CS 4500 - Operating Systems
Module 10: Memory Management - Part 2

Stanley Wileman

Department of Computer Science
University of Nebraska at Omaha
Omaha, NE 68182-0500, USA

June 10, 2017
In This Module...

- We conclude our study of memory management by considering how to reduce fragmentation, eliminate the need for contiguous memory, and remove the physical memory size limit on programs.
- We will first consider overlays, a semi-automatic technique for managing the residence of program modules.
- Most of this module, however, will focus on virtual memory, both paged and segmented, and some related theoretical results.
Q. How can we remove, or at least reduce, the relationship between program size and physical memory size?

A. A variety of techniques have been proposed to deal with this relationship. Overlays, paged virtual memory, and segmented virtual memory are the three most popular techniques. We will examine each of these.
Overlays

Rather than load an entire program into memory at once, we can divide it into those modules that are always required and those that are required only at certain times during program execution. Many program modules do not all need to be in memory at the same time. Thus we can develop a scheme that loads only those that are currently needed, leaving those that aren’t needed at the current time on secondary storage (e.g. disk).

The “overlay” technique dates to before 1960 and is still used in some memory-constrained environments. Although they are now little-used on conventional architectures, the techniques used – particularly those used to implement inter-segment calls – point the way to dynamic linking.
A Sample Program

We only need f1 or f2 in memory at any time. Likewise, only f1a or f1b need be in memory when f1 is executing.

```c
f1() {
    f1a();
    f1b();
}
f2() { ... }
main() {
    f1();
    f2();
}
```
Several overlay implementations have been used. There are basically two schemes:

- Require the application to explicitly make system calls to request that the needed modules be brought into memory before they are referenced.
- Allow the operating system (through the linker) to transparently modify the program to perform the necessary loading of overlay modules at the time they are invoked (that is, when they are called).

In both schemes, the user must identify module dependencies, essentially a tree similar to that shown in the sample program just examined.

Then these details must be coded into a “script” that is used by the linker to prepare the various overlay components.
Summary of Problems with the Previous Schemes

- Using fixed-sized partitions is wasteful and inefficient.
- Using variable-sized partitions, small holes are created. These small holes aren’t individually large enough to hold programs. Compaction, and the associated wasted time, must then be used.
- All schemes require the use of contiguous memory for a process.
- Programs larger than the available physical memory cannot be run (except with overlays).
- Creating overlays is time consuming and potentially error-prone. The result will not necessarily be optimal for every situation. Modules may be loaded and unloaded unnecessarily, depending on the memory resources available when the program is executed.
A Memory “Wish List”

- Eliminate the need for contiguous memory for every process.
- Eliminate, or at least greatly reduce, the amount of memory wasted through internal and external fragmentation.
- Allow arbitrary program sizes without undue regard for the size of primary memory.
- Eliminate the need to manually determine program overlay structure.
Since the need for sufficiently large regions of contiguous memory is identified as a problem, we must ask “why is contiguous memory even required?”

The answer is because of the way physical memory is referenced.

Basically, memory is an array of elements (now usually bytes).

And what is the assumed memory layout of the elements in an array?

That’s right – array elements must be in contiguous memory locations, each identified by an integer from a sequence (e.g. 0, 1, 2, …).

Recall the formula for computing the address $a$ of element $i$ of an array with element size $s$ and starting address $b$ (assuming the first element has subscript 0):

$$a = b + i \times s$$
What Does Non-contiguous Memory Look Like?

- If the memory used by a process isn’t stored in a contiguous region, then it must obviously be stored piecewise in several smaller memory regions.

- Without loss of generality, we may assume that each of these smaller regions is contiguous. In the limit, of course, we would have a large number of one-element regions of memory.

- Historically, there have been two approaches to dividing the memory used by a process into non-contiguous regions:
  - In **paged** systems, each of the smaller pieces has the same size, which is independent of the program characteristics. Paged systems are very common, and we’ll spend most of our time discussing them.
  - In **segmented** systems, the smaller pieces are variable sized, and correspond to the functional modules of the program. Pure segmented systems are uncommon.
Eliminating the Need for Contiguous Memory

There are two obvious ways we could eliminate the need for contiguous memory:

- Make the process aware of the physical location and size of the various non-contiguous pieces of memory we’re using; or
- Leave the process unmodified – that is, let it continue to think it’s using contiguous memory – but change the mechanism used to map the addresses used by the process to physical memory addresses.

Changing a program’s code to deal with the layout of the non-contiguous regions of memory being used by a process is not promising, and is in direct opposition to one of the main purposes of an operating system: make system resources easier to use.

So we need a scheme to map the “assumed contiguous” addresses used by a program to the real memory locations used to store the code and data being used by the process.

The Memory Management Unit (or MMU) is the system component that implements this mapping scheme.
A process occupies virtual addresses starting at 0 and increasing sequentially. The process executes in a virtual address space, the set of all possible virtual addresses. A process only deals with and knows about virtual addresses during its execution.

