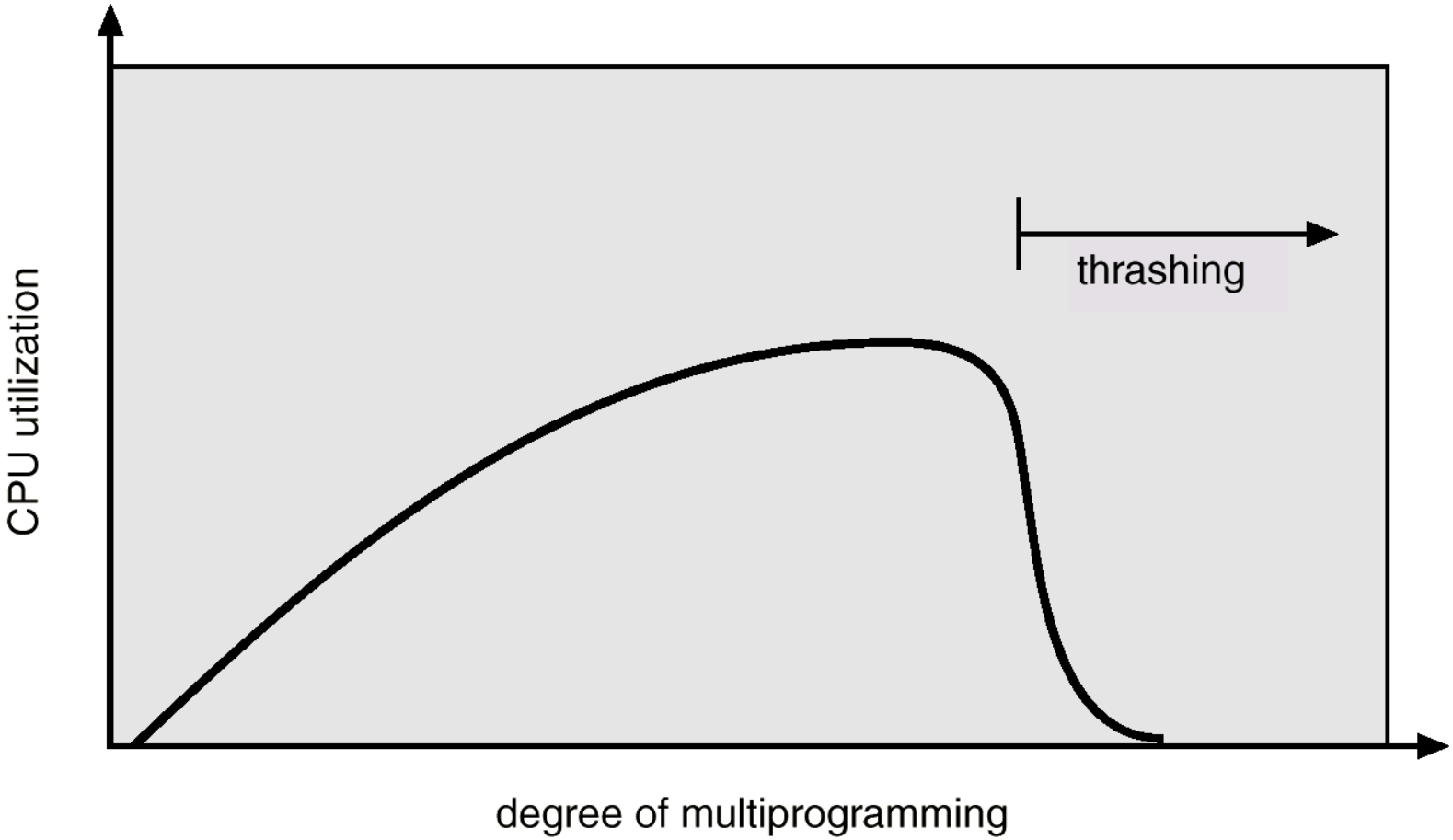
- If a system does not have "enough" frames, the page-fault rate is very high. This leads to:
  - Low CPU utilization.
  - The long-term scheduler thinks that it needs to increase the degree of multiprogramming.
  - More processes are added to the system.
- *Thrashing* is a situation where a system is busy bringing pages in and out and does nothing useful.
  - Why does thrashing occurs?

Thrashing (1/2)

- If a system does not have "enough" frames, the page-fault rate is very high. This leads to:
  - Low CPU utilization.
  - The long-term scheduler thinks that it needs to increase the degree of multiprogramming.
  - More processes are added to the system.
- *Thrashing* is a situation where a system is busy bringing pages in and out and does nothing useful.
  - Why does thrashing occurs?
    - $\sum size\ of\ locality > total\ memory\ size$

Thrashing (2/2)

Thrashing (2/2)



Questions

- Any questions?

Working-set model (1/2)

Working-set model (1/2)

- The *working-set model* is based on the assumption of locality.

