Memory Management

Yücel Saygın

These slides are based on your text book and on the slides prepared by Andrew S. Tanenbaum
Memory Management

• Ideally programmers want memory that is
  – large
  – fast
  – non volatile

• Memory hierarchy
  – small amount of fast, expensive memory – cache
  – some medium-speed, medium price main memory
  – gigabytes of slow, cheap disk storage

• Memory manager handles the memory hierarchy
Three simple ways of organizing memory for an OS with one process.

- (a) was used for mainframes and minicomputers
- (b) is used on some palmtop computers and embedded systems
- (c) was used by early PCs running MS-DOS where BIOS device drivers were in ROM
Multiprogramming with Fixed Partitions
(used by OS/360 on large IBM mainframes, mostly for batch systems
not supported anymore)

• Fixed size memory partitions
  – Option 1: separate input queues for each partition
  – Option 2: single input queue

**Multiple Queues**: Put the job in The input queue for the smallest Partition large enough to hold it

**Single Queue**: When a partition becomes Available, choose the closest job in the queue that could fit or the largest job that can fit
Modeling Multiprogramming

CPU utilization as a function of number of processes in memory

Suppose that a process spends a fraction $p$ of its time waiting for I/O to complete with $n$ processes in memory at a time, the probability that all $n$ processes are waiting for I/O is $p^n$. CPU utilization is $1 - p^n$ (with the assumption that the processes are independent)
Analysis of Multiprogramming System Performance

- Arrival and work requirements of 4 jobs
- CPU utilization for 1 – 4 jobs with 80% I/O wait
- Sequence of events as jobs arrive and finish
  - note numbers show amount of CPU time jobs get in each interval
Relocation and Protection

• Cannot be sure where program will be loaded in memory
  – address locations of variables, code routines cannot be absolute
  – must keep a program out of other processes’ partitions

• Use base and limit values
  – address locations added to base value to map to physical addr
  – address locations larger than limit value is an error
Swapping (1)

Memory allocation changes as
- processes come into memory
- leave memory

Shaded regions are unused memory
Swapping

• Fragmentation: Small memory blocks that can not fit a reasonable size program
  – External fragmentation
  – Internal fragmentation

• What happens when the process needs more memory?
Swapping (2)

- Allocating space for growing data segment
- Allocating space for growing stack & data segment
Memory Management with Bit Maps

- Part of memory with 5 processes, 3 holes
  - tick marks show allocation units
  - shaded regions are free
- Corresponding bit map
- Same information as a list
Memory Allocation

- Memory allocation is done in chunks (memory units of fixed size)
- What is the tradeoff between having small and large chunks?
- In case of bitmaps for memory management:
  - Bitmap size depends on the size of the chunks when the memory size is fixed
  - Internal fragmentation
- In case of linked list based memory management
  - External fragmentation
Memory Allocation

• Bitmaps vs Linked Lists
• Searching bitmaps for k units of consecutive free memory chunks is time consuming
• Maintaining linked lists is hard
Memory Management with Linked Lists

Four neighbor combinations for the terminating process X

Before X terminates

(a)  
| A | X | B |

(b)  
| A | X |

(c)  
| X | B |

(d)  
| X |

After X terminates

becomes

(b)  
| A | B |

(c)  
| A |

(d)  
| B |
MM with Linked Lists:
Searching for available memory

• *First fit* is the simplest method that searches the linked list until it finds a hole that is big enough
  – Relatively fast

• *Next fit* is like first fit but it starts the search from where it left off last time (slightly worse than first fit as simulations showed)

• *Best fit* searches the entire list and takes the smallest hole large enough to fit the request. (slower and wastes a lot of memory, think why)

• *Worst fit* is the opposite of best fit
MM with Linked Lists:
Searching for available memory

- Keep separate lists for processes and holes to speed up the search (actually holes themselves can be used to form the data structure)
- Sort the list of holes wrt size to make best fit and first fit faster (next fit does not make sense in this case)
- Now extra overhead of maintaining two separate lists but it may be worth it
- Quick fit maintains separate lists of holes for some of the more common sizes requested
Buddy System (Used for Unix Memory Allocation)