Physical addresses are those corresponding to the possible physical, or real, memory locations. Since the operating system must manage the physical memory, some of it must deal with physical addresses. The physical address space is the set of all possible physical addresses.
Address and Memory Space Sizes

- The term virtual or physical address space refers to the set of all logically possible virtual or physical addresses, and depends only on the format of addresses (that is, the number of bits an address contains).

- The term memory space refers to the set of all valid physical addresses for a real system, and depends on the amount of installed memory.

- For example, many modern processors support 64-bit addresses, but it is unlikely that any real system will soon have $2^{64}$ (over $10^{19}$) bytes of physical memory.
While uncommon, it is not impossible for the size of the virtual address space to be different than the size of the physical address space.

That is, there may be more or fewer bits in a virtual address than there are bits in the physical address.

As noted earlier, it is common for the size of the virtual and physical address spaces to exceed the size of the memory space.
Memory Terminology Summary

CPU

Virtual Addresses (from Virtual Address Space)

Memory Management Unit (MMU)

Set of all actual addresses forms the Memory Space

Physical Addresses (from Physical Address Space)

Physical (Real) Memory
To use paged memory management:

- Divide the virtual address space used by a program into pieces with fixed sizes, each of which is called page. The size of each page is (somewhat obviously) called the page size.
- Divide the computer’s physical address space into page frames, each of which has the same size as a page.
- Put each page of the virtual address space into an unused page frame.

Since any page frame can hold any page, there are no unusable holes in memory. Thus we have effectively eliminated external fragmentation.
The contents of a page must be brought into primary memory, usually from a secondary storage device like a disk, before it can be referenced by the processor.

One technique is to delay bringing a page into memory until it is explicitly referenced.

But a page might also be brought into memory before it is referenced, in anticipation of its use.

Bringing a page into memory only as a result of an explicit reference is called demand paging.

Bringing pages into memory prior to explicit reference is called prepaging.

We will assume demand paging in our study unless stated otherwise.
Almost Rady To Execute . . .

- We have now gotten the program loaded into physical memory, as fixed-sized pieces in potentially non-contiguous locations (i.e. non-contiguous page frames).
- Before we can execute the program, though, we must have some technique for transforming the virtual addresses generated by the program into the corresponding physical addresses used to reference the computer’s physical memory.
- As noted previously, this is the function of the Memory Management Unit, or MMU.
- Although the earlier slide shows the MMU as as a separate component, most modern processors include the MMU. Logically, however, we can treat it as a separate component.
To achieve the mapping from a virtual address (which Intel calls a *linear address*) to a physical address, we first divide a virtual address $V$ into two components:

- The **page number** $P$ is the virtual address $V$ divided by the page size $S$:
  \[ P = \frac{V}{S} \]

  This yields the high-order bits of the virtual address.

- The **offset** $O$ is the virtual address $V$ modulo the page size $S$:
  \[ O = V \mod S \]

  This yields the low-order bits of the virtual address.

Physical addresses can also be divided into two parts, but the high order bits are called the **page frame number**.

Note that the offset of a byte in a page remains the same regardless of the page frame into which the page is placed.
Assume we have a page size (and page frame size) of 256 bytes.

An offset in a page is 8 bits long, so the low-order 8 bits of a virtual or physical address identifies the offset in a page or page frame.

The high-order bits of an address - that is, all but the low-order 8 bits in this example - identify the page or page frame number.

Virtual address \(600_{10} = 001001011000_2\). The low-order 8 bits are \(01011000_2\) or \(88_{10}\). The remaining high-order bits contain \(0010_2\) or \(2_{10}\).

So the memory referenced is in page 2 at offset 88. If page 2 is placed in page frame 6 (at physical address 1536), the byte at virtual address 600 is in physical address \(1624_{10}\), or \(011001011000_2\). The offset is still \(01011000_2\).
Since offsets are not changed by the MMU, all that’s left is to provide some facility for mapping page numbers to page frame numbers.

A page table provides the data for the mapping. It is an array with subscripts corresponding to page numbers. The page table entry with subscript $i$ contains the page frame number into which page $i$ has been placed.

During execution, the page number of each virtual address is mapped by the MMU using the page table, dynamically replacing the page number in the virtual address with the page frame number to yield a physical address.
A Small Program and Page Table

<table>
<thead>
<tr>
<th>Page 0</th>
<th>Frame 0</th>
<th>Page No.</th>
<th>Frame No.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Page 1</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Page 2</td>
<td></td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Page 3</td>
<td></td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Program (Virtual Address Space)  Memory (Physical Address Space)
Execution Algorithm - Version 1

Each time the contents of a memory location identified by a virtual address must be referenced, the following algorithm is used to complete the reference to the corresponding physical memory location. $PT$ represents the page table (as an array).