Working-set model (1/2)

- The *working-set model* is based on the assumption of locality.
  - It defines a parameter named *working-set window*, Δ, to be the number of page references.

Working-set model (1/2)

- The *working-set model* is based on the assumption of locality.
  - It defines a parameter named *working-set window*, Δ, to be the number of page references.
  - The set of pages in the most recent Δ page references is the *working-set*.

Working-set model (1/2)

- The *working-set model* is based on the assumption of locality.
  - It defines a parameter named *working-set window*, Δ, to be the number of page references.
  - The set of pages in the most recent Δ page references is the *working-set*.
    - Thus, the working-set is an approximation of the process's locality.

Working-set model (1/2)

- The *working-set model* is based on the assumption of locality.
  - It defines a parameter named *working-set window*, Δ, to be the number of page references.
  - The set of pages in the most recent Δ page references is the *working-set*.
    - Thus, the working-set is an approximation of the process's locality.
  - We denote the WSS_i as the working set of process P_i in the most recent Δ.

Working-set model (1/2)

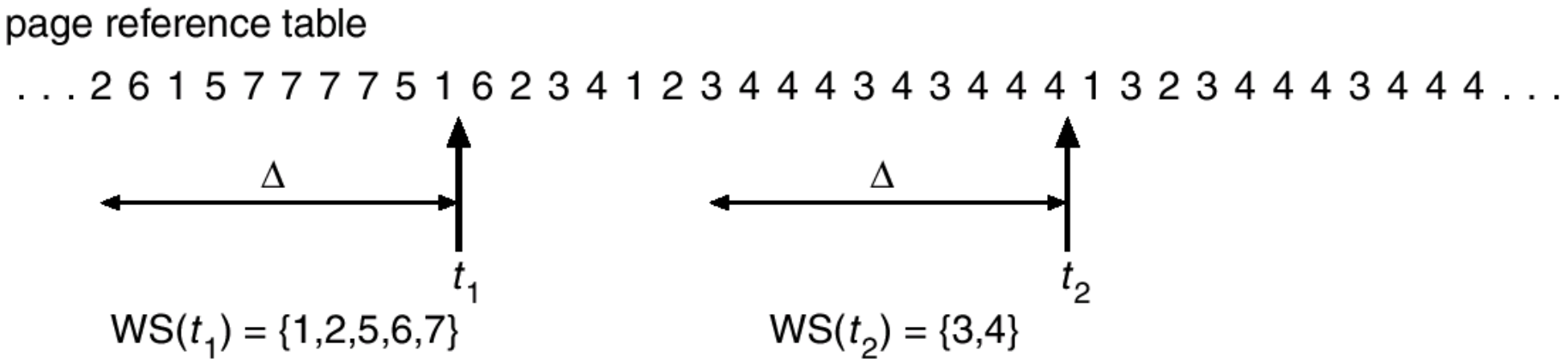
- The *working-set model* is based on the assumption of locality.
  - It defines a parameter named *working-set window*, Δ, to be the number of page references.
  - The set of pages in the most recent Δ page references is the *working-set*.
    - Thus, the working-set is an approximation of the process's locality.
  - We denote the WSS_i as the working set of process P_i in the most recent Δ.
    - Then, $D = \sum WSS_i$ is the total demand frames.

Working-set model (1/2)

- The *working-set model* is based on the assumption of locality.
  - It defines a parameter named *working-set window*, Δ, to be the number of page references.
  - The set of pages in the most recent Δ page references is the *working-set*.
    - Thus, the working-set is an approximation of the process's locality.
  - We denote the WSS_i as the working set of process P_i in the most recent Δ.
    - Then, $D = \sum WSS_i$ is the total demand frames.
    - If $D >$ total memory size, the system may be thrashing.

Working-set model (2/2)

Working-set model (2/2)



Use of working-set model

Use of working-set model

- The operating system monitors the working-set of each process and allocates to that working-set enough frames to provide it with its working-set size.

Use of working-set model

- The operating system monitors the working-set of each process and allocates to that working-set enough frames to provide it with its working-set size.
- If there are enough extra frames, new processes can be created; else if the sum of working-set size exceeds the total number of available frames, the OS selects a process to suspend.

Use of working-set model

- The operating system monitors the working-set of each process and allocates to that working-set enough frames to provide it with its working-set size.
- If there are enough extra frames, new processes can be created; else if the sum of working-set size exceeds the total number of available frames, the OS selects a process to suspend.
- *The working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible.*

Questions

- Any questions?

Outline

Address space layout (1/2)

Address space layout (1/2)

- With the introduction of the virtual memory, a huge, continuous and private address space is reserved to each process in the system.

Address space layout (1/2)

- With the introduction of the virtual memory, a huge, continuous and private address space is reserved to each process in the system.
  - On a 32-bit architecture, the address space has 2^{32} bytes.

