Lecture IX
Memory Management

§1. Simple Model of Memory Allocation

- Programs must be stored in MM for execution. These programs must first be loaded from disk storage. This gives rise to the fundamental problem of **memory management**, where "memory" refers to main memory (or MM, a.k.a. RAM).

- Let us clarify the basic issues here. The CPU appears to "directly" address locations in the main memory (MM). For example, a machine instruction might say `LOAD R1, ADDR` where R1 is a register and ADDR refers to a location in MM.

  > In our Toy Hardware, we have a similar instruction, `LOI R1, R2` where the address is stored in register R2. Observe that each instruction of THE Machine involves at most one memory reference (these are LOAD, STORE and JUMP instructions). In particular, to move data between two memory locations, we need a load and a store instruction.

  But it is an illusion to think that the CPU can directly get to the main memory as easily as it can access registers. This illusion is accomplished with the help of a piece of hardware called the MMU. Note that we have no similar illusions about CPU being able to address disk space – that is why we normally do not consider it possible for programs to be stored in disks during execution.

  In short, during program execution, getting information from/to main memory to CPU is nontrivial because such operations are one or two orders of magnitude slower than a CPU cycle. To emphasize that this reference to memory is a significant event, we talk about **memory reference** events during execution. Various issues in memory management has to do with overcoming the inherent slowness of memory references.

- Simplest memory management model: (1) The entire process image must be in memory. (2) Contiguous space is allocated for a process image.

  Straighforward Implementation:

  We need to maintain two data structures called the **FREE MEMORY** and **USED MEMORY**.

  We need to implement two functions: **free memory** and **allocate memory**.

- Each machine instruction for a program is written for a **logical address space**. E.g., the ADDR in `LOAD R1, ADDR` is an address in this logical address space. After the program is loaded as a process, each logical address will refer to an address in MM, i.e., an address in **physical address space**.

  The transformation from logical address to physical address is rather simple, and is process-dependent. We associate with each process two special registers, a **base register** and a **limit register**. The physical address is simply logical address plus the value of the base register. The limit register stores the address of the largest legal physical address, and is used for checking if an address is in range. This is the simplest form of **relocatable addressing**.

- The preceding memory allocation scheme creates **holes**, i.e., the unused spaces between two consecutive allocated blocks of memory. Note that initially, we might be able to avoid such holes, by allocating space consecutively. But because of the dynamic nature of processes which come and go, eventually holes will develop. Holes that are too small for allocation is wasted. This is the **fragmentation problem**. (We shall shortly refine this terminology to call this **external fragmentation**.)

- Free Memory:

  Typically a linked list of all the contiguous blocks of available memory. These are in sorted order to allow merging. Similarly for Used Memory.

- Free Memory Function:

  Add the returned block to the free list Merge adjacent free blocks if possible.

- Allocate Memory Function:

  Find a free block that is large enough, and allocate part of the block. Retain the rest in free memory.

- STRATEGIES TO ALLOCATE SPACE:

  1. First Fit (OK).
     ACTUALLY, there is an implicit notion that "first" means as it occurs in the free memory list. But one can generalize this to mean free in some other criteria. In this case, this can even give us "best fit".
  2. Best Fit or "Smallest fit" (OK).
     The smallest one that fits. To implement this, we need another data structure to to list the free blocks in size order.
  3. Worst Fit (NO)
     Idea: avoid getting small useless blocks. PROBLEM: you quickly deplete your biggest blocks, and this may prevent future allocation for a large request.

4. Quick Fit
   Maintain a separate pool of typical block sizes...

§2. Memory Cache

- We begin with a simple observation: since instructions are stored in the process memory, each instruction must be fetched from memory before it can be executed. Hence, *each instruction requires at least one memory access, regardless of whether the instruction directly address memory.*

  An example of an instruction from THE Machine that does not explicitly address memory is `ADD R1, R2` where R1, R2 are registers.

- The above observation raises a concern when we recall that CPU speeds are one or two orders of magnitude faster than memory access speed. Intuitively, one order of magnitude speed gap means that the CPU is idle about 90% of the time waiting for data from memory! Two orders of magnitude translates into the CPU idling about 99.9% of the time. This situation is clearly not acceptable.

  This is sometimes known as the **CPU-memory bottleneck** (there is a corresponding bottleneck between memory and disk). This bottleneck is apparently getting worse since the 1980s, because CPU speeds have been increasing at a much faster rate than memory speeds.

- The standard solution for this CPU-memory bottleneck is to add **cache memory** whose speeds can better match the CPU speed. The preferred terminology here is **latency** rather than speed of data access, whether applied to memory, cache or CPU registers. Such cache memory is expensive relative to memory, and so we have a limited amount of them. Alternatively, one uses different levels of caches.

- The cache memory is basically a duplicate of portions of the memory. When a memory address is to be read, we first look for this in the cache. If the data is in cache, we have a **cache hit** and otherwise a **cache miss**. In cache hit, we can directly use the data without going to the memory. Otherwise, we first need to fetch the data from memory.