1. The MAR (Memory Address Register) is assumed to contain the virtual address to be referenced.
2. Set VPN (Virtual Page Number) = MAR div pagesize.
3. Set offset = MAR mod pagesize.
4. Set PPN (Physical Page Number) = PT [VPN].
5. Set PADDR (Physical ADDRess) = PPN × pagesize + offset.
6. Reference the memory at PADDR.
Example Execution

(For clarity, addresses are shown as decimal numbers.)

1. Page size is 256, MAR = 600
2. VPN = 600 \( \text{div} \) 256 = 2
3. Offset = 600 \( \text{mod} \) 256 = 88
4. PPN = \( PT [2] \) = 6
5. PADDR = 6 \( \times \) 256 + 88 = 1624
6. Reference memory at physical address 1624.
A Few Q and A about Page Tables

Q. How large are page tables?

A. For 32-bit address spaces with a 4K page size (like the Intel x86), a million pages are possible, with one entry per page. A page table entry on the Intel x86 requires 4 bytes. So the typical largest page table is 4MB long. A separate page table is needed for each process.

Q. Where are page tables stored?

A. In most systems, the page tables are stored in primary memory. Each reference to memory in the virtual address space will require two physical memory references, one to the page table and one for the item being referenced.
Although many programs won’t require a virtual address space as large as the physical address space, we still need a page table that can map an arbitrary virtual address referenced by the program.

In modern systems, a typical primary memory reference requires about 100 nanoseconds (that is, $100 \times 10^{-9}$ seconds) A modern Intel x86 running at a 3.0 GHz processor clock speed requires about 1 nanosecond to move a 32-bit data item between memory and a processor register.

Since each virtual address reference requires two physical memory address references (ignoring the possibility of caching), programs will run only about half as fast as if paging was not used (based on the time required to reference memory).
Allowing Arbitrary Program Sizes

- As we learned when considering overlays, not all parts of a program need to be resident in memory at the same time.
- With overlays, the programmer was able to direct the system to load several independent modules at the same locations in memory, with only one being present at any given time.
- We will now examine a technique that allows the system to execute a program without all its pages in primary memory, to automatically detect when a page is needed, and then to dynamically load it into an unused page frame.
If we want to be able to dynamically copy a needed page into a page frame, there are several things we must do. First, we must be able to identify when it’s not in a page frame.

To do this, we add an extra field containing a single bit to each page table entry.

This presence bit is set to 1 when a page is present (and the mapping provided by that page table entry is valid). The presence bit is 0 if the page isn’t loaded in primary memory.
Execution Algorithm - Version 2

The page table entries now have two fields, presence and framenumber.

1. The MAR (Memory Address Register), as before, contains the virtual address to be referenced.

2. Set VPN (Virtual Page Number) = MAR div pagesize.

3. Set offset = MAR mod pagesize.

4. If PT[VPN].presence == 0, then a page fault will occur. In response, the operating system will
   4.1 load the needed page into an unused page frame,
   4.2 set PT[VPN].framenumber to the page frame number, and
   4.3 set PT[VPN].presence = 1.

5. Set PPN (Physical Page Number) = PT[VPN].framenumber.

6. Set PADDR (Physical ADDRess) = PPN × pagesize + offset.

7. Reference the memory at PADDR.
Example with Presence Bits

Program (Virtual Address Space)

Page Table

<table>
<thead>
<tr>
<th>Page No.</th>
<th>Frame No.</th>
<th>Presence Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0 1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6 0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5 1</td>
</tr>
</tbody>
</table>

Memory (Physical Address Space)

Page 0
Page 1
Page 2
Page 3

Frame 0
Frame 1
Frame 2
Frame 3
Frame 4
Frame 5
Frame 6

?
Example Execution with Presence Bits

(For clarity, addresses are shown as decimal numbers.)

1. Page size is 256, MAR = 600
2. VPN = 600 div 256 = 2
3. Offset = 600 mod 256 = 88
4. PT[2].presence = 0, a page fault occurs, and
   4.1 OS loads page 2 into an unused frame, say frame 3
   4.2 PT[2].framenumber is set to 3
   4.3 PT[2].presence is set to 1
5. PPN = PT[2].framenumber = 3
6. PADDR = 3 × 256 + 88 = 856
7. Reference memory at physical address 856.
Page Table Storage, Reconsidered

Q. What if the page table was stored in special high-speed registers, and not in regular memory? Wouldn’t that make page table references faster?

Yes, but each context switch would require reloading the special registers with the page table entries for the process about to execute. If the page table is instead entirely in primary memory, we can use a single special purpose register to point to the page table. Then we only need to change the value in that single special register during a context switch.

For example, on the Intel x86 family the physical memory address of the page table is stored in special register CR3 (Control Register 3). Changing the contents of this register effectively changes the entire contents of the page table being used by the MMU to map virtual addresses to physical addresses.
Q. Since page tables are indexed like arrays (in fact, they are arrays), shouldn’t they be stored in contiguous memory if indexing is going to work?

A. Yes! And since page tables are frequently larger than the size of a page, we still have a contiguous memory requirement in the system. Ugh!