- Overcomes the drawbacks of dynamic and fixed partitioning
  - Dynamic partitioning is complex to maintain and needs compaction due to external fragmentation
  - Fixed partitioning may use space inefficiently when process sizes and partition sizes do not match
- In buddy system memory blocks are of size $2^K$, $L \leq K \leq U$ where $2^L$ is the smallest size block that is allocated and $2^U$ is the largest size block allocated (usually entire memory)
- Initially the entire memory is treated as a single empty block
- If a request for a block of size $S$ arrives and $2^{(U-1)} < S \leq 2^U$, then the entire block is allocated, otherwise the block is split into two equal buddies of size $2^{(U-1)}$. And check is done recursively on a free buddy.
- The checking process continues until smallest block greater than or equal to $s$ is generated and allocated to the request.
- The system maintains a list of free blocks of each size $2^I$
- When a pair of buddies on the $i$ list both become free then they are removed from the $i$ list and combined into a single block and moved to $(i+1)$ list
### Example of Buddy System

1 Megabyte Block

<table>
<thead>
<tr>
<th>Request</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>100K (A)</td>
<td>128K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>240K (B)</td>
<td>128K</td>
<td>256K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256K (D)</td>
<td>128K</td>
<td>256K</td>
<td>256K</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Release</td>
</tr>
<tr>
<td>A</td>
<td>128K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>256K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>256K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Release</td>
</tr>
<tr>
<td>A</td>
<td>128K</td>
<td>256K</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Release</td>
</tr>
<tr>
<td>A</td>
<td>128K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>256K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>256K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Release</td>
</tr>
<tr>
<td>A</td>
<td>128K</td>
<td>256K</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Release</td>
</tr>
<tr>
<td>A</td>
<td>128K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>256K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>256K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>128K</td>
<td>256K</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Release</td>
</tr>
<tr>
<td>A</td>
<td>128K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>256K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>256K</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1M
Tree Representation of Buddy System
Addresses referenced in a program

- **Physical address**: (also called absolute address) is an actual location in main memory
- **Logical address**: is a reference to a memory location independent of the current assignment to memory. Must be translated to physical address before the actual memory reference is made
- **Relative address**: is a special case of a logical address where all the references are made relative to the beginning of the program.
Address Space Abstraction

- A process is provided with a linear address space that is used to read and write the array of bytes in the primary memory hardware.
- In early systems, the linear address space used by the programmer was the physical memory address space, and an executing program could directly access any primary memory address in the machine.
- Starting from the 60s, multiprogramming OSs introduced the address space abstraction for primary memory.
Paging

- Memory is arranged into fixed size chunks called **page frames**
- This scheme allows external fragmentation but external fragmentation is not a problem since nonconsecutive blocks of memory can be allocated for a process
Paging

- A process’ address space is divided into pages, a number of page frames are allocated in main memory to store the pages belonging to a process.
- Operating system maintains a page table for each process to keep track of which page frames are allocated for the process.
- OS also maintains a single list of free page frames.

![Diagram of Address Space and Physical Memory](image)
Memory addresses

• A memory address is expressed as a sequence of bits.
• For example if we have a very tiny memory of 4 bytes, and the memory is byte-addressable
• Question: How many bits are needed to address this memory?
Memory addresses

- A memory address is expressed as a sequence of bits.
- For example, if we have a very tiny memory of 4 bytes, and the memory is byte-addressable
- 2 bits are enough to address this tiny memory as shown below:
Memory addresses

• if we have a tiny memory of 16 bytes, and the memory is byte-addressable then we need to have 4 bits
• In general if we have $n$ bytes, then we need $\log_2 n$ bits to address this memory
Memory addresses with Paging

- A logical address is expressed as a page number and an offset.
- Now we need to set the page size, let's say 4 bytes.
- Then we will have 4 pages in this memory that has a total of 16 bytes.
- How many bits are we going to need to address these 4 pages?