Address space layout (1/2)

- With the introduction of the virtual memory, a huge, continuous and private address space is reserved to each process in the system.
  - On a 32-bit architecture, the address space has 2^{32} bytes.
  - Because all the processes must access the operating system kernel to request services, the operating system kernel must reside in the address space to be accessible from processes.

Address space layout (1/2)

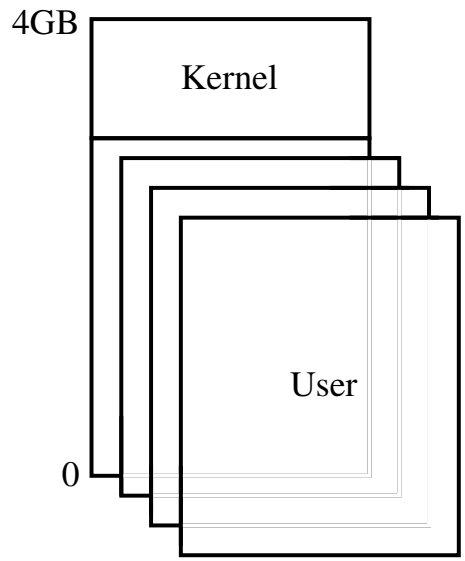
- With the introduction of the virtual memory, a huge, continuous and private address space is reserved to each process in the system.
  - On a 32-bit architecture, the address space has 2^{32} bytes.
  - Because all the processes must access the operating system kernel to request services, the operating system kernel must reside in the address space to be accessible from processes.
    - The part of the space allocated to operating system kernel is called *kernel space*, which is invisible to the process;

Address space layout (1/2)

- With the introduction of the virtual memory, a huge, continuous and private address space is reserved to each process in the system.
  - On a 32-bit architecture, the address space has 2^{32} bytes.
  - Because all the processes must access the operating system kernel to request services, the operating system kernel must reside in the address space to be accessible from processes.
    - The part of the space allocated to operating system kernel is called *kernel space*, which is invisible to the process;
    - The rest of the space is called *user space*, which can be used by the process.

Address space layout (1/2)

- With the introduction of the virtual memory, a huge, continuous and private address space is reserved to each process in the system.
  - On a 32-bit architecture, the address space has 2^{32} bytes.
  - Because all the processes must access the operating system kernel to request services, the operating system kernel must reside in the address space to be accessible from processes.
    - The part of the space allocated to operating system kernel is called *kernel space*, which is invisible to the process;
    - The rest of the space is called *user space*, which can be used by the process.



Address space layout (2/2)

Address space layout (2/2)

- The layout within the kernel and user space varies from one operating system to another.

Address space layout (2/2)

- The layout within the kernel and user space varies from one operating system to another.
  - For example, which part of user space holds the *text section*, *data section*, *stack* or *heap* of a process.

Address space layout (2/2)

- The layout within the kernel and user space varies from one operating system to another.
  - For example, which part of user space holds the *text section*, *data section*, *stack* or *heap* of a process.
- We will take Windows 2000 and Linux as the examples to explain their address space layout.

Windows 2000

Windows 2000

- References:

Windows 2000

- References:
  - Mark E. Russinovich and David A. Solomon, Windows Internals (Part 1, Part 2), 6th Edition, Microsoft Press, 2012.

Windows 2000

- References:
  - Mark E. Russinovich and David A. Solomon, Windows Internals (Part 1, Part 2), 6th Edition, Microsoft Press, 2012.
    - 中文版：深入解析Windows操作系统（第6版），潘爱民、范德成译