This contiguous memory requirement existed in the DEC VAX-11 system. The kernel of the operating system (VAX-VMS) had to arrange for sufficient contiguous kernel memory to hold the process table for each process.
To eliminate the need for continuous memory for page tables, some systems divide a virtual address into three (or possibly more) parts:

- a high-level page number (high-order bits),
- a low-level page number (middle bits), and
- an offset (low-order bits)

A level-1 page table containing pointers to several level-2 page tables is indexed by the high-level page number. The selected level-2 page table is then indexed by the low-level page number to locate the page frame. Each level-1 and level-2 page table fit in one page frame.
Example Using Three-Level Addressing

Virtual Address

Level 1 Page Table

Level 2 Page Table

Page Frame
It is interesting to note that since each level-2 page table occupies exactly one page, they may be optionally absent from memory, for example, when none of the pages identified by that page table is being referenced.

If a level-1 page table entry referencing an absent level-2 page table is used, a page fault is generated for it. As expected, the system retrieves the appropriate level-2 page table, puts it in an unused page frame and updates the corresponding level-1 page table entry.
Each virtual address has 32 bits. The high-order 10 bits (0..1023) index a page directory (what we’ve called a level-1 page table) which has 4-byte entries; the page directory thus occupies exactly 4K bytes.

The middle 10 bits index the level-2 page table identified by the page directory entry. Each page table has 1024 4-byte entries, occupying a total of 4K bytes.

The low-order 12 bits (0..4095) are used as the offset into a page, also 4K bytes long.
Most systems with paged virtual memory provide additional fields in their page table entries (PTEs). The most important include these:

- Protection - indicates what type of accesses are permitted (e.g. readonly, executeonly, noaccess, readwrite, etc.).
- Modified - indicates when the mapped page has been written (possibly changed).
- Referenced - indicates the mapped page has been accessed.
- Cache Disabled - indicates this PTE must not be cached (we'll discuss caching shortly).
What’s Left on the Memory Wish List?

Let’s review our earlier goals for memory:

- Eliminate the need for contiguous memory for every process. - **DONE**!
- Eliminate, or at least greatly reduce, the amount of memory wasted through internal and external fragmentation. - **DONE**, at least for external fragmentation!
- Allow arbitrary program sizes without undue regard for the size of primary memory. - **DONE**!
- Eliminate the need to manually determine program overlay structure. - **DONE**!

But we’ve introduced a few new problems...
New Problems

- We still haven’t addressed how to eliminate the need to access primary memory twice (the page table and the actual memory location) for each reference to a virtual address. Remember this problem makes paged virtual memory twice as slow as unpaged memory (measured in the number of memory references).

- And we still haven’t considered what to do where there are no available empty page frames to use when a page fault occurs.
Most processes exhibit an effect called **locality of reference**. This means that most of the memory references made by the process over a period of time are to a relatively invariant set of pages.

If we can save the page table entries for these pages somewhere that can be accessed more rapidly than primary memory, and save them in a way that will can find the needed entries rapidly (a sequential search through many entries in fast memory is still slow!), then we can eliminate many of the extra references to memory needed to get a page table entry.
Associative memory, also called Content Addressable Memory (CAM), is used in the MMU to implement the Translation Lookaside Buffer (TLB), a device that can rapidly map from a virtual page number to a physical page number.

As expected, CAM is implemented using fast special-purpose memory.

Each entry in the TLB effectively holds a recently referenced page table entry.

When a PTE is required, the entries in the TLB are searched first, in parallel, for the desired entry. Only if the PTE is not found in the TLB is the primary memory page table referenced.
The cost of associative memory is high, so relatively few entries are provided.

Since the PTEs stored in associative memory are for the currently executing process, when a context switch occurs the system must flush the associative memory, or invalidate each entry.

After a flush, the associative memory is “empty,” so the first few references to the “new” program will need to use the primary memory page table.

The associative memory is automatically updated by the hardware after each page table reference.

If the needed page table entry is found in the TLB, it’s called a “hit.” If the entry is not found, it’s called a “miss.”
Simple Associative Memory Logic

PTEs in CAM

Frame# | Page#
---|---
5 | 7
3 | 4
9 | 12

Desired Page # (input)

\[ a \]
\[ b \]
\[ c \]

Success Indicator

Comparator

Page Frame Number
Execution Algorithm - Version 3

1. Set $VPN = \text{MAR div pagesize}$
2. Set $offset = \text{MAR mod pagesize}$
3. If the associative memory has the appropriate PTE
   3.1 THEN $PPN = \text{PTE.framenumber}$
   3.2 ELSE If $\text{PT}[VPN].presence == 0$
      3.2.1 THEN generate page fault (and handle appropriately)
      3.2.2 ELSE $PPN = \text{PT}[VPN].framenum\text{ber}$
4. Set $PADDR = PPN \times \text{pagesize} + offset$
5. Reference memory at $PADDR$
A TLB is employed as usual.