![Diagram showing memory addresses with paging](attachment://memory_addresses.png)
Memory addresses with Paging

00  Page 0

01  Page 1

10  Page 2

11  Page 3

0000
0001
0100
0101
0010
0011
0110
0111
0100
0101
1000
1001
1010
1011
1100
1101
1110
1111
Memory addresses with Paging

- **Page 0**
  - 00
  - 0000
  - 0001
  - 0010
  - 0011

- **Page 1**
  - 01
  - 0100
  - 0101
  - 0110
  - 0111

- **Page 2**
  - 10
  - 1000
  - 1001
  - 1010
  - 1011
  - 1100

- **Page 3**
  - 11
  - 1100
  - 1101
  - 1110
  - 1111
Memory addresses with Paging

![Diagram showing page 1 with addresses 0100, 0101, 0110, and 0111]
Memory addresses with Paging

01  Page 1

00
01
10
11
Memory addresses with Paging

- **Example:**
  - assume 16 bit addresses are used and the page size is 1K = 1024 bytes
- **Questions**
  - How many pages do we have in the memory?
  - How many bits are needed to address a page?
  - How many bits are needed to address a specific byte within a page?
Memory addresses with Paging

• **Example:**
  – assume 16 bit addresses are used and the page size is 1K = 1024 bytes
  – Another question: Consider the logical address 1502 whose binary representation is (0000010111011110)
    • What is the page number referred by this address?
    • What is the offset?
Memory addresses with Paging

• **Example:**
  
  – assume 16 bit addresses are used and the page size is 1K = 1024 bytes
  
  – A logical address 1502 (0000010111011110) has 10 bits to represent the offset in the page (0111011110) and the most significant part after the offset bits represent the page number (000001).
Virtual Memory with Paging

- Address Space is the set of addresses that a process can reference.
- Virtual (logical) addresses are used by processes to reference memory.
- Memory Management Unit (MMU) maps virtual addresses to physical addresses.
Paging (2)

The relation between virtual addresses and physical memory addresses given by page table
Memory addresses with Paging

- Translation of the logical addresses to physical addresses:
  - A page table is used to store which pages are stored in which page frame.
  - The page number in the logical address is used as an index to the page table.
  - The corresponding page table entry contains the page frame number that stores the page.
  - Offset field of the logical address is used to find the offset within the page frame.

Internal operation of MMU with 16 4 KB pages
Page Tables (3)

Typical page table entry

- Present/absent bit keeps track of which pages are present in main memory
- If the present/absent bit is 0, then a **page fault** occurs and a trap to the operating system is caused which then brings the requested page from the disk to the main memory (does and I/O)
- Modified (or dirty) bit is used to decide if the page stored in the corresponding page frame needs to be written back to disk.
- Referenced bit is set when a page is referenced for reading or writing. It is used by page replacement algorithms
Where to store the page tables

• **Page tables could be stored in registers:**
  – When a process is started up, the OS loads the registers with the page table (from a copy in the main memory)
  – No more memory references are needed to access the page table during process execution
  – But it is expensive the load the whole page table into the registers (it could be very large)

• **Page tables can be stored entirely in main memory:**
  – A single register is needed to point to the start of the page table (to change the memory map when a context switch occurs)
  – One or more memory references are needed to read the page tables during the process execution

• **Usually none of the above is used in its pure form**
Page Tables (2)

- 32 bit address with 2 page table fields
- Two-level page tables
**TLBs – Translation Lookaside Buffers**

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virtual page</th>
<th>Modified</th>
<th>Protection</th>
<th>Page frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>140</td>
<td>1</td>
<td>RW</td>
<td>31</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>0</td>
<td>R X</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>130</td>
<td>1</td>
<td>RW</td>
<td>29</td>
</tr>
<tr>
<td>1</td>
<td>129</td>
<td>1</td>
<td>RW</td>
<td>62</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>0</td>
<td>R X</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>0</td>
<td>R X</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>860</td>
<td>1</td>
<td>RW</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>861</td>
<td>1</td>
<td>RW</td>
<td>75</td>
</tr>
</tbody>
</table>

- TLB concept is based on the observation that a large number of page references are done to a small number of pages. Therefore a small number of page table entries are heavily referenced.
- A small hardware component inside the MMU (called TLB) is used to map virtual addresses to physical addresses without going through the page table.
- When a virtual address is presented to the MMU for translation, the hardware first checks to see if the virtual page number is in TLB (all TLB entries are compared in parallel). If the page number is found and the access does not violate the protection then the page frame number is directly taken.
TLBs – Translation Lookaside Buffers