- Typical speeds. Formula for effective speeds:

§3. Relocatable Addressing

- Compiled programs work with **logical addresses**.

  In execution, the CPU periodically generates a LOGICAL ADDRESS (0 to some limit).

- This must be transformed into a PHYSICAL ADDRESS.

- Hardware and RUNTIME support for this: MMU has a RELOCATION REGISTER holding a value.
- PHYSICAL ADDRESS = LOGICAL ADDRESS + RELOCATION REGISTER
- REMARK: sometimes "LOGICAL" is called "VIRTUAL". E.g., "virtual address" and "virtual memory". But it is best to reserve virtual memory for another concept.
- Definition: RELOCATION is the translation of logical address to physical address.
- Applications: simple memory protection, and allows compaction of processes in main memory.

§4. Paging

- Paging is a memory allocation solution that addresses the problem of (external) fragmentation and removes the need for contiguous memory allocation.
- All modern memory allocation use some form of paging.
- Paging replaces external fragmentation with its own **internal fragmentation** which is less severe than external fragmentation.
- Needed: concept of (LOGICAL) PAGES and FRAMES. Pages and frames are two sides of the coin: one in logical space, other is physical space.
- Need a PAGE TABLE to map from pages to frames.

  The simplest form of the Page Table is as follows: it is an array `PT` with as many entries as there are logical pages. Suppose there are N many logical pages. Then `PT[i]` stores the number of the frame that contains page i, where $i = 0, 1, \ldots, N - 1$.
- Invariably, physical space is (much) smaller than logical space. Since not all logical pages can fit into physical space, we must allow the possibility that `PT[i]` does not refer to a valid frame number (e.g., it might store a negative number to indicate this).
- E.g., Suppose we have 32-bit address space, and each address refers to a byte. This is 4 GB. Suppose each page is 4 KB. This means we have 1 Million pages. If each page table is 4 bytes, we need 4 MB of memory for the page table.
- Moreover, in the simplest paging model, we assume the Page Table is stored in MM. In the preceding example, it means we need 4MB of MM for each process. This is a big memory requirement, giving rise to issues of large page table sizes.

Improvements to Paging (1): Caching.

- Since page tables reside in main memory, it means each memory reference requires at least 2 references to main memory. Once to look up the page table entry, and second to access the actual frame containing the data in the logical address.

  Then each process only need to keep ONE register ("Page Table Base Register" (PTBR)) to point to the location in main memory.

- Solution to the 2 memory references issue is a form of caching:

  We use a specialized hardware cache called **translation look-aside buffer** (TLB). This is an associative memory: each entry is a pair (key,value). Typically, TLB has 64 to 1024 entries.

  HOW to use TLB? It contains only some entries from the page table which is store in Main Memory (as in Solution 1).

  In case of a TLB MISS, we act as in Solution 1. But we also add this new entry to TLB. Thus, we need some cache replacement policy.

  NOTE: "Wired down entries" means they will never be replaced. Useful for critical pages.

  It is important to uderstand that TLB is complementary to memory caches discussed above. The TLB is just caching the information of the page table, not the contents of memory. In the absence of a memory cache, at best the TLB can only reduce the "effective time" for a single memory access to $MAT + TLBT$, where MAT is the time to access memory, and $TLBT$ is the time to access the TLB.

- REFINEMENT of TLB idea:

  We normally need to refresh TLB with each context switch.

  One way to avoid this is to store with each TLB Entry a identifier (ASID's) for the process. This way, we do not need to flush the TLB with each context switch!

Improvements to Paging (2): Reducing Table Size

- Since most pages are not used, it is wasteful to allocate memory to all the potential pages. A simple way (see Silberschatz, p.296) is to store the highest numbered page that is used in a special register **page table length register** (PTLR). Logical address references are checked against this register, and an error trap is generated when out of range. This register value could be dynamically modified if desired.

Improvements to Paging (3): Reducing Internal Fragmentation

- Roughly speaking, the space wasted to internal fragmentation is half-a-page per process. To avoid waste, we can reduce the page size.

- However, there is a tradeoff: Smaller page size increases the size of the page table.

  It also increases the **per-byte** memory I/O time since each memory I/O has a fixed non-negligible cost.

  NOTE: smaller page size appears to improve the hit-ratio.

- To balance these tradeoffs, we could introduce 2 or more page sizes. For illustration, suppose we allow pages of sizes 16KB and 1KB. Call the smaller pages **fragment pages**. Then up to the last 15 pages of a process can be fragment pages.

- However, for practical reasons, we can use fewer than 15. For instance, in the FreeBSD system, each process has at most one fragment page. Fragment pages are effective if many small processes that can fit into a single fragment page.

§5. Protection

- Each page has associated protection bits, stored in page table.

  Previously, we assume that `PT[i]` contains just a frame number. We now see that it may contain additional protection bits.

- These bits typically gives permissions to Read/Write/Execute. Thus it is similar to the sets of 3 bits associated with unix file permissions.

- If an instruction violates these permissions, a hardware error trap is sprung.