If the TLB has the needed entry (a “hit”), then processing proceeds as usual.

But if the TLB doesn’t have the required page table entry (that is, a “miss”), a page fault is generated, and the operating system, in software

- looks up the appropriate entry from the RAM-resident page table,
- loads the page (if necessary),
- updates the TLB, and
- restarts the instruction that caused the miss.

This approach depends on good TLB performance for efficiency, since it omits the automatic fetch of page table entries from primary memory in hardware. This makes the MMU much less complex, and thus less expensive to build.
Inverted Page Tables

- This technique is motivated by very large virtual address spaces (for example, those with 64-bit virtual addresses).
- An inverted page table is indexed by page frame number, not by virtual page number, since there are far fewer frames than virtual pages.
- A TLB is used, as normal.
- On a TLB miss, the inverted page table is searched (linearly, or often with a hash table) for the entry.
- If not found, the conventional page table (on disk) is used to find the appropriate page.
- In either case, a new TLB entry is created.
If the operating system only loads pages in response to page faults, we’ll soon run out of free page frames.

Of course, each page loaded for a program is not continually used after it is loaded; some of these pages can be replaced.

So, the OS uses a page replacement algorithm to select pages to be removed when additional free page frames are needed.

Although “page replacement algorithm” is the traditional term used for such an algorithm, there is often no explicit replacement involved.

Instead, there is usually a “pool” of unused or available page frames maintained by the system.

The “page replacement algorithm” is occasionally used to select pages to be removed from the page frames they occupy in order to increase the size of the “pool,” usually when the number of unused pages in the pool falls below some specified limit.
This algorithm, also called “OPT,” selects for replacement that page which will be referenced furthest in the future.

The OPT algorithm is guaranteed to produce optimal results, when measured in the number of page faults that are generated; no algorithm can do better.

How do we identify the page that will be referenced furthest in the future?

Without knowledge of the future we can’t know!

This algorithm is useful only for evaluating the performance of other page replacement algorithms.

Q. Research question: who invented this algorithm?
Recall two fields from page table entries:

- R, the referenced bit, is set whenever a page has been read or written (since it was brought into primary memory, when R is set to 0)
- M, the modified (or dirty) bit, is set whenever a page has been written (again, since it was brought into primary memory, when M is set to 0)

These two fields can be helpful in the identification of pages suitable for replacement.
The R bit can be used in a clever way to identify pages that have not been used recently. Here's how:

- Periodically reset the R bit of each page in memory to 0.
- Later, when a specified period of time has elapsed, scan the R bit in page table entries to identify those pages that have not been referenced since the R bits were set to 0.
- These pages will have R bits still set to 0.
- The pages that haven’t been referenced recently are likelier candidates for removal than those that have been referenced recently.
The Not-Recently-Used Page Replacement Algorithm

- Use the method we just examined to identify those pages that have not been used recently.
- Each page in memory (that is, those that occupy a page frame) can be classified as follows using the value \( R \times 2 + M \):
  - 0 (\( R=0, M=0 \)): not referenced, not modified
  - 1 (\( R=0, M=1 \)): not referenced, modified
  - 2 (\( R=1, M=0 \)): referenced, not modified
  - 3 (\( R=1, M=1 \)): referenced, modified
- Pages in the lowest-numbered classes are likely candidates for removal.

Thought Q. How could a page have been modified if it hasn't been referenced (as in case 1)?
The First-In-First-Out (FIFO) Page Replacement Algorithm

- Select for replacement that page which was first brought into primary memory.
- This algorithm requires that we maintain a list of pages in the order of their entry into primary memory.
- Rationale: pages that have been in memory the longest are likely of no further use.
- Problem: this may replace pages that are being heavily used!
The Second Chance Page Replacement Algorithm

This algorithm attempts to compensate for the “heavy-handed” behavior of the FIFO algorithm. It gives a page the length of time it takes to move from the tail of the list to the head of the list to get referenced.

Algorithm:

- Examine the page at the head of the FIFO list. This would normally be the page that is to be replaced.
- If its R bit is 0, then select it for replacement.
- Otherwise
  - clear its R bit,
  - move it from the head to the tail of the FIFO list, and
  - repeat the algorithm to consider the new page at the head of the list.
The Clock Page Replacement Algorithm

- The second chance algorithm continually moves pages around on its list.
- The clock algorithm keeps the page frames in a circular list, with a “hand” pointing to the oldest page.
- At appropriate times (e.g., in response to a page fault), the R bit of the page pointed to by the hand is inspected:
  - \( R = 0 \): Remove the page in this frame, if any.
  - \( R = 1 \): Set R to 0.
- The examination of pages can continue until a suitable number of page frames have been emptied.
- A modern variation of the clock algorithm uses two hands that always have a fixed number of pages between them. When a frame is touched by the first hand, the R bit is set to 0. The R bit is checked, as before, when the second hand touches a frame. The goal is to eliminate the potentially huge time allowed for a process to reference a page when primary memory sizes are very large – as they typically are in modern machines.
The Least Recently Used Page Replacement Algorithm

- We observe that pages with recent heavy usage are not likely to be good candidates for replacement, as they will likely continue to be used in the near future.