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virtual page</th>
<th>Modified</th>
<th>Protection</th>
<th>Page frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>140</td>
<td>1</td>
<td>RW</td>
<td>31</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>0</td>
<td>R X</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>130</td>
<td>1</td>
<td>RW</td>
<td>29</td>
</tr>
<tr>
<td>1</td>
<td>129</td>
<td>1</td>
<td>RW</td>
<td>62</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>0</td>
<td>R X</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>0</td>
<td>R X</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>860</td>
<td>1</td>
<td>RW</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>861</td>
<td>1</td>
<td>RW</td>
<td>75</td>
</tr>
</tbody>
</table>

- TLB fault occurs when the corresponding virtual page number is not found in the TLB then depending on the system, either
  - the MMU detects the miss and does a page table lookup to bring the page table entry to the TLB
  - or MMU generates a TLB fault and passes the problem of bringing the page table entry to TLB
Inverted Page Tables

Comparison of a traditional page table with an inverted page table
Page Replacement Algorithms

• Page fault forces choice
  – which page must be removed
  – make room for incoming page

• Modified page must first be saved
  – unmodified just overwritten

• Better not to choose an often used page
  – will probably need to be brought back in soon
Optimal Page Replacement Algorithm

• Replace page needed at the farthest point in future
  – Optimal but unrealizable

• Estimate by …
  – logging page use on previous runs of process
  – although this is impractical
Not Recently Used Page Replacement Algorithm

• Each page has Reference bit, Modified bit
  – bits are set when page is referenced, modified

• Pages are classified
  1. not referenced, not modified
  2. not referenced, modified
  3. referenced, not modified
  4. referenced, modified

• NRU removes page at random
  – from lowest numbered non empty class
FIFO Page Replacement Algorithm

- Maintain a linked list of all pages
  - in order they came into memory

- Page at beginning of list replaced

- Disadvantage
  - page in memory the longest may be often used
Second Chance Page Replacement Algorithm

Page loaded first

- Operation of a second chance
  - pages sorted in FIFO order
  - Page list if fault occurs at time 20, A has R bit set (numbers above pages are loading times)
The Clock Page Replacement Algorithm

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
Least Recently Used (LRU)

• Assume pages used recently will used again soon (Locality of reference)
  – throw out page that has been unused for longest time

• Must keep a linked list of pages
  – most recently used at front, least at rear
  – update this list every memory reference !!

• Alternatively you can implement LRU by a special hardware that maintains a 64 bit counter and stores the current counter value for the page when referenced
  – choose page with lowest counter value
  – periodically zero the counter
Simulating LRU in Hardware

• For a machine with $n$ page frames, the LRU hardware can maintain a matrix of $n \times n$ bits
• Initially all of them are 0
• Whenever a page frame $k$ is referenced, the hardware first sets all the bits of row $k$ to 1, then sets all the bits of column $k$ to 0.
• At any instant, the row whose binary value is lowest is the least recently used.
Simulating LRU in Hardware

LRU using a matrix – pages referenced in order 0,1,2,3,2,1,0,3,2,3
Simulating LRU in Software

• When a machine does not have special hardware, then LRU can be implemented in software

• A version of LRU is called NFU (Not Frequently Used)
  – Requires a software counter associated with each page, which is initially 0
  – At each clock interrupt, the OS scans all the pages in memory, and for each page, R bit is added to the counter.
  – When a page fault occurs, page with the lowest counter value is chosen for replacement

• One drawback of the algorithm is that it never forgets!
Simulating LRU in Software

• NRU with aging is used to overcome this drawback
• With aging, counters are shifted one position to the right, and R bit is added to the most significant bit
• Again, when page fault occurs, the page with the lowest counter value is chosen for replacement
Simulating LRU in Software

(a)