Windows 2000

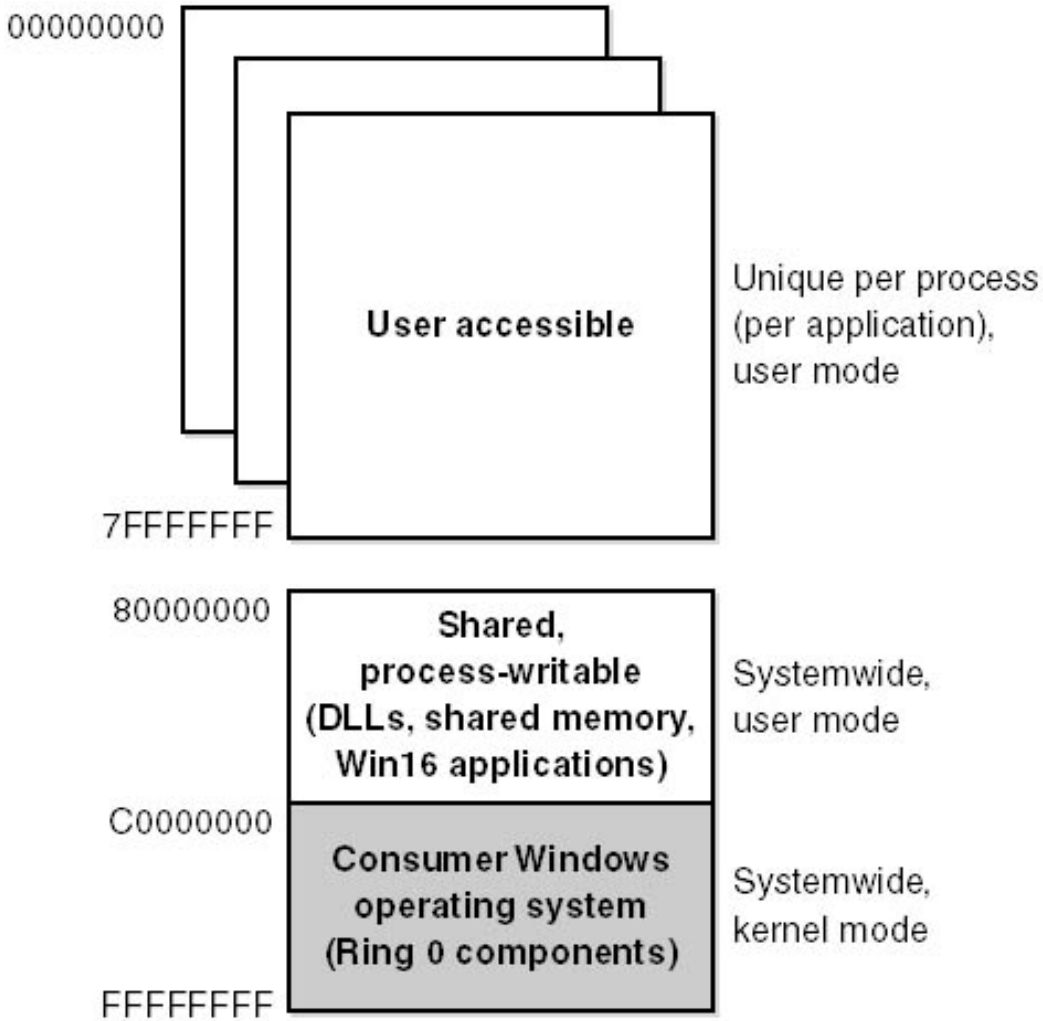
- References:
  - Mark E. Russinovich and David A. Solomon, Windows Internals (Part 1, Part 2), 6th Edition, Microsoft Press, 2012.
    - 中文版：深入解析Windows操作系统（第6版），潘爱民、范德成译
  - Jeffrey Richter, *Programming Applications for Microsoft Windows*, 4th Edition, Microsoft Press, 1999.

Windows 2000

- References:
  - Mark E. Russinovich and David A. Solomon, Windows Internals (Part 1, Part 2), 6th Edition, Microsoft Press, 2012.
    - 中文版：深入解析Windows操作系统（第6版），潘爱民、范德成译
  - Jeffrey Richter, *Programming Applications for Microsoft Windows*, 4th Edition, Microsoft Press, 1999.
    - 中文版：Windows核心编程（第4版），王建华译

Address space layout of Windows 2000 Professional

Address space layout of Windows 2000 Professional



User space layout of Windows 20000 Professional

User space layout of Windows 20000 Professional

- Command **VMMap.EXE** <**pid**> can be used to dump the user space layout of a Windows process

User space layout of Windows 20000 Professional

- Command **VMMap.EXE** <**pid**> can be used to dump the user space layout of a Windows process
  - *VMMap.EXE* can be found in the companion CD of the book *Programming Applications for Microsoft Windows*.

User space layout of Windows 20000 Professional

- Command **VMMap.EXE** <**pid**> can be used to dump the user space layout of a Windows process
  - *VMMap.EXE* can be found in the companion CD of the book *Programming Applications for Microsoft Windows*.

```
00010000 Private       4096  1  -RW-
00020000 Private       4096  1  -RW-
00030000 Private    1048576  3  -RW- Thread Stack
00130000 Mapped       12288  1  -R--
00260000 Mapped       90112  1  -R-- \Device\HarddiskVolume1\WINDOWS\system32\unicode.nls
00280000 Mapped      249856  1  -R-- \Device\HarddiskVolume1\WINDOWS\system32\locale.nls
002C0000 Mapped      266240  1  -R-- \Device\HarddiskVolume1\WINDOWS\system32\sortkey.nls
00310000 Mapped       24576  1  -R-- \Device\HarddiskVolume1\WINDOWS\system32\sorttbls.nls
00320000 Mapped      266240  1  -R--
00370000 Private      65536  2  -RW-
00380000 Mapped       12288  1  -R-- \Device\HarddiskVolume1\WINDOWS\system32\ctype.nls
00390000 Private      65536  2  -RW-
00400000 Image        45056  4  ERWC C:\foo.exe
0040B000 Free     264196096
10000000 Image        49152  5  ERWC C:\foo.dll
1000C000 Free    1820278784
7C800000 Image      1163264  5  ERWC C:\WINDOWS\system32\kernel32.dll
7C920000 Image       606208  5  ERWC C:\WINDOWS\system32\ntdll.dll
```

Questions

- Any questions?