- This suggests that we might profitably select for replacement the least recently used page in memory.

- Compare this algorithm, which selects the resident page last used furthest in the past, with OPT, which selects the page that will be used furthest in the future.
Implementation of “pure” LRU is expensive, as it requires that the position of each page in a list of referenced pages be updated on every memory reference!

To use this algorithm, some hardware support is necessary. We’ll look at two approaches:

- Use a real clock to record page reference times.
- Record an artificial time for each page reference.
Maintain a clock which is updated after every instruction is executed. Since each “time” must be unique, this clock will likely require more than 32 bits. With a 32 bit clock and a 30 MIPS execution rate, the clock will overflow in just over 2 minutes.

Record the “time” of the last reference in each page table entry. This will require a field as large as the clock in each page table entry, more than doubling the size of the page tables.

The page selected for replacement is that one with the smallest time in its PTE.
Keep an $n \times n$ array of bits, initially containing zeroes, where $n$ is the number of page frames in the system.

Whenever page frame $k$ is referenced

1. set all bits of row $k$ to 1, and
2. set all bits of column $k$ to 0.

The page selected for replacement is that one in the frame with the smallest binary value in any row of the array.

For a machine with $2^{20}$ page frames (like a common Intel system with 4GB in 4KB pages), this approach will require an array with $2^{40}$ total bits (or 137,438,953,472 bytes) and setting $2^{21} = 2,097,152$ bits after each memory reference. Ugh!
It should be clear that hardware assists for LRU implementation are not reasonable. A viable alternative, however, is to use software to approximate the effect of the LRU algorithm.

The first approach could be called Not Frequently Used or Least Frequently Used.

Periodically (each time a given timer expires), add the R bit for each PTE to a counter (initially zero) for each page (another field in the PTE) and reset the R bit.

This, in effect, tracks how many times a page has been referenced.

The page selected for replacement is that one that has the smallest counter value.

Unfortunately, these counts never get smaller, so recently loaded pages have a big disadvantage. They must accumulate a reference count larger than the pages that have been in memory for a longer time.
We can make a simple modification to the NFU/LFU algorithm by using aging to cause a page’s reference count to dimish if it’s not referenced frequently.

There are two changes to LFU:

- on each timer expiration, shift all counters right by one bit, and
- then insert the R bit into the leftmost bit position of each counter.

The counters maintain a history of recent references to each page during the interval between timer expirations.

The page selected for replacement is that one that has the smallest counter value.
Some Definitions

- A **reference string** (also called a **page trace**) is a list, in order, of the pages referenced by a program during execution.
- The **page fault rate** is the average probability of a page fault being generated during execution of an instruction. It is computed by counting page faults during execution of a reference string, then dividing that count by the length of the reference string.
Q. What will happen to the total number of page faults for a particular reference string if we increase the size of primary memory by one page frame?

That is, suppose we have a fixed program and fixed set of data to process using only a specific number, say \( N \), of page frames. If we execute the same program on the same data, but allow it to use \( N + 1 \) page frames, will the number of page faults decrease, stay the same, or increase?

Expected A. Since the size of memory is being increased, we expect the number of page faults will either decrease, or at the worst, will remain the same.

Correct A. The number of page faults could decrease, stay the same, or increase!
Belady’s Anomaly

- How the page fault rate (and the number of page faults) changes depends on the particular reference string, the number of page frames, and the particular page replacement algorithm being used.
- Belady’s Anomaly is the name given to this unexpected situation.
- It is easily illustrated using the FIFO page replacement algorithm.
Belady’s Anomaly – An Example

Consider this page trace with 12 entries:

1  2  3  4  1  2  5  1  2  3  4  5

When the FIFO page replacement algorithm is used with 3 page frames for this page trace, 9 page faults will result.

When the number of page frames is increased from 3 to 4, the number of page faults increases to 10!
FIFO: 9 Page Faults with 3 Frames

<table>
<thead>
<tr>
<th>Trace:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault?:</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frame 1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame 2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Frame 3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
### FIFO: 10 Page Faults with 4 Frames

<table>
<thead>
<tr>
<th>Trace:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault?:</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

| Frame 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 5 | 5 | 5 | 5 | 4 | 4 |
| Frame 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 5 |
| Frame 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 |
| Frame 4 | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 3 | 3 | 3 | 3 |

---

S. Wileman, UNO  
CS 4500 - Operating Systems, Module 10: Memory Management - Part 2  
65
### OPT: 7 Page Faults with 3 Frames

<table>
<thead>
<tr>
<th>Trace:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault?:</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

| Frame 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 4 | 4 |
| Frame 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| Frame 3 | 3 | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
## OPT: 6 Page Faults with 4 Frames

