Lecture 6: Memory management
Linking and Loading
Contents

- Paging on demand
- Page replacement
- Algorithm LRU and its approximation
- Process memory allocation, problem of thrashing
- Linker vs. loader
- Linking the executable
- Libraries
- Loading executable
- ELF – UNIX format
- PE – windows program
- Dynamic libraries
Page fault

- With each page table entry a valid–invalid bit is associated (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries
- Example of a page table snapshot:

<table>
<thead>
<tr>
<th>Frame #</th>
<th>valid-invalid bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

- During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault
Paging techniques

- Paging implementations
  - *Demand Paging (Demand Segmentation*)
    - Lazy method, do nothing in advance
  - *Paging at process creation*
    - Program is inserted into memory during process start-up
  - *Pre-paging*
    - Load page into memory that will be probably used
  - *Swap pre-fetch*
    - With page fault load neighborhood pages
  - *Pre-cleaning*
    - Dirty pages are stored into disk
Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
  - Slow start of application

- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ page fault $\Rightarrow$ bring to memory

- Page fault solution
  - Process with page fault is put to waiting queue
  - OS starts I/O operation to put page into memory
  - Other processes can run
  - After finishing I/O operation the process is marked as ready
Steps in Handling a Page Fault

1. Trap
2. Page is on backing store
3. Operating system
4. Bring in missing page
5. Reset page table
6. Restart instruction

Load M
Locality In A Memory-Reference Pattern

<table>
<thead>
<tr>
<th>page numbers</th>
<th>memory address</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td></td>
</tr>
</tbody>
</table>

execution time
Locality principle

- Reference to instructions and data creates clusters
- Exists time locality and space locality
  - Program execution is (excluding jump and calls) sequential
  - Usually program uses only small number of functions in time interval
  - Iterative approach uses small number of repeating instructions
  - Common data structures are arrays or list of records in neighborhoods memory locations.
- It’s possible to create only approximation of future usage of pages
- Main memory can be full
  - First release memory to get free frames
Other paging techniques

- Improvements of demand paging
  - **Pre-paging**
    - Neighborhood pages in virtual space usually depend and can be loaded together — speedup loading
    - **Locality principle** — process will probably use the neighborhood page soon
    - Load more pages together
    - Very important for start of the process
    - Advantage: Decrease number of page faults
    - Disadvantage: unused page are loaded too
  - **Pre-cleaning**
    - If the computer has free capacity for I/O operations, it is possible to run copying of changed (dirty) pages to disk in advance
    - Advantage: to free page very fast, only to change validity bit
    - Disadvantage: The page can be modified in future - boondoggle
What happens if there is no free frame?

- Page replacement – find some page (victim) in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults

- Same page may be brought into memory several times
Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Some pages cannot be replaced, they are locked (page table, interrupt functions,…)
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk

- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory
- **Want lowest page-fault rate**
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
Page Replacement with Swapping

1. Swap out victim page.
2. Change to invalid.
3. Swap desired page in.
4. Reset page table for new page.
Graph of Page Faults Versus The Number of Frames
Algorithm First-In-First-Out (FIFO)

- 3 frames (memory with only 3 frames)

<table>
<thead>
<tr>
<th>Reference: 1 2 3 4 1 2 5 1 2 3 4 5</th>
<th>Frame number</th>
<th>Frame content</th>
<th>Page faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 1 4 4 4 5 5 5 5 5 5</td>
<td>9 Page faults</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2 2 2 1 1 1 1 3 3 3 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3 3 3 2 2 2 2 4 4 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 4 frames of memory

<table>
<thead>
<tr>
<th>Reference: 1 2 3 4 1 2 5 1 2 3 4 5</th>
<th>Frame number</th>
<th>Frame content</th>
<th>Page faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 1 1 1 1 5 5 5 5 5 4 4</td>
<td>10 Page faults</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2 2 2 2 2 2 1 1 1 1 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3 3 3 3 3 3 2 2 2 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4 4 4 4 4 4 3 3 3 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Beladyho anomalie (more frames – more page faults)

- FIFO – simple, not effective
  - Old pages can be very busy
Optimal algorithm

- **Victim** – Replace page that will not be used for longest period of time
- We need to know the future
  - Can be only predicted
- Used as comparison for other algorithms
- Example: memory with 4 frames
  - As example we know the whole future

<table>
<thead>
<tr>
<th>Reference:</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>Frame number</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>Frame content</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

6 Page faults
(The best possible result)
Least Recently Used

- Prediction is based on history
  - Assumption: Page, that long time was not used will be probably not used in future
- Victim – page, that was not used for the longest period
- LRU is considered as the best approximation of optimal algorithm
- Example: memory with 4 frames
- Best result 6 page faults, LRU 8 page faults, FIFO 10 page faults

<table>
<thead>
<tr>
<th>Reference:</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>Frame number</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Frame content</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Page faults</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Page faults</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
LRU – implementation

- It is not easy to implement LRU
  - The implementation should be fast
  - There must be CPU support for algorithm – update step cannot be solved by SW because it is done by each instruction (each memory reading)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement
Approximation of LRU

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists). We do not know the order, however.

