Address Translation

*Throughout the course we will use overheads that were adapted from those distributed from the textbook website. Slides are from the book authors, modified and selected by Jean Mayo, Shuai Wang and C-K Shene.

If a machine is expected to be infallible,
It cannot also be intelligent.

Alan Turing
Address generation has three stages:

- **Compile**: compiler
- **Link**: linker or linkage editor
- **Load**: loader
Three Address Binding Schemes

- **Compile Time**: If the compiler knows the location a program will reside, it can generate absolute code. Example: compile-go systems and MS-DOS `.COM`-format programs.

- **Load Time**: A compiler may not know the absolute address. So, the compiler generates *relocatable* code. Address binding is delayed until load time.

- **Execution Time**: If the process may be moved in memory during its execution, then address binding must be delayed until run time. This is the commonly used scheme.
Address Generation: Compile Time

Unresolved Address Table

<table>
<thead>
<tr>
<th>source</th>
<th>target</th>
<th>in which seg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>$j$</td>
<td>code</td>
</tr>
<tr>
<td>$k$</td>
<td>$y$</td>
<td>data</td>
</tr>
</tbody>
</table>

Diagram:
- Code segment:
  - JUMP X
  - ADD #4
  - LOAD Y
- Data segment:
  - 1234
- Unresolved Address Table:
  - $i$ to $j$ in code
  - $k$ to $y$ in data
Address Generation: Static Linking

0  code
  1000
  JUMP ?X
  ADD #4
  LOAD ?Y

0  data
  500
  1234

Unresolved Address Table

<table>
<thead>
<tr>
<th>source</th>
<th>target</th>
<th>which seg?</th>
</tr>
</thead>
<tbody>
<tr>
<td>i+1000</td>
<td>j+1000</td>
<td>code</td>
</tr>
<tr>
<td>k+1000</td>
<td>y+500</td>
<td>data</td>
</tr>
</tbody>
</table>
Code and data are loaded into memory at addresses 10000 and 20000, respectively.

Every unresolved address must be adjusted.
Main Points

- **Address Translation Concept**
  - How do we convert a virtual address to a physical address?

- **Flexible Address Translation**
  - Base and bound
  - Segmentation
  - Paging
  - Multilevel translation

- **Efficient Address Translation**
  - Translation Lookaside Buffers (TLB)
  - Virtually and physically addressed caches
Address Translation Goals

- Memory protection
- Memory sharing
  - Shared libraries, interprocess communication
- Sparse addresses
  - Multiple regions of dynamic allocation (heaps/stacks)
- Efficiency
  - Memory placement
  - Runtime lookup
  - Compact translation tables
- Portability
Goals: 1/4

- Memory Protection
- Memory Sharing
- Flexible Memory Placement
- Sparse Addresses
- Runtime Lookup Efficiency
- Compact Translation Tables
- Portability
Goals: 2/4

- Memory Protection
  - We need the ability to limit the access of a process to certain regions of memory

- Memory Sharing
  - We want to allow multiple processes to shared selected regions of memory (e.g., shared memory segments)

- Flexible Memory Placement
  - We want to allow the operating system the flexibility to place a process (and each part of a process) anywhere in physical memory.
Goals: 3/4

- **Sparse Addresses**
  - Many programs have multiple dynamic memory regions that can change (e.g., heap, stack, etc.). Modern processors have 64-bit address spaces, but making the address translation more complex.

- **Runtime Lookup Efficiency**
  - Hardware address translation occurs on every instruction fetch and every data load and save. Thus, translation has to be efficient and faster than the instructions.
Compact Translation Tables

We need some tables to aid address translation. These table data structures have to be compact enough to save memory.

Portability

Different hardware implementation use different choices to implement address translation. If an operating system is to be easily portable, it needs to be able hardware independent.
Logical, Virtual, Physical Address

- **Logical Address**: the address generated by the CPU.

- **Physical Address**: the address seen and used by the memory unit.

- **Virtual Address**: Run-time binding may generate different logical address and physical address. In this case, logical address is also referred to as virtual address. (Logical = Virtual in this course)
Address Translation Concept
Virtually Address Base and Bounds
Because executables may run in any area, relocation and protection are needed.

Recall the base/limit register pair for memory protection.

It could also be used for relocation if the linker generates executables starting from 0.

Linker generates relocatable code starting with 0. The base register contains the starting address.
Relocation and Protection: 2/2

- **limit**
  - logical address
  - yes
- **base**
  - physical address
  - no
  - not your space
  - traps to the OS
  - addressing error

**protection**

**relocation**

**memory**
Relocation: How does it work?

Actual address = \( a + x \)

process moved to a new address \( k \)

Actual address = \( k + x \)
Virtually Addressed Base and Bounds

**Pros?**
- Simple
- Fast (2 registers, adder, comparator)
- Safe
- Can relocate in physical memory without changing process

**Cons?**
- Can’t keep program from accidentally overwriting its own code
- Can’t share code/data with other processes
- Can’t grow stack/heap as needed
Segmentation: 1/6

- Segment is a contiguous region of virtual memory
- Each process has a segment table (in hardware)
  - Entry in table = segment
- Segment can be located anywhere in physical memory
  - Each segment has: start, length, access permission
- Processes can share segments
  - Same start, length, same/different access permissions
Each process has at four segments: code, data, heap and stack.

Each virtual address is divided into a segment # and an offset in that segment. Suppose we have 31-bit address, and this address is divided into 16-bit for offset and 15-bit for segment number. In this way, 16-bit offset means segment max. size is $2^{16} = 64K$ bytes and $2^{15}$ segments.
A process is divided into segments. The chunks that a program is divided into which are not necessarily of the same length.

Early systems (MULTICS and Burroughs B5700/B6700) used segmentation memory management.

Burroughs Corporation was founded in 1886, in 1986 merged with Sperry UNIVAC and renamed Unisys. In the 1970’sm Burroughs developed some large systems based on the block-based (i.e., ALGOL) languages.
Burroughs B5700/B6700 are interesting as their processors are designed around the language blocked-structured (e.g., ALGOL, PL/I, etc.).

Procedures can be declared as local procedures, which are called by the containing procedure.

Thus, procedures do have a tree structure.

Each procedure is in its own segment.

A hardware pointer \( ip \) is the program counter.

Another pointer \( EP \) points to the activation record on a stack.
There are two sets of pointers, one pointing to the executing code and the order to the its corresponding execution environment.

Each “procedure” is in its own segment, and the segment directory table has pointers to each segment.

The diagram shows 6 procedures and the ip pointer indicates that the processor is executing procedure C!

The ip (3,j) indicates the next instruction is in segment 3 and offset j.

There are two set of pointers, one pointing to the executing code and the order to the its corresponding execution environment.

Each “procedure” has its own “environment” (i.e., segment), and the segment Display points to the Location of its activation record.

This diagram shows the corresponding activation record on the stack.

The left diagram shows 5 procedures and the EP pointers indicate the processor is executing procedure C!
UNIX fork and Copy on Write

- UNIX fork
  - Makes a complete copy of a process

- Segments allow a more efficient implementation
  - Copy segment table into child
  - Mark parent and child segments read-only
  - Start child process; return to parent
  - If child or parent writes to a segment (ex: stack, heap)
    - trap into kernel
    - make a copy of the segment and resume