<table>
<thead>
<tr>
<th>Trace:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault?:</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

| Frame 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 4 | 4 |
| Frame 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| Frame 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| Frame 4 | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
LRU: 10 Page Faults with 3 Frames

<table>
<thead>
<tr>
<th>Trace:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault?</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

| Frame 1 | 1 | 1 | 1 | 4 | 4 | 4 | 5 | 5 | 5 | 3 | 3 | 3 |
| Frame 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 4 | 4 |   |
| Frame 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 5 |   |   |
LRU: 8 Page Faults with 4 Frames

<table>
<thead>
<tr>
<th>Trace:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault?:</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

| Frame 1 | 1 | 1 | 1 | 1 | 1 | 1 | 5 | 5 | 5 | 5 | 1 | 5 |
| Frame 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| Frame 3 | 3 | 3 | 3 | 3 | 3 | 5 | 5 | 5 | 5 | 4 | 4 | 4 |
| Frame 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 3 | 3 | 3 | 3 |
A certain class of page replacement algorithms has been proved to never exhibit Belady’s anomaly.

Algorithms in this class are called stack algorithms.

LRU and OPT are stack algorithms.

FIFO is not a stack algorithm.
We use an array, called $M$, to record the pages that have been referenced, both those that are in primary memory and those that are not.

- The upper part of the array, called $m$, keeps track of the $m$ pages in primary memory.
- The lower part of $M$ keeps track of the pages that have been referenced but are not currently in primary memory.
- Pages that have never been referenced are not anywhere in $M$.
- Initially $M$ is empty.
Operation of the Model

When a page is referenced...

- If the page is already in $m$, some reordering of the pages in $m$ may take place, but no pages will enter or leave primary memory.
- If the referenced page is not in $m$ and an empty frame exists in $m$, then the new page is moved in, a fault occurs, and pages can be reordered (if necessary, depending on the page replacement algorithm being modeled).
- If an empty frame does not exist in $m$ for the new page, a fault occurs, some page currently in $m$ is removed (to the lower part of $M$), and the new page is entered.
Observations about Stack Algorithms

- Modeling a stack algorithm, like OPT, clearly shows why it does not exhibit Belady's anomaly.
- Since the pages in $M$ (including those not in $m$) are ordered using the same algorithm at each time interval.
- The ordering is independent of the size of $m$.
- Increasing the value of $mm$ will not change the ordering of the pages in $M$. Thus, if $m$ corresponds to an increased number of page frames, pages that were in $m$ will still be in $m'$. This is called the inclusion property.
### Modeling LRU: m has 3 Frames

<table>
<thead>
<tr>
<th>Trace:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>most recently used page</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The rows above the heavy line represent the pages that are in \( m \) – that is, those that are in memory. The rows below the heavy line represent the pages that have been referenced, but are not in memory. However, *they are still in the ordering produced by the page replacement algorithm* – in this case, LRU.

If the heavy line is moved down to include more rows (representing a larger primary memory), the ordering of pages will not change, and the pages that were previously in memory will still be in memory. Thus the number of pages faults cannot increase.
A distance string is just the sequence of (1-origin) positions of each referenced page in $M$ at the time the page is referenced.

$\infty$ is used in the distance string for pages that have never been referenced.

A vector, $C$, records the number of times a particular distance occurs in a distance string. For example $C_k$ is the number of times a distance $k$ appears in the distance string.
### A Distance String and C Vector

#### Trace:

<table>
<thead>
<tr>
<th>Trace:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
</table>

#### most recently used page

<table>
<thead>
<tr>
<th>most recently used page</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
</table>

#### Distance string

| Distance string | ∞ | ∞ | ∞ | ∞ | 4 | 4 | ∞ | 3 | 3 | 5 | 5 | 5 |

<table>
<thead>
<tr>
<th>( C_1 )</th>
<th>( C_2 )</th>
<th>( C_3 )</th>
<th>( C_4 )</th>
<th>( C_5 )</th>
<th>( C_\infty )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

\( C_i = \) number of times \( i \) appears in the distance string
Predicting Page Fault Rates

- Start with a reference string, then compute the distance string.
- The distance string is independent of memory size, but is dependent on the page replacement algorithm.
- Compute the values in the C vector.
- The number of faults that will occur when executing in a memory of size $n$ is just $C_{n+1} + C_{n+2} + \ldots$. 
The set of pages a process is currently using is called its working set.

Because of locality, the membership of the working set changes slowly.

If the working set is resident (that is, in primary memory), few faults will be generated (as the set membership changes) and the process will run efficiently.

If the available memory is insufficient to hold the working set, the process will generate many page faults and run slowly.
Consider the writing of a program to fill a large two-dimensional array with zeroes.

Let’s write this program in two ways, one filling the array in row-major order (program A) and the other filling the array in column-major order (program B).