- Second chance
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules
  - In fact it is FIFO with second chance
Algorithm Second Chance

Page fault test the frame that is pointed by clock arm.

Depend on access a-bit:

- if \( a=0 \):
  
  take this page as victim

- if \( a=1 \):
  
  turn \( a=0 \), and keep page in memory
  
  turn the clock arm forward

- if you have no victim do the same for the next page

Numerical simulation of this algorithm shows that it is really close to LRU
Modification LRU

- **NRU** – not recently used
  - Use a-bit and dirty bit d-bit
  - Timer regularly clean a-bit and therefore it is possible to have page with d-bit=1 and a-bit=0.
  - Select page in order (da): 00, 01, 10, 11
    - Priority of d-bit enable to spare disk operation and time

- **Ageing**
  - a-bit is regularly saved and old-values are shifted
  - Time window is limited by HW architecture
  - If the history of access to page is 0,0,1,0,1, then it corresponds to number 5 (00101)
  - The page with the smallest number well be removed
Counter algorithms

- **Reference counter**
  - Each frame has reference counter
    - For “swap-in” – the counter is set to 0
    - Each reference increments the counter

- **Algorithm LFU** *(Least Frequently Used)*
  - replaces page with smallest count

- **Algorithm MFU** *(Most Frequently Used)*
  - based on the argument that the page with the smallest count was probably just brought in and has yet to be used
Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another

Local replacement – each process selects from only its own set of allocated frames

Principles of frame allocation
  - Fixed allocation
    - Process receives fixed number of frames (Can be fixed for each process or can depends on it's virtual space size)
  - Priority allocation
    - Process with higher priority receives more frames to be able to run faster
    - If there is page fault process with higher priority gets frame from process with lower priority
Fixed Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – Allocate according to the size of process
- Example:

  - $s_i = \text{size of process } p_i$
  - $S = \sum s_i$
  - $m = \text{total number of frames}$
  - $a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

\[
\begin{align*}
\text{Example:} & \\
& m = 64 \quad s_1 = 10 \quad s_2 = 127 \\
& a_1 = \frac{10}{137} \times 64 \approx 5 \\
& a_2 = \frac{127}{137} \times 64 \approx 59
\end{align*}
\]
Dynamic Allocation

- Priority allocation
  - Use a proportional allocation scheme using priorities rather than size
  - If process \( P_i \) generates a page fault,
    - select for replacement one of its frames
    - select for replacement a frame from a process with lower priority number

- Working set
  - Dynamically detect how many pages is used by each process
Thrashing

If a process does not have “enough” pages, the page-fault rate is very high. This leads to:

- low CPU utilization
- operating system thinks that it needs to increase the degree of multiprogramming
- another process can be added to the system

Thrashing ≡ a process is busy swapping pages in and out
Working-Set Model

- How many pages process need?
- Working set define set of pages that were used by last N instructions
- Detection of space locality in process
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction
- $WSS_i (\text{working set of Process } P_i) =$ total number of pages referenced in the most recent $\Delta$ (varies in time)
  - if $\Delta$ too small will not encompass entire locality
  - if $\Delta$ too large will encompass several localities
  - if $\Delta = \infty \implies$ will encompass entire program
- $D = \sum WSS_i \equiv$ total demand frames
- if $D > m \implies$ Thrashing
- Policy if $D > m$, then suspend one of the processes
Working-set model

Page reference table

\[ \ldots 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 4 4 4 1 3 2 3 4 4 4 3 4 4 4 \ldots \]

\[ \Delta \]

\[ t_1 \]

\[ \Delta \]

\[ t_2 \]

\[ WS(t_1) = \{1,2,5,6,7\} \]

\[ WS(t_2) = \{3,4\} \]
Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units
Working set

- If sum of working sets for all process $P_i$ - $WS_i$ exceeds the whole capacity of physical memory it creates *thrashing*

- Simply protection before thrashing
  - Whole one process is swapped out
**Page size**

- **Big pages**
  - Small number of page faults
  - Big fragmentation
  - If page size is bigger then process size, virtual space is not necessary

- **Small pages**
  - Big number of small pages
    - Page is more frequently in memory → low number of page faults
  - Smaller pages means
    - Smaller fragmentation but decrease the effectiveness of disk operations
    - The bigger page table and more complicated selection of victim for swap out
  - Big page table
    - PT must be in memory, cannot be swapped out – PT occupying real memory
    - Placing part of PT into virtual memory leads to more page faults (access to invalid page can create 2 page faults, first fault of page table and fault of page)
Programming techniques and page faults