- To support multiprocessing, we also have valid-invalid bits. Invalid means page does not belong to the current logical space (but belongs to another processor's).

- Since most processors use only a fraction of its page table, hardware support this by providing a Page-Table Length Register (PTLR) to indicate size of page table.

- Another related issue is **shared pages**. A code is **re-entrant** (or pure) if it cannot be modified at execution time. Such code can be shared.

§6. Page Table Structure

- See sect.8.5 of text. We had started out above with the simplest model, the page table as an array.

- Because of large table sizes, we may need to go to hierarchical page tables.

§7. Process Swapping

- MEMORY MODEL: Main Memory (or RAM) and Secondary Memory (or Disk).
- Processes must be in Main Memory to run.
- Backing Store: reserved space on Disk for processes not currently active.
- SWAPPING: the movement of processes between Main Memory and Backing Store.
- Typical time to swap between Main Memory and Backing Store: 420 ms.

  E.g., 1 MB image at 5MB/sec transfer rate takes 200 ms. Add some overhead.

  This is expensive, so must use it carefully!
- On UNIX: swapping is only enabled when there are many processes are running, and the Main Memory usage has exceeded some threshhold.

§8. Segmentation

- Paging solves one problem, that of allocating memory.

  That is, we need to map physical (i.e., virtual) memory to physical memory.
- Segmentation solves another problem, that of providing structure to the allocated memory.

  In other words, the memory for a given process is not a homogeneous linear array of bytes, but can be viewed a variable size units called SEGMENTS.
- Now, we want to map "segmented memory" to logical memory (or to physical memory).

  Each "segmented memory" address is a pair (Segment-number, offset).

  This mapping is contained in the SEGMENT TABLE. In this table, we store a SEGMENT BASE and SEGMENT LIMIT for each Segment.
- E.g., a typical process will have these segments:

  symbol table, stack, data, routines, etc.

  When libraries are linked, we might get many separate segments.
- One benefit of segmentation is that we can achieve protection of segments.

  Protection bits can be set per segment. E.g., read-only, execute-only, etc.

  This makes sense since segments are often semantically meaningful units.
- Another benefit: code sharing.

§9. Segmentation with Paging

- So far, we have tried to describe segmentation and paging as distinct mechanisms. In practice, we ought to combine the two. Let us look the implementation of this in Intel 80x86 (or x86) family of processors.
- The Intel 80x86 processors is perhaps the most famous family of CPU's in the world. It is so-called because the processors in this family had numbers ending in 86 (as in 8086, 80186, 80286, etc). It started in 1981 and each successive generation (286, 386, 486, 586, etc) is backwards compatible. The 586 family is also known as Pentium line. See Wikipedia under x86 Architecture or Pentium.
- Addresses are segmented in the 8086 family: 16-bit segment number, and a 16-bit offset.
- Please refer to Figure 9.21, page 355, in text. Our goal is to understand this figure.
- Each process has up to 16K segment, and each segment size is at most 4GB. Each page size is 4KB.
- The logical address space of processes is divided into 8K private segments and 8K shared segments.

  To manage these segments, we have a **local descriptor table** (LDT) for the private segments, and a **global descriptor table** (GDT) for the shared segments.
- Each LDT/GDT entry consists of an 8-byte segment descriptor (base location, limit, etc), refering to main memory.

  The logical address is a pair (SELECTOR,OFFSET). The SELECTOR is 16 bits (13-bit segment, 1-bit for GDT/LDT flag, 2-bit protection). The OFFSET is 32 bits.
- The CPU has 6 **segment registers** (so a program can access up to 6 segments at once).

  It also has 6 **microprogram registers** to hold descriptors from LDT or GDT.
- Refer to Figure 9.21, page 355, in text for the following. The main algorithm to form a 32-bit physical address:

  1. Start with the logical address =(SELECTOR,OFFSET).
  2. Use SELECTOR in segment register to load entry from LDT/GDT
  3. Use base in microprogram register to form a **linear address**. Use limit in microprogram register to check validity of address.
  4. If valid, offset (from logical address) is added to to linear address.
  5. Translate resulting address to physical address.

Exercises

Exercise 9.1: Please explain the following analogy:

`page : logical space :: frame : physical space.`

◊

Exercise 9.2: Explain the remark that paging enables a separation between the program's view of memory from the physical memory. ◊

Exercise 9.3: Assume a word-addressable memory system (a word is 4 bytes). Suppose page size is 16 KB, and physical memory is 2GB.
(i) If our logical address space is 32-bits, How many entries would the page table have?
(ii) What is the minimum soze of the page table?
(iii) What if the logical address space is 32-bits?
(iv) Assume that we solve the problem of large table size in (iii) using hierarchical page tables, where each node of the hierarchy is a page. What is the minimum depth of the trey ◊

Exercise 9.4: What is a solution to fragmentation? Describe what is needed to solve this problem. What new issues arise?

◊

Exercise 9.5: What is "paging" a solution to?

◊

Exercise 9.6: What are some solutions to huge page tables?

◊

End Exercises