Page
0  10000000
1  00000000
2  10000000
3  00000000
4  10000000
5  10000000

R bits for pages 0-5, clock tick 0
1 0 1 0 1 1
Simulating LRU in Software

<table>
<thead>
<tr>
<th>Page</th>
<th>R bits for pages 0-5, clock tick 0</th>
<th>R bits for pages 0-5, clock tick 1</th>
<th>R bits for pages 0-5, clock tick 2</th>
<th>R bits for pages 0-5, clock tick 3</th>
<th>R bits for pages 0-5, clock tick 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000000</td>
<td>11000000</td>
<td>11100000</td>
<td>11110000</td>
<td>01111000</td>
</tr>
<tr>
<td>1</td>
<td>00000000</td>
<td>10000000</td>
<td>11000000</td>
<td>01100000</td>
<td>10110000</td>
</tr>
<tr>
<td>2</td>
<td>10000000</td>
<td>01000000</td>
<td>00100000</td>
<td>00100000</td>
<td>10001000</td>
</tr>
<tr>
<td>3</td>
<td>00000000</td>
<td>00000000</td>
<td>10000000</td>
<td>01000000</td>
<td>00100000</td>
</tr>
<tr>
<td>4</td>
<td>10000000</td>
<td>11000000</td>
<td>01100000</td>
<td>10110000</td>
<td>01011000</td>
</tr>
<tr>
<td>5</td>
<td>10000000</td>
<td>01000000</td>
<td>10100000</td>
<td>01010000</td>
<td>00101000</td>
</tr>
</tbody>
</table>

(a)  | (b)                                | (c)                                | (d)                                | (e)                                |
The Working Set Page Replacement Algorithm (1)

- Demand paging: pages are loaded to MM when needed
- Processes do not reference all the pages in their address space in sequence, instead **locality of reference** is observed
- Locality of reference means that during any phase of execution, the process references only a relatively small fraction of the pages in its address space
- The set of pages that a process is currently using is called its **working set**, and it slowly changes over time as the process goes into a new execution phase.
- If the entire working set is in memory then very few page faults will occur
- **Prepaging** (i.e., loading pages in the working set to MM before letting the process run) is used to reduce the initial page faults.
The Working Set Page Replacement Algorithm (1)

• What happens if the MM is small and the working set cannot fit in MM?
The Working Set Page Replacement Algorithm (1)

- I : 1 2 3 4 5 6 7 8 9 10 11 12 13 14
- P: 3 5 2 9 7 3 2 1 2 3 5 2 3 1
The Working Set Page Replacement Algorithm (1)

- The working set is the set of pages used by the $k$ most recent memory references
- $w(k,t)$ is the size of the working set at time, $t$
The Working Set Page Replacement Algorithm (2)

The working set algorithm

- **Information about one page**
  - Time of last use
    - 1980
  - Page referenced during this tick
    - 1213: 0
    - 2014: 1
  - Page not referenced during this tick
    - 2020: 1
    - 2032: 1
    - 1620: 0

- **Scan all pages examining R bit**:
  - if \( R = 1 \)
    - set time of last use to current virtual time
  - if \( R = 0 \) and \( \text{age} > \tau \)
    - remove this page
  - if \( R = 0 \) and \( \text{age} \leq \tau \)
    - remember the smallest time

- **Current virtual time**: 2204
The Working Set Page Replacement Algorithm (2)

• Working sets can be maintained by shift registers and duplicate elimination
• Use a shift register of size k (i.e., size of the working set)
• With every memory reference shift the registers one position and insert the most referenced page number to the register
• The set of k page number in the shift register is the working set
• However it is hard to maintain the shift register at every page fault so an approximation based on the virtual time of the process is used.
The WSClock Page Replacement Algorithm

Each entry contains: time of last use together with the R and M bits (M bit is not shown here).

Page repl alg:
1) If the R bit of the page is set to 1, then R bit is set to 0 (no good candidate for page repl.)
2) If the R bit is 0, then
   - if the age is > T and the page is clean, then it is replaced. If the
   - if the age is > T and the page is dirty, then a write to disk is scheduled and the clock head advances one position.

