Segmentation and Paging

Recall the problems with base&bound relocation:

- Fragmentation of Memory: variable size holes
- Only one segment. How can two processes share code while keeping private data areas (e.g. shared editor)?

Solutions

- *Paging:* Divide physical and virtual memory into fixed size chunks (pages). Keep a table mapping virtual page (numbered consecutively from 0) to a physical page (arbitrary, not necessarily sequential).
- *Segmentation:* Processes split up into several logical areas of memory, e.g., code, data, stack. Some segments can be made read-only.

Multiple segments (segmentation)

Permit process to be split between several areas of memory, called *segments.*

Use a separate base and bound for each segment, and also add two protection bits (read and write).

Segment	**Base**	**Bounds**	**RW**
0	4000	677	10
1	0000	477	11
2	3000	777	11
3	XXX	XXX	00

Each memory reference indicates a segment and offset:

- Top bits of address select segment, low bits the offset.

segment #	offset

Segment table holds the bases and bounds for all the segments of a process. What happens on a context switch?

Memory mapping procedure consists of a table lookup + add + compare. (Draw a picture)

Segmentation example: 2-bit segment number, 9-bit offset.

Segment	Base	Bounds	RW
0	4000	677	10
1	0000	477	11
2	3000	777	11
3	XXX	XXX	00

Main program (addresses are virtual, octal):

```
240:  put #1106 in R2
244:  CALL SIN
250:  ...
```

SIN procedure (addresses are virtual and byte addressing is used):

```
360:  move contents of address stored in R2 to R3.
      ...
      RETURN
```

- Where is 240 in physical memory?
- Suppose the Stack Pointer is initially 2650. Where is it in physical memory?

- Where is 1106 in physical memory?

- Which portions of the virtual and physical address spaces are used by this process?

- Simulate the call to SIN; make sure that the data value really gets picked up from the right place and that the procedure returns correctly.

- Do segments have to start on even 000 physical addresses?

- What if hardware provides eight segments but your operating system (e.g. UNIX) only uses two or three?

Segmentation pros:

- o Segments can be shared, swapped, and assigned to storage independently. Show an example where code is shared between two processes safely.

Assume that code is segment 0 for both P1 and P2 and we would like them to share the code in physical memory, but not share data (e.g., two users using the gcc library).

Similar ideas apply to sharing libraries, so that only one copy of large libraries (e.g., X windows) is needed in physical memory when running multiple applications that use the library.

Segmentation Cons:

(This problem also applies to base and bound schemes)

Paging

The goal of paging is to make allocation and swapping easier, and to reduce memory fragmentation.

Make all chunks of memory the same size, call them *pages*. Typical sizes range from 512 to 8k bytes.

For each process, a page table defines the base address of each of that process' pages along with read-only and valid bits.

Translation process: page number always comes directly from the virtual address. Since page size is a power of two, no comparison or addition is necessary. Just do a table lookup and bit substitution. (Show a picture)

Pros:

- Easy to allocate: keep a free list of available pages and grab the first one. Easy to swap since everything is the same size, which is usually the same size as disk blocks to and from which pages are swapped.

Cons:

- Efficiency of access: even small page tables are generally too large to load into fast memory in the relocation box. Instead, page tables are kept in main memory and the relocation box only has the page table's base address. It thus takes one overhead reference for every real memory reference.

  Solution from EECS 370 to get access time back to close to one real memory reference?

- Table space: if pages are small, the table space could be substantial. In fact, this is a problem even for normal page sizes: consider a 32-bit address architecture with 1KB pages. What if the whole table has to be present at once? How large will the page table for each process be, assuming that each page table entry is 4 bytes long?

  What if there are 100 such processes on the system? How much memory is taken up by page tables?

  Solution: We will see a solution to this problem later when we discuss virtual memory.

- Internal fragmentation: page size doesn't match up with information size. The larger the page, the worse this is. But no external fragmentation.

Assuming 3 partial pages per process, a page size of 4K, and 100 processes, how much space is wasted due to internal fragmentation? Is it significant?

- Can we share pages between processes?

- Can we distinguish between code, data, and stack?