Linux

- References

- References
  - Randal E. Bryant, David O' Hallaron, *Computer System: A Programmer's Perspective*, John & Wiley, 2004.

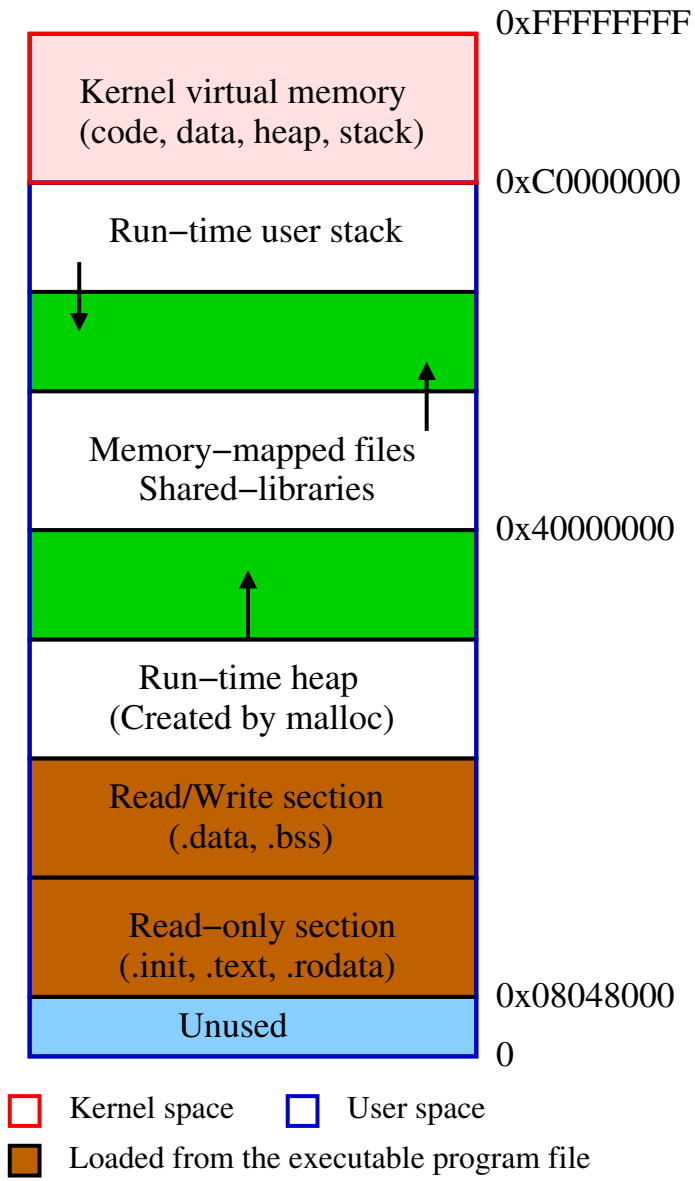
- References
  - Randal E. Bryant, David O' Hallaron, *Computer System: A Programmer's Perspective*, John & Wiley, 2004.
    - 中文版：深入理解计算机系统，龚奕利和雷迎春译

Linux

- References
  - Randal E. Bryant, David O' Hallaron, *Computer System: A Programmer's Perspective*, John & Wiley, 2004.
    - 中文版：深入理解计算机系统，龚奕利和雷迎春译
  - The Linux kernel source code: `http://www.kernel.org`.