lock algorithm
## Review of Page Replacement Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>Not implementable, but useful as a benchmark</td>
</tr>
<tr>
<td>NRU (Not Recently Used)</td>
<td>Very crude</td>
</tr>
<tr>
<td>FIFO (First-In, First-Out)</td>
<td>Might throw out important pages</td>
</tr>
<tr>
<td>Second chance</td>
<td>Big improvement over FIFO</td>
</tr>
<tr>
<td>Clock</td>
<td>Realistic</td>
</tr>
<tr>
<td>LRU (Least Recently Used)</td>
<td>Excellent, but difficult to implement exactly</td>
</tr>
<tr>
<td>NFU (Not Frequently Used)</td>
<td>Fairly crude approximation to LRU</td>
</tr>
<tr>
<td>Aging</td>
<td>Efficient algorithm that approximates LRU well</td>
</tr>
<tr>
<td>Working set</td>
<td>Somewhat expensive to implement</td>
</tr>
<tr>
<td>WSClock</td>
<td>Good efficient algorithm</td>
</tr>
</tbody>
</table>
Modeling Page Replacement Algorithms

Belady's Anomaly

- FIFO with 3 page frames
- FIFO with 4 page frames
- P's show which page references show page faults
Modeling Page Replacement Algorithms

Belady's Anomaly

- FIFO with 3 page frames
- FIFO with 4 page frames
- $P$'s show which page references show page faults
Modeling Page Replacement Algorithms

Belady's Anomaly

- FIFO with 3 page frames
- FIFO with 4 page frames
- P's show which page references show page faults

![Diagram showing page replacement algorithms and Belady's Anomaly](image)

(a) All pages frames initially empty

(b)
Modeling Page Replacement Algorithms

Belady's Anomaly

- FIFO with 3 page frames
- FIFO with 4 page frames

- $P$'s show which page references show page faults
Modeling Page Replacement Algorithms
Belady's Anomaly

- FIFO with 3 page frames
- FIFO with 4 page frames
- \( P \)'s show which page references show page faults
Modeling Page Replacement Algorithms

Belady's Anomaly

- FIFO with 3 page frames
- FIFO with 4 page frames
- P's show which page references show page faults

<table>
<thead>
<tr>
<th>Youngest page</th>
<th>Oldest page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 0 1 4 4 4 2 3 3</td>
<td>0 1 2 3 0 1 1 1 4 2 2</td>
</tr>
<tr>
<td>0 1 2 3 0 0 0 1 4 4</td>
<td>9 Page faults</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>Youngest page</th>
<th>Oldest page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 3 3</td>
<td>0 1 2 2</td>
</tr>
<tr>
<td>0 1 2 2</td>
<td>0 1 1</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
</tr>
</tbody>
</table>

(b)
Modeling Page Replacement Algorithms
Belady's Anomaly

- FIFO with 3 page frames
- FIFO with 4 page frames
- \( P \)'s show which page references show page faults
Modeling Page Replacement Algorithms

Belady's Anomaly

- FIFO with 3 page frames
- FIFO with 4 page frames
- P's show which page references show page faults
Modeling Page Replacement Algorithms

Belady's Anomaly

- FIFO with 3 page frames
- FIFO with 4 page frames
- \( P \)'s show which page references show page faults

---

<table>
<thead>
<tr>
<th>Youngest page</th>
<th>Oldest page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 0 1 4 4 4 2 3 3</td>
<td>0 1 2 3 0 1 1 1 4 2 2</td>
</tr>
<tr>
<td>0 1 2 3 0 0 0 1 4 4</td>
<td>9 Page faults</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>Youngest page</th>
<th>Oldest page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 3 3 3 4 0</td>
<td>0 1 2 2 2 3 4</td>
</tr>
<tr>
<td>0 1 1 1 2 3</td>
<td>0 0 0 1 2</td>
</tr>
</tbody>
</table>

(b)
Modeling Page Replacement Algorithms

Belady's Anomaly

<table>
<thead>
<tr>
<th>Youngest page</th>
<th>Oldest page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 0 1 4 4 4 2 3 3</td>
<td>0 1 2 3 0 1 1 1 4 2 2</td>
</tr>
<tr>
<td>0 1 2 3 0 0 0 0 1 4 4</td>
<td></td>
</tr>
</tbody>
</table>

- FIFO with 3 page frames
- FIFO with 4 page frames
- P's show which page references show page faults
Modeling Page Replacement Algorithms
Belady's Anomaly

- FIFO with 3 page frames
- FIFO with 4 page frames
- \( P \)'s show which page references show page faults
Paging Systems

• Paging systems can be characterized by three items
  1. The reference string of the executing process
  2. The page replacement algorithm
  3. The number of page frames available in memory, m.
### Stack Algorithms