Programming techniques have influence to page faults
double data[512][512];
- Suppose that double occupy 8 byts
- Each line of array has 4 KB and is stored in one page 4 KB

Approach 1:
for (j = 0; j < 512; j++)
  for (i = 0; i < 512; i++)
    data[i][j] = i*j;

Can have
512 x 512 = 262 144
page faults

Approach 2:
for (i = 0; i < 512; i++)
  for (j = 0; j < 512; j++)
    data[i][j] = i*j;

Only 512 page faults

- It is good to know how the data are stored in virtual space
Paging in Windows XP

- Uses demand paging with pre-paging **clusters**. Clustering brings in pages surrounding the faulting page.
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum
- There can be thrashing
  - Recommended minimal memory size – 128 MB
  - Real minimal memory size – 384 MB
Linking and Loading
Background

- Operating system is responsible for starting programs
- Program must be brought into memory and placed within a process memory space for it to be executed
- User programs go through several steps before being run
- Linkers and loaders prepare program to execution
- Linkers and loaders enable to binds programmer’s abstract names to concrete numeric values – addresses
Linker vs. Loader

- Program loading – copy program from secondary storage into main memory so it’s ready to run
  - In some cases it is copying data from disk to memory
  - More often it allocate storage, set protections bits, arrange virtual memory to map virtual addresses to disk space

- Relocation
  - Each object code program address started at 0
  - If program contains multiple subprograms all subprograms must be loaded at non-overlapping addresses
  - In many systems the relocation is done more than once

- Symbol resolution
  - The reference from one subprogram to another subprogram is made by using symbols

- Linker and loader are similar
- Loader does program loading and relocation
- Linker does symbol resolution and relocation
- There exists linking loaders
Binding of Instructions and Data to Memory

- **Compile time**: If memory location is known a priori, *absolute code* can be generated; must recompile code if starting location changes.

- **Load time**: Must generate *relocatable code* if memory location is not known at compile time.

- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., *base* and *limit registers*).
Two pass linking

- Linker’s input is set of object files, libraries, and command files.
- Output of the linker is executable file, link/load map and/or debug symbol file
- Linker uses two-pass approach
- Linker first pass
  - Scan from input files segment sizes, definitions and references
  - Creates symbol table of definitions and references
  - Determine the size of joined segments
- Linker second pass
  - Assign numeric location to symbols in new segments
  - Reads and relocates the object code, substituting numeric address for symbol references
  - Adjusting memory address according new segments
  - Create execution file with correct:
    - Header information
    - Relocated segments
    - New symbol table information
    - For dynamic linking linker generates “stub” code or an array of pointers that need
Object code

Compilers and assemblers create object files from source files

Object files contains:
- Header information – overall information about file, like size of the code, size of the data, name of the source file, creation date
- Object code – binary instructions and data
- Relocation – list of places in object code, that have to be fixed up, when the linker or loader change the address of the object code
- Symbols – global symbols defined in this object file, this symbols can be used by other object files
- Debugging information – this information is optional, includes information for debugger, source file line numbers and local symbols, description of data structures
Library

- Library is sequence of object modules
- UNIX files use an “archive” format of file which can be used for collection of any types of files
- Linking library is iterative process:
  - Linker reads object files in library and looks for external symbols from program
  - If the linker finds external symbol it adds the concrete object file to program and adds external symbols of this library object to external symbols of program
  - The previous steps repeat until new external symbols and objects are added to program
- There can be dependencies between libraries:
  - Object A from lib A needs symbol B from lib B
  - Object B from lib B needs symbol C from lib A
  - Object C from lib A needs symbol D from lib B
  - Object D from lib B needs symbol E from ............
UNIX ELF

- Structure for object and executable programs for most UNIX systems
- Successor of more simple format a.out
- ELF structure is common for relocatable format (object files), executable format (program from objects), shared libraries and core image (core image is created if program fails)
- ELF can be interpreted as a set of sections for linker or set of segments for loader

ELF contains:
- ELF header – magic string `\177ELF`, attributes - 32/64 bit, little-endian/big-endian, type – relocatable/executable/shared/core image, architecture SPARC/x86/68K,….
- Data – list of sections and segments depending on ELF type
ELF relocatable

- Created by compiler and is prepared for linker to create executable program
- Relocatable files – collection of section defined in header. Each section is code, or read-only data, or rw data, or relocation entries, or symbols.
- Attribute alloc means that loader must allocate space for this section
- Sections:
  - .text – code with attribute alloc+exec
  - .data – data with initial value, alloc+write
  - .rodata – constants with only alloc attribute
  - .bss – not initialized data – nobits, alloc+write
  - .rel.text, .rel.data, .rel.rodata – relocation information
  - .init – initialization code for some languages (C++)
  - .symtab, .dynsym – linker symbol tables (regular or dynamic)
  - .strtab, .dynstr – table of strings for .symtab resp. .dynsym (.dynsym has alloc because it’s used at runtime)
ELF - executable

