CSci 340
Foundations of Computer Systems

Christian Collberg
November 21, 2000

Dynamic Relocation

Copyright © 2000 C. Collberg

Slide 21–1

Some loaders will load a program into the same location in memory every time.

In the program on the next slide the executable file header contains
loadAddr The address in memory at which the program should be loaded, and
startAddr The address at which we should begin executing the program.

Two views of memory

Logical address space: This is memory from the program’s point of view.

Physical address space: This is memory from the machine’s point of view.

.data
arr .space 200
i: .word

.text
lw $7,1200($0)
li $8,1000
mul $9,$7,$2
add $8,$8,$9
sw $7,($8)
structure execFileHeader {
  unsigned int startAddr;
}

char* relativeLoader (FILE *execFile) {
  struct execHeader header = readHeader(execFile);
  char *byteAddr = LOAD_ADDR;
  for all bytes b in execFile do {
    *byteAddr = b;
    byteAddr++;
  }
  return (char*) header.startAddr;
}
Dynamic Relocation...

- Dynamic relocation is handled by hardware called the memory-management unit (MMU). The MMU sits between the CPU and the memory and relocates all addresses issued by the CPU.

Static Relocation – Problems

1. A relative loader must find a region of unused memory large enough to hold the program.
2. Same problem as heap allocation – use First-Fit, Best-Fit, etc.
3. Each segment (data, text, bss) can be relocated individually – allocate 3 small holes rather than 1 big one.
4. Can’t relocate program while it’s running ⇒ can’t compact free space.
5. Programs can interfere with one another and the operating system. A bug in one program can cause another to crash.

Dynamic Relocation...

- These days the MMU is on the CPU chip.
- The MMU translates each program-generated address (called a logical or virtual address) into a real or physical address before it is given to the memory system.
- Dynamic relocation happens on every memory reference.

Dynamic Relocation

- Suppose, however, instead of relocating a program’s addresses when it is loaded, we relocate them while it’s running. This is called dynamic relocation, and the idea is to relocate (change) the address on every memory access.
  1. Once again a problem is solved via another level of indirection, the computer scientist’s secret weapon.
  2. Think of it like call-forwarding – you call one number but your call is forwarded to another number.
Disadvantages of dynamic relocation

1. Addresses must be relocated on every memory access, slowing them down.
2. Relocation must also be done before accessing the caches.

Dynamic Relocation...

- Dynamic relocation leads to two views of memory, called address spaces.
  1. The virtual address space (VAS) is what the program sees; addresses in the program are virtual addresses. The text segment starts at virtual address 0, followed by the data segment and stack segment.
  2. The physical address space (PAS) is what the memory system provides; it starts at physical address 0 and goes up to the amount of memory in the machine (ignore memory-mapped devices for now).

Advantages of dynamic relocation

1. Each process starts at virtual address 0.
2. No relocation during loading
3. A program can be moved to another location in physical memory without changing its virtual addresses: Simply change the way MMU does virtual-to-physical mapping so that the virtual addresses now refer to the new physical addresses.
4. If we limit the size of a program’s VAS, programs cannot access each other’s memory. This prevents one program from crashing another.

```c
struct execFileHeader {
  unsigned int startAddr, textSize, dataSize, bssSize;
}

char* dynamicLoader (FILE *execFile) {
  struct execHeader header = readHeader(execFile);
  unsigned int pgmSize = header.textSize +
                          header.dataSize + header.bssSize;
  char* byteAddr = getMemory(pgmSize);
  for all bytes b in execFile do {
    *byteAddr = b;
    byteAddr++;
  }
  setMemoryMapping(byteAddr, pgmSize);
  return (char*) header.startAddr;
}
```
**Base & Bound Relocation**

- A simple dynamic relocation scheme uses an MMU that consists of two registers: a base register that contains the physical address (PA) of the start of the program, and a bound register that contains the last valid virtual address (VA) in the program’s VAS.

![Diagram of Base & Bound Relocation](image)

**Base & Bound Relocation...**

- Only the operating system (in supervisor mode) can modify the base and bound registers.
- Base & bound is cheap – only 2 registers – and fast – the add and compare can be done in parallel.
- Examples: CRAY-1

**Base & Bound Relocation...**

- On each memory reference, the VA is compared to the bound register, and added to the base register.
  1. If the VA > bound, a segmentation violation (fault) occurs.
  2. The PA = VA + base is passed to the memory system.
- Each program appears to have a completely private memory of size equal to the bound register plus 1. Programs are protected from each other. No static address relocation is necessary.
The segment table holds the bases and bounds for all the segments of a program:

Problems with base & bound relocation

1. Can’t grow existing program because stack grows downwards.
2. Only one segment. How can two processes share code while keeping private data areas (e.g. shared editor)?

Solution: multiple segments per program

1. Split program between several areas of physical memory, e.g. separate area for text, data, and stack.
2. Use a separate base and bound for each segment, and also add two protection bits (read and write).
3. Programs can share segments (e.g. share text).
4. Different segments can have different permissions (e.g. text is read-only).
240:    la     $a0, 0x1108
244:    jal    360
...
360:    subu   $sp, $sp, 24
364    sw     $fp, 0($sp)
368    addu   $fp, $sp, 24
36c    lw     $t0, 0($a0)
370    la     $a0, 0x3000
374    sw     $t0, 0($a0)
378    lw     $fp, -24($sp)
37c    addu   $sp, $sp, 24
380    jr      $ra
...
1108:    42

- Given a VA, which segment is it in? (which base/bound do we use?)
  1. Top bits of address select segment, low bits the offset (shown above). This is the most common, and the best.
  2. From operation underway (e.g. code vs. data segment).
  3. From segment register indicated by field in instruction.
- The last two are caused by too few bits in the virtual address space.

**Segmentation example...**

- What PA corresponds to VA 0x240?
  00  0010 0100 0000
  Seg=0 Offset=0010 0100 0000=0x240
  +  0x4000 (=base)
    0x4240

- What PA corresponds to VA 0x1108?
  01  0001 0000 1000
  Seg=1 Offset=0001 0000 1000=0x108
  +  0x0 (=base)
    0x108

**Segmentation example**

- 2-bit segment number, 12-bit offset, 16-bit PA.

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base (hex)</th>
<th>Bounds (hex)</th>
<th>RW (binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4000</td>
<td>6FF</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4FF</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>3000</td>
<td>FFF</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>00</td>
</tr>
</tbody>
</table>
Segmentation example...

- Suppose $sp$ is initially set to VA $0x2650$. What is the corresponding PA?

\[
\begin{array}{cccc}
10 & 0110 & 0101 & 0000 \\
\text{Seg}=2 & \text{Offset}=0110 & 0101 & 0000=0x650 \\
+ & 0x3000 (\text{=base}) \\
\hline
0x3650
\end{array}
\]

Readings and References