State of memory array, $M$, after each item in reference string is processed

| Reference string | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 7 | 4 | 7 | 3 | 3 | 5 | 5 | 3 | 1 | 1 | 1 | 7 | 1 | 3 | 4 | 1 |
|                  | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 7 | 4 | 7 | 3 | 3 | 5 | 5 | 3 | 1 | 1 | 1 | 7 | 1 | 3 | 4 | 1 |
|                  | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 7 | 4 | 7 | 3 | 3 | 5 | 3 | 3 | 1 | 7 | 1 | 3 | 4 | 1 |
|                  | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 3 | 4 | 7 | 7 | 7 | 5 | 5 | 5 | 3 | 3 | 7 | 1 | 3 | 4 |
|                  | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 6 | 6 | 6 | 4 | 4 | 7 | 7 | 7 | 5 | 5 | 5 | 7 | 7 | 7 | 5 |
|                  | 0 | 2 | 1 | 1 | 5 | 5 | 5 | 5 | 5 | 6 | 6 | 4 | 4 | 4 | 4 | 4 | 4 | 5 | 5 | 5 | 6 | 6 |
|                  | 0 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
|                  | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
|                  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Distance string  | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | 4 | ∞ | 4 | 2 | 3 | 1 | 5 | 1 | 2 | 6 | 1 | 1 | 4 | 2 | 3 | 5 | 3 |
Stack Algorithms Cont’d

• Stack algorithms have the following property:
  – The set of pages included in the top part of M for a memory with m page frames after r memory references are also included in M for a memory with m+1 page frames.

• The algorithms that have this property, i.e., stack algorithms do not suffer from Belady’s anomaly.
The Distance String

Probability density functions for two hypothetical distance strings
The Distance String

- Computation of page fault rate from distance string
  - the $C$ vector
  - the $F$ vector

\begin{itemize}
\item \[C_1 = 4\]
\item \[C_2 = 2\]
\item \[C_3 = 1\]
\item \[C_4 = 4\]
\item \[C_5 = 2\]
\item \[C_6 = 2\]
\item \[C_7 = 1\]
\item \[C_\infty = 8\]
\end{itemize}

\begin{itemize}
\item \[F_1 = 19\]
\item \[F_2 = 17\]
\item \[F_3 = 16\]
\item \[F_4 = 12\]
\item \[F_5 = 10\]
\item \[F_6 = 10\]
\item \[F_7 = 8\]
\item \[F_\infty = 8\]
\end{itemize}

\begin{itemize}
\item \[C_2 + C_3 + C_4 + \ldots + C_\infty\]
\item \[C_3 + C_4 + C_5 + \ldots + C_\infty\]
\item \[C_4 + C_5 + C_6 + \ldots + C_\infty\]
\item \[\# \text{ of page faults with 5 frames}\]
\end{itemize}
Design Issues for Paging Systems
Local versus Global Allocation Policies (1)

- Original configuration
- Local page replacement (Each process is assigned a fixed number of page frames)
- Global page replacement (Num of page frames of each process is dynamic)
- A hybrid approach would be to adopt the num of pages assigned by looking at page fault frequency (PFF)
Local versus Global Allocation Policies (2)

Page fault rate as a function of the number of page frames assigned

Page fault rate as a function of the number of page frames assigned
Load Control

• Despite good designs, system may still thrash

• When PFF algorithm indicates
  – some processes need more memory
  – but no processes need less

• Solution:
  Reduce number of processes competing for memory
  – swap one or more to disk, divide up pages they held
  – reconsider degree of multiprogramming
Page Size (1)

Small page size

• Advantages
  – less internal fragmentation

• Disadvantages
  – programs need many pages, larger page tables
  – Transferring a small page from disk to memory is same as transferring a large page due to seek time and rotational delay.
Page Size (2)

- Overhead due to page table and internal fragmentation

\[
\text{overhead} = \frac{s \cdot e}{p} + \frac{p}{2}
\]

- Where
  - \( s \) = average process size in bytes
  - \( p \) = page size in bytes
  - \( e \) = page entry

Optimized when
\[
p = \sqrt{2se}
\]
Separate Instruction and Data Spaces

- One address space
- Separate I and D spaces
Two processes sharing same program sharing its page table
Cleaning Policy

• Need for a background process, paging daemon
  – periodically inspects state of memory

• When too few frames are free
  – selects pages to evict using a replacement algorithm

• It can use same circular list (clock)
  – as regular page replacement algorithm but with different pointers for clean and dirty pages
Implementation Issues
Operating System Involvement with Paging

Four times when OS involved with paging

1. **Process creation**
   - determine program size
   - create page table

2. **Process execution**
   - MMU reset for new process
   - TLB flushed

3. **Page fault time**
   - determine virtual address causing fault
   - swap target page out, needed page in

4. **Process termination time**
   - release page table, pages
Page Fault Handling (1)

1. Hardware traps to kernel
2. General registers saved
3. OS determines which virtual page needed
4. OS checks validity of address, seeks page frame
5. If selected frame is dirty, write it to disk
Page Fault Handling (2)

6. OS brings schedules new page in from disk
7. Page tables updated
   - Faulting instruction backed up to when it began
6. Faulting process scheduled
7. Registers restored
   - Program continues
Instruction Backup

MOVE.L #6(A1), 2(A0)

An instruction causing a page fault
Locking Pages in Memory

• Virtual memory and I/O occasionally interact
• Proc issues call for read from device into buffer
  – while waiting for I/O, another processes starts up
  – has a page fault
  – buffer for the first proc may be chosen to be paged out
• Need to specify some pages locked
  – exempted from being target pages
The SWAP Area

• Swap area is the disk space allocated for storing process images

• Two approaches to implement the swap area:
  – Static swap area: contains the images of all processes
  – Dynamic swap area: contains images of swapped out processes only
(a) Paging to static swap area
(b) Backing up pages dynamically
Separation of Policy and Mechanism

Page fault handling with an external pager
Segmentation (1)

- One-dimensional address space with growing tables
- One table may bump into another
Segmentation (2)

Allows each table to grow or shrink, independently
## Segmentation (3)

### Comparison of paging and segmentation

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need the programmer be aware that this technique is being used?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>How many linear address spaces are there?</td>
<td>1</td>
<td>Many</td>
</tr>
<tr>
<td>Can the total address space exceed the size of physical memory?</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can procedures and data be distinguished and separately protected?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Can tables whose size fluctuates be accommodated easily?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Is sharing of procedures between users facilitated?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Why was this technique invented?</td>
<td>To get a large linear address space without having to buy more physical memory</td>
<td>To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection</td>
</tr>
</tbody>
</table>
Implementation of Pure Segmentation

(a)-(d) Development of checkerboarding
(e) Removal of the checkerboarding by compaction
Segmentation with Paging: MULTICS (1)

- Descriptor segment points to page tables
- Segment descriptor – numbers are field lengths
Segmentation with Paging: MULTICS (2)

A 34-bit MULTICS virtual address
Segmentation with Paging: MULTICS (3)

Conversion of a 2-part MULTICS address into a main memory address
Segmentation with Paging: MULTICS (4)

- Simplified version of the MULTICS TLB
- Existence of 2 page sizes makes actual TLB more complicated

<table>
<thead>
<tr>
<th>Comparison field</th>
<th>Segment number</th>
<th>Virtual page</th>
<th>Page frame</th>
<th>Protection</th>
<th>Age</th>
<th>Is this entry used?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>Read/write</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>Read only</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>3</td>
<td>1</td>
<td>Read/write</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>Execute only</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>12</td>
<td>Execute only</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>
• Pentium has 16K independent segments
• Two segment tables:
  1. GDT (Global Descriptor Table) : for system segments, including the OS
  2. LDT (Local Descriptor Tables) : For segments local to each program
• To access a segment, a Pentium program first loads a selector for that segment into one of the machine’s segment registers (ex: CS, DS registers)
• How many segments descriptors exist in LDT?
Segmentation with Paging: Pentium (2)

- Pentium code segment descriptor
- Data segments differ slightly
Segmentation with Paging: Pentium (3)

Conversion of a (selector, offset) pair to a linear address
Segmentation with Paging: Pentium (4)

Mapping of a linear address onto a physical address
Segmentation with Paging: Pentium (5)

Protection on the Pentium