- Similar to ELF-relocatable but the data are arranged so that are ready to be mapped into memory and run
- Sections are packed into segments, usually code and read-only data into read-only segment and r/w data into r/w segment
- Segments are prepared to be loaded at defined address
- Usually it is:
  - Stack from 0x8000000
  - Text with ro-data from 0x8048000 – 0x48000 is stack size
  - Data behind text
  - Bss behind data
- Relocation is necessary if dynamic library is colliding with program – Relocated is dynamic library
- Segments are not align to page size, but the offset is used and some data are copied twice
Microsoft Portable Executable format

- Portable executable (PE) is Microsoft format for Win NT. It is mix of MS-DOS executable, Digital’s VAX VMS, and Unix System V. It is adapted from COFF, Unix format between a.out and ELF.
- PE is based on resources – cursors, icons, bitmaps, menus, fonts that are shared between program and GUI.
- PE is for paged environment, pages from PE can be mapped directly into memory.
- PE can be executable file (EXE) or shared libraries (DLL).
- PE starts with small DOS.EXE program, that prints “This program needs Microsoft Windows”.
- Then contains PE header, COFF header and “optional” headers.
- Each section is aligned to memory page boundary.
PE sections

- Each section has address in file and size, memory address and size (not necessarily same, because disk section use usually 512 bytes, page size 4kB)
- Each section is marked with hardware permissions, read, write, execute
- The linker creates PE file for a specific target address – imagebase
- If the address space is free than loader do no relocation
- Otherwise (in few cases) the loader has to map the file somewhere else
- Relocation is done by fix-ups from section .reloc. The PE is moved as block, each pointer is shifted by fixed offset (target address – image address). The fix-up contains position of pointer inside page and type of the pointer.
- Other sections – Exports (mainly for DLL, EXE only for debugging), Imports (DLL that PE needs), Resources (list of resources), Thread Local Storage (Thread startup data)
Shared libraries - static

- It is efficient to share libraries instead linking the same library to each program.
- For example, probably each program uses function printf and if you have thousands of programs in computer there will be thousands of copy printf function.
- The linker search library as usual to find modules that resolve undefined external symbols. Rather than coping the contents of module into output file it creates the table of libraries and modules into executable.
- When the program is started the loader finds the libraries and map them to program address space.
- Standards systems shares pages that are marked as read-only.
- Static shared libraries must used different address.
- Assigning address space to libraries is complicated.
Dynamic Libraries

- Dynamic Libraries can be relocated to free address space
- Dynamic Libraries are easier to update. If dynamic library is updated to new version the program has no change
- It is easy to share dynamic libraries
- Dynamic linking permits a program to load and unload routines at runtime, a facility that can otherwise be very difficult to provide
- Routine can be loaded when it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
ELF dynamic libraries

- ELF dynamic libraries can be loaded at any address, it uses position independent code (PIC)
- Global offset table (GOT) contains pointer to all static data referenced in program
- Lazy procedure linkage with Procedure Linkage Table (PLT)
  - For each dynamic function PLT contain code that use GOT to find address of this function
  - At program load all addresses point to stub – dynamic loader
  - After loading dynamic library entry in GOT is changed to real routine address
- Dynamic loader (library ld.so) finds the library by library name, major and minor versions numbers. The major version number guarantee compatibility, the minor version number should be the highest.
- Dynamic loading can be run explicitly by dlopen(), dlsym(), … functions
Dynamic Linking Libraries - DLL

- Similar to ELF dynamic libraries
- Dynamic linker is part of the windows kernel
- DLL is relocated if the address space is not free (windows call it rebasing)
- Lazy binding postpones binding until execution time
- Each function exported by DLL is identified by a numeric ordinal and by name
- Addresses of functions are defined in Export Address table
Architectural Issues

- Linkers and loaders are extremely sensitive to the architectural details of CPU and OS
- Mainly two aspects of HW architecture affect linkers
  - Program addressing
  - Instruction format

- Position independent code – enable to implement dynamic libraries
  - Separate code from data and generate code, that won’t change regardless of the address at which it is loaded
  - ELF – PIC group of code pages followed by group of data pages
  - Regardless of where the in the address space the program is loaded, the offset from the code to the data doesn’t change
  - Linker creates Global Offset Table containing pointers to all of the global data
  - Advantage – no load relocation, share memory pages of code among processes even though they don’t have the same address
  - Disadvantage – code is bigger and slower than non-PIC
End of Lecture 5

Questions?