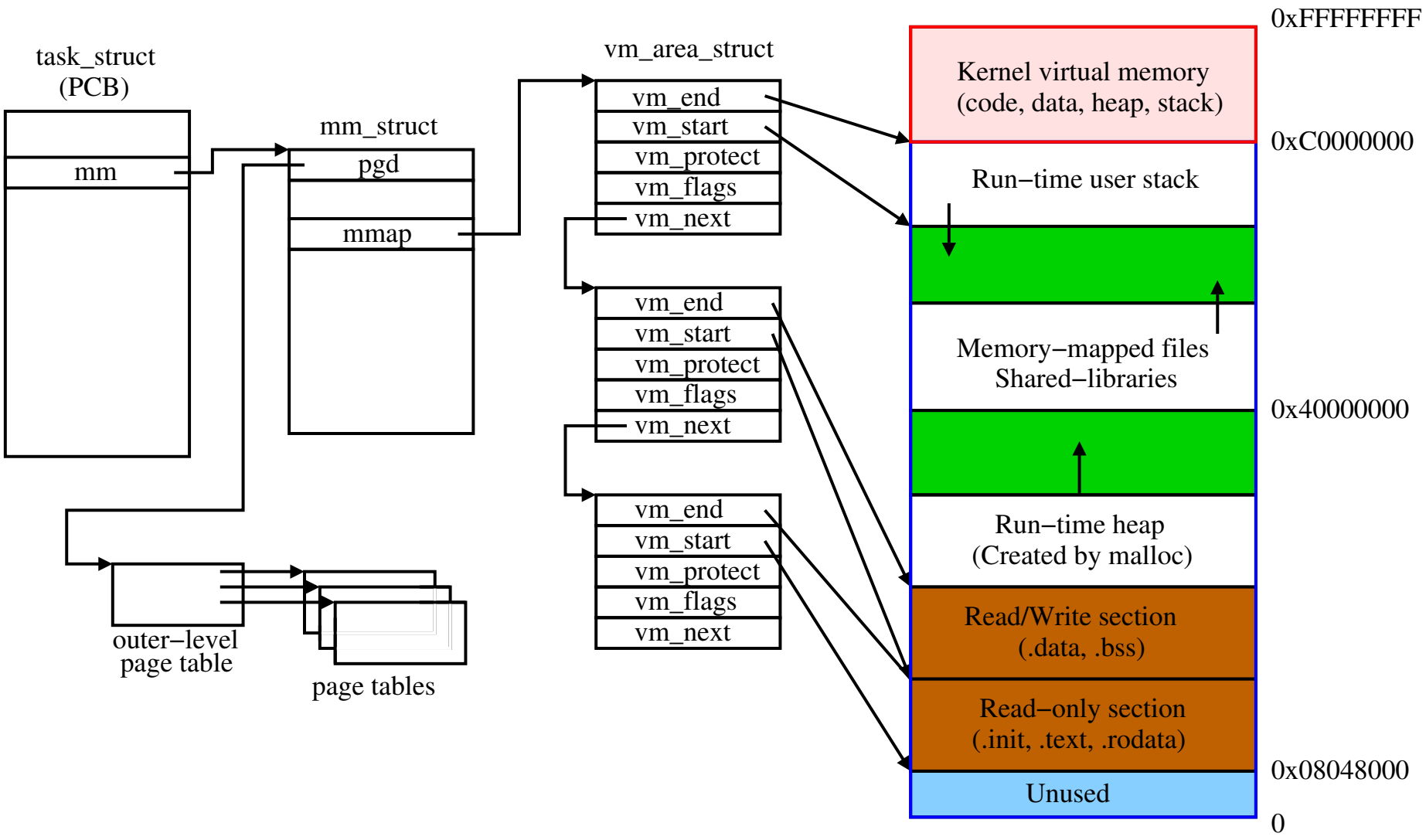
User space layout of Linux (1/2)

User space layout of Linux (1/2)



User space layout of Linux (2/2)

User space layout of Linux (2/2)



Example

Example

- *[hmj@hmj]$* **cat /proc/$(pidof foo)/maps**

Example

- *[hmj@hmj]$* **cat /proc/$(pidof foo)/maps**

```
08048000-08049000 r-xp 00000000 00:0d 14446  /home/hmj/foo
08049000-0804a000 rwxp 00000000 00:0d 14446  /home/hmj/foo
0804a000-0806b000 rwxp 0804a000 00:00 0      [heap]
b7e06000-b7e07000 rwxp b7e06000 00:00 0
b7e07000-b7f22000 r-xp 00000000 07:00 9506   /lib/libc-2.3.6.so
b7f22000-b7f23000 r-xp 0011b000 07:00 9506   /lib/libc-2.3.6.so
b7f23000-b7f26000 rwxp 0011c000 07:00 9506   /lib/libc-2.3.6.so
b7f26000-b7f29000 rwxp b7f26000 00:00 0
b7f2e000-b7f30000 rwxp b7f2e000 00:00 0
b7f30000-b7f46000 r-xp 00000000 07:00 9507   /lib/ld-2.3.6.so
b7f46000-b7f48000 rwxp 00015000 07:00 9507   /lib/ld-2.3.6.so
bfc31000-bfc46000 rw-p bfc31000 00:00 0      [stack]
ffffe000-fffff000 ---p 00000000 00:00 0      [vdso]
```

Example

- *[hmj@hmj]$* **cat /proc/$(pidof foo)/maps**

```
08048000-08049000 r-xp 00000000 00:0d 14446  /home/hmj/foo
08049000-0804a000 rwxp 00000000 00:0d 14446  /home/hmj/foo
0804a000-0806b000 rwxp 0804a000 00:00 0      [heap]
b7e06000-b7e07000 rwxp b7e06000 00:00 0
b7e07000-b7f22000 r-xp 00000000 07:00 9506   /lib/libc-2.3.6.so
b7f22000-b7f23000 r-xp 0011b000 07:00 9506   /lib/libc-2.3.6.so
b7f23000-b7f26000 rwxp 0011c000 07:00 9506   /lib/libc-2.3.6.so
b7f26000-b7f29000 rwxp b7f26000 00:00 0
b7f2e000-b7f30000 rwxp b7f2e000 00:00 0
b7f30000-b7f46000 r-xp 00000000 07:00 9507   /lib/ld-2.3.6.so
b7f46000-b7f48000 rwxp 00015000 07:00 9507   /lib/ld-2.3.6.so
bfc31000-bfc46000 rw-p bfc31000 00:00 0      [stack]
ffffe000-fffff000 ---p 00000000 00:00 0      [vdso]
```

- You can use the following command to get more details

Example

- *[hmj@hmj]$* **cat /proc/$(pidof foo)/maps**

```
08048000-08049000 r-xp 00000000 00:0d 14446  /home/hmj/foo
08049000-0804a000 rwxp 00000000 00:0d 14446  /home/hmj/foo
0804a000-0806b000 rwxp 0804a000 00:00 0      [heap]
b7e06000-b7e07000 rwxp b7e06000 00:00 0
b7e07000-b7f22000 r-xp 00000000 07:00 9506   /lib/libc-2.3.6.so
b7f22000-b7f23000 r-xp 0011b000 07:00 9506   /lib/libc-2.3.6.so
b7f23000-b7f26000 rwxp 0011c000 07:00 9506   /lib/libc-2.3.6.so
b7f26000-b7f29000 rwxp b7f26000 00:00 0
b7f2e000-b7f30000 rwxp b7f2e000 00:00 0
b7f30000-b7f46000 r-xp 00000000 07:00 9507   /lib/ld-2.3.6.so
b7f46000-b7f48000 rwxp 00015000 07:00 9507   /lib/ld-2.3.6.so
bfc31000-bfc46000 rw-p bfc31000 00:00 0      [stack]
ffffe000-fffff000 ---p 00000000 00:00 0      [vdso]
```

- You can use the following command to get more details
  - **cat /proc/$(pidof foo)/smaps**

Questions

- Any questions?

Relocation (revisited)

Relocation (revisited)

- References

Relocation (revisited)

- References
  - John R. Levine, *Linkers and Loaders*, Morgan-Kauffman, 1999.

Relocation (revisited)

- References
  - John R. Levine, *Linkers and Loaders*, Morgan-Kauffman, 1999.
  - GNU C Run-time library (glibc) source code: http://www.gnu.org/glibc

Address binding

Address binding

- Binding the symbols of the global variables and functions of a program to the addresses within the memory.

Address binding

- Binding the symbols of the global variables and functions of a program to the addresses within the memory.
- Address binding may happen in three stages:

Address binding

- Binding the symbols of the global variables and functions of a program to the addresses within the memory.
- Address binding may happen in three stages:
  1. Compile time (known as *static binding*)

Address binding

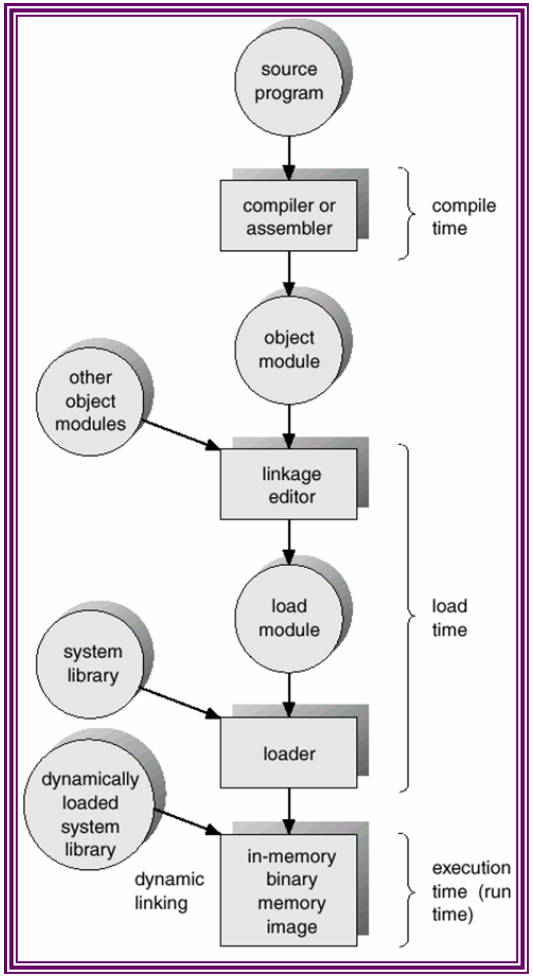
- Binding the symbols of the global variables and functions of a program to the addresses within the memory.
- Address binding may happen in three stages:
  1. Compile time (known as *static binding*)
  2. Load time

Address binding

- Binding the symbols of the global variables and functions of a program to the addresses within the memory.
- Address binding may happen in three stages:
  1. Compile time (known as *static binding*)
  2. Load time
  3. Run time (known as *dynamic binding*)

Address binding

- Binding the symbols of the global variables and functions of a program to the addresses within the memory.
- Address binding may happen in three stages:
  1. Compile time (known as *static binding*)
  2. Load time
  3. Run time (known as *dynamic binding*)



Relocation with the virtual memory

Relocation with the virtual memory

- With the virtual memory, because each process has a private address space,

Relocation with the virtual memory

- With the virtual memory, because each process has a private address space,
  - The executable file can always be loaded into the preferred address, i.e., can be static bound.

Relocation with the virtual memory

- With the virtual memory, because each process has a private address space,
  - The executable file can always be loaded into the preferred address, i.e., can be static bound.
  - For example, Windows programs are statically bound to the address *0x400000* and Linux programs to *0x08048000*.

Relocation with the virtual memory

- With the virtual memory, because each process has a private address space,
  - The executable file can always be loaded into the preferred address, i.e., can be static bound.
  - For example, Windows programs are statically bound to the address *0x400000* and Linux programs to *0x08048000*.
- But the shared libraries are still NOT so lucky.

Relocation of shared libraries

Relocation of shared libraries

- When shared libraries (*.DLL for WIN32, *.so for *nix) are compiled and linked, the programmers do not know where they will be loaded.

Relocation of shared libraries

- When shared libraries (*.DLL for WIN32, *.so for *nix) are compiled and linked, the programmers do not know where they will be loaded.
  - So, the shared libraries have to be statically bound to an preferred address.

Relocation of shared libraries

- When shared libraries (*.DLL for WIN32, *.so for *nix) are compiled and linked, the programmers do not know where they will be loaded.
  - So, the shared libraries have to be statically bound to an preferred address.
- If a process needs to load additional shared libraries at run time dynamically

Relocation of shared libraries

- When shared libraries (*.DLL for WIN32, *.so for *nix) are compiled and linked, the programmers do not know where they will be loaded.
  - So, the shared libraries have to be statically bound to an preferred address.
- If a process needs to load additional shared libraries at run time dynamically
  - If the preferred address of the shared library has been occupied by another one, the operating system has to load it to another address. In this case, the relocation is absolutely needed.

Relocation of shared libraries in Windows

Relocation of shared libraries in Windows

- In Microsoft Visual Studio, the shared libraries are statically bound to the address 0x10000000 by default.

Relocation of shared libraries in Windows

- In Microsoft Visual Studio, the shared libraries are statically bound to the address 0x10000000 by default.
  - If an application uses several shared libraries which are all bound to the same preferred address, relocation is inevitable.

Relocation of shared libraries in Windows

- In Microsoft Visual Studio, the shared libraries are statically bound to the address 0x10000000 by default.
  - If an application uses several shared libraries which are all bound to the same preferred address, relocation is inevitable.
- The relocation does burn some CPU cycles, so it will slow down the loading of shared libraries and applications.

Relocation of shared libraries in Windows

- In Microsoft Visual Studio, the shared libraries are statically bound to the address 0x10000000 by default.
  - If an application uses several shared libraries which are all bound to the same preferred address, relocation is inevitable.
- The relocation does burn some CPU cycles, so it will slow down the loading of shared libraries and applications.
  - After being relocated, the shared library may NOT be shared by other processes.

Relocation of shared libraries in Windows

- In Microsoft Visual Studio, the shared libraries are statically bound to the address 0x10000000 by default.
  - If an application uses several shared libraries which are all bound to the same preferred address, relocation is inevitable.
- The relocation does burn some CPU cycles, so it will slow down the loading of shared libraries and applications.
  - After being relocated, the shared library may NOT be shared by other processes.
- So, if you are developing an application which contains several shared libraries, relocation may be avoided by *re-base*ing them to different preferred addresses.

Relocation of shared libraries in Linux

Relocation of shared libraries in Linux

- The shared libraries in Linux can be compiled and linked into the so-called “Position Independent Code” (PIC).

Relocation of shared libraries in Linux

- The shared libraries in Linux can be compiled and linked into the so-called "Position Independent Code" (PIC).
  - The PIC can be loaded and executed (without being relocated) at ANY addresses.

Relocation of shared libraries in Linux

- The shared libraries in Linux can be compiled and linked into the so-called "Position Independent Code" (PIC).
  - The PIC can be loaded and executed (without being relocated) at ANY addresses.
  - If a shared library has to use the external symbols (global variables or functions) defined in other shared libraries.

Relocation of shared libraries in Linux

- The shared libraries in Linux can be compiled and linked into the so-called "Position Independent Code" (PIC).
  - The PIC can be loaded and executed (without being relocated) at ANY addresses.
  - If a shared library has to use the external symbols (global variables or functions) defined in other shared libraries.
    - A per-process private *jump table* is used.

Questions

- Any questions?