MEMORY MANAGEMENT

Just as processes share the CPU, they also share memory. This section is about mechanisms for doing that sharing.

EXAMPLES OF MEMORY USAGE:

Calculation of an **effective address**

Fetch from instruction
Use index offset

Example: (Here index is a pointer to an address)

```
loop:
load        register, index
add         42, register
store               register, index
inc         index
skip_equal  index, final_address
branch      loop
.           ... continue ....
```

- **Relocatable** Means that the program image can reside anywhere in physical memory.

- **Binding** Mapping logical to physical addresses.
  <<< FIGURE 8.1 >>>

  This binding can be done at compile/link time. Converts symbolic to relocatable. Data used within compiled source is offset within object module.

  Can be done at load time. Binds relocatable to physical.

  Can be done at run time. Implies that the code can be moved around during execution.

- The next example shows how a compiler and linker actually determine the locations of these effective addresses.

```
 4 void       main()
 5     {
 6     printf( "Hello, from main\n" );
 7     b();
 8 }
 9
10
11 void b()
12     {
13     printf( "Hello, from 'b'\n" );
14 }
```

ASSEMBLY LANGUAGE LISTING

```
000000B0: 6BC23FD9  stw       %r2,-20(%sp       ; main()
000000B4  37DE0080  ldo       64(%sp),%sp
000000B8  E8200000  bl        0x000000C0,%r1    ; get current addr=BC
000000BC  D4201C1E   depi     0,31,2,%r1
000000C0  34213E81  ldo       -192(%r1),%r1     ; get code start area
000000C4  E8400028  bl        0x000000E0,%r2    ; call printf
000000C8  B43A0040   addi     32,%r1,%r26       ; calc. String loc.
000000CC  E8400040  bl        0x000000F4,%r2    ; call b
000000D0  6BC23FD9   stw      %r2,-20(%sp)      ; store return addr
000000D4  4BC23F59  ldw       -84(%sp),%r2
000000D8  E840C000  bv        %r0(%r2)          ; return from main
000000DC  37DE3F81   ldo      -64(%sp),%sp

                                                STUB(S) FROM LINE 6
000000E0: E8200000  bl        0x000000E8,%r1
000000E4  28200000   addil    L%0,%r1
000000E8: E020E002  be,n      0x00000000(%sr7,%r1)

000000EC  08000240  nop                         void    b()
000000F0: 6BC23FD9  stw       %r2,-20(%sp)
000000F4: 37DE0080  ldo       64(%sp),%sp
000000F8  E8200000  bl        0x00000100,%r1     ; get current addr=F8
000000FC  D4201C1E   depi     0,31,2,%r1
00000100  34213E01  ldo       -256(%r1),%r1     ; get code start area
00000104  E85F1FAD  bl        0x000000E0,%r2    ; call printf
00000108  B43A0010   addi     8,%r1,%r26
0000010C  4BC23F59  ldw       -84(%sp),%r2
00000110  E840C000  bv        %r0(%r2)          ; return from b
00000114  37DE3F81   ldo      -64(%sp),%sp
```

```
            EXECUTABLE IS DISASSEMBLED HERE
00002000  0009000F                                    ; . . . .
00002004  08000240                                    ; . . . @
00002008  48656C6C                                    ; H e l l
0000200C  6F2C2066                                    ; o ,   f
00002010  726F6D20                                    ; r o m
00002014  620A0001                                    ; b . . .
00002018  48656C6C                                    ; H e l l
0000201C  6F2C2066                                    ; o ,   f
00002020  726F6D20                                    ; r o m
00002024  6D61696E                                    ; m a i n

000020B0  6BC23FD9  stw           %r2,-20(%sp)        ; main
000020B4  37DE0080  ldo           64(%sp),%sp
000020B8  E8200000  bl            0x000020C0,%r1
000020BC  D4201C1E  depi          0,31,2,%r1
000020C0  34213E81  ldo           -192(%r1),%r1
000020C4  E84017AC  bl            0x00003CA0,%r2
000020C8  B43A0040  addi          32,%r1,%r26
000020CC  E8400040  bl            0x000020F4,%r2
000020D0  6BC23FD9  stw           %r2,-20(%sp)
000020D4  4BC23F59  ldw           -84(%sp),%r2
000020D8  E840C000  bv            %r0(%r2)
000020DC  37DE3F81  ldo           -64(%sp),%sp
000020E0  E8200000  bl            0x000020E8,%r1       ; stub
000020E4  28203000  addil         L%6144,%r1
000020E8  E020E772  be,n          0x000003B8(%sr7,%r1)
000020EC  08000240  nop
000020F0  6BC23FD9  stw           %r2,-20(%sp)         ; b
000020F4  37DE0080  ldo           64(%sp),%sp
000020F8  E8200000  bl            0x00002100,%r1
000020FC  D4201C1E  depi          0,31,2,%r1
00002100  34213E01  ldo           -256(%r1),%r1
00002104  E840172C  bl            0x00003CA0,%r2
00002108  B43A0010  addi          8,%r1,%r26
0000210C  4BC23F59  ldw           -84(%sp),%r2
00002110  E840C000  bv            %r0(%r2)
00002114  37DE3F81  ldo           -64(%sp),%sp

00003CA0  6BC23FD9  stw           %r2,-20(%sp)         ; printf
00003CA4  37DE0080  ldo           64(%sp),%sp
00003CA8  6BDA3F39  stw           %r26,-100(%sp)
00003CAC  2B7CFFFF  addil         L%-26624,%dp
00003CB0  6BD93F31  stw           %r25,-104(%sp)
00003CB4  343301A8  ldo           212(%r1),%r19
00003CB8  6BD83F29  stw           %r24,-108(%sp)
00003CBC  37D93F39  ldo           -100(%sp),%r25
00003CC0  6BD73F21  stw           %r23,-112(%sp)
00003CC4  4A730009  ldw           -8188(%r19),%r19
00003CC8  B67700D0  addi          104,%r19,%r23
00003CCC  E8400878  bl            0x00004110,%r2
00003CD0  08000258  copy          %r0,%r24
00003CD4  4BC23F59  ldw           -84(%sp),%r2
00003CD8  E840C000  bv            %r0(%r2)
00003CDC  37DE3F81  ldo           -64(%sp),%sp
00003CE0  E8200000  bl            0x00003CE8,%r1
00003CE8  E020E852  be,n          0x00000428(%sr7,%r1)
```

- **Dynamic loading** Routine isn't called into memory until needed. May or may not require binding at run time.

- **Dynamic Linking** Code is mapped or linked at execution time. Example is system libraries.

- **Memory Management** Performs the above operations. Usually requires hardware support.

LOGICAL VERSUS PHYSICAL ADDRESS SPACE:

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

Logical address generated by the CPU; also referred to as a virtual address.

Physical address address seen by the memory unit (hardware).

- Logical and physical addresses are the same in compile-time and load-time address binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding schemes.

- **Memory-management Unit** (**MMU**) is a hardware device that maps virtual to physical addresses.

- The user program deals with logical addresses; it never sees the real physical addresses.

SWAPPING

- Several processes share the same physical memory and are swapped to/from disk in turn. What are pros and cons of this?
- Medium term scheduler tries to make sure ALL processes get share of the action.
- If a higher priority job wants action, then can swap IN that process by swapping OUT some other process.
- Swapping requires a backing store.
- How much time is required for swapping? (DO calculation)

SINGLE PARTITION ALLOCATION

BARE MACHINE:

- No protection, no utilities, no overhead.
- This is the simplest form of memory management.
- Used by hardware diagnostics, by system boot code, real time/