Then we’ll compare the execution of these programs in a paged virtual memory environment.
Program A – Row-Major Order

#define N 1024
char a[N][N];
main()
{
    int i, j;
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            a[i][j] = 0;
}
Program A – Row-Major Order

```c
#define N 1024
char a[N][N];
main()
{
    int i, j;
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            a[j][i] = 0;
}
```
Comparing the Programs

Suppose we assume a page has 1024 bytes.

We have the following memory requirements for each program:

- $1024 \times 1024 = 1$MB for the array $a$,
- probably one page for the code, and
- probably one page for the stack.

The program has a total of 1026 unique pages.
Consider the performance of program A with only 3 page frames of memory. It will incur a total of 1026 page faults, which is the minimum possible for this program.

What if program B was executed with only 3 page frames of memory? It would likely incur 1,048,578 page faults! Do you see why?

What is the working set size for program B?
The Working Set Principle

A process that spends most of its time waiting for paging operations to complete (likely doing disk I/O) is said to be thrashing.

The working set principle requires that a process have its working set in memory before allowing the process to run.

Prepaging may be beneficial if sufficient free page frames are available, as it reduces the average time required to load a page from disk (fewer seek operations are required).
Local versus Global Page Replacement Decisions

- From which process (or processes) should page frames be obtained when the supply of free page frames becomes low?
- There are a variety of answers, ranging from taking pages from the process that generated the fault, to taking pages from the global set of all processes.
When free page frames are obtained globally, every process potentially suffers for the performance of other processes that generate an excessive number of page faults.

Global algorithms do not require complex bookkeeping algorithms.
Local Page Replacement

- This approach appears to be fairer than global page replacement, since a process that generates lots of page faults pays for its behavior, not all the other processes in the system.

- Unfortunately, local page replacement is expensive to implement, and does not always yield expected performance benefits.
One way to tell if a process has a sufficient number of page frames is to record the number of page faults it generates during each time interval (e.g. one second).

If the rate becomes too high, the process might be prohibited from running until sufficient additional page frames are made available to it. If the rate becomes very low, the process may have excess, unneeded page frames which could be removed from memory, with the page frames they occupied being used for other processes.
An operating system may choose to use a logical page size that is a multiple of the physical page size.

For example, if the physical page size is 1024 bytes, the system might choose to use a logical page of 2048 bytes.

Each fault for either of the two related physical pages will cause the entire logical page to be loaded (into two physical page frames).

This technique is, in effect, similar to prepaging.
Physical Page Size Selection

- With a large page size
  - page tables will be smaller (fewer page table entries), and
  - there will be more internal fragmentation (on the average, half a page will be wasted at the end of the heap and the top of the stack).

- With a small page size
  - page tables will be larger (more page table entries), and
  - there will be less internal fragmentation.
In segmented virtual memory systems, the virtual memory for a process is divided into variable-sized components corresponding to the major functional pieces of the process (e.g. a function’s code or data, or a large data structure).

These components, called segments, are brought into memory, as necessary.

Segment placement in primary memory is done using the same techniques used in MVT. That is, a segment is essentially a variable-sized object that must be allocated a continuous region of memory.
Addresses in Segmented Systems

- A program running in a segmented system generates two-part addresses of the form `<segmentNumber,offset>`.
- The `segmentNumber` is an indirect reference (through a segment table) to the location where the segment is loaded.
- The `offset` is used in a manner similar to its use in paged systems. That is, it identifies the operand in the selected segment.
A segment table serves a function similar to that of a page table, but has a few differences.

- Instead of a page frame number, the memory address of a segment is included.
- Each entry contains the segment size, which is used to catch offsets that exceed the segment size. (Page tables dont need to specify a page size.)

Segment table entries, usually called segment descriptors, can be cached in associative memory, just like page table entries.
Simple Segmented System Execution

1. A process generates a reference to an address of the form \(<s,o>\).
2. \(s\) is used as an index to the segment table to obtain the appropriate segment table entry.
3. If the segment isn’t loaded, a segment fault occurs (and the OS loads the segment, adjusts the segment table, and the reference is repeated).
4. If the offset is larger than the segment size . . .
   4.1 then the reference is invalid (the segment could optionally dynamically grow to accommodate the offset).
   4.2 else the segment location and offset are added to yield the physical memory location for the reference.
Segment sharing is much easier than page sharing, since segments are logical entities.

Segment references in programs are usually symbolic, and linking can be done dynamically at execution time. Linking is usually done statically, before execution, in paged systems.

Some systems support paging of each segment. In these, the segment table entry points to a page table, which is then used to locate the appropriate page frame.
In paged systems, there is no external fragmentation, but there is an average of half a page internal fragmentation at the end of each logically-contiguous program region (e.g. heap and stack).

In segmented systems there is no internal fragmentation, but external fragmentation occurs as in MVT.
In the next module we’ll look at file systems and security.

We’ll examine the basic functions of a file system, and consider some of the design decisions that must be made.

Then we’ll consider the basics of security, and look at some techniques for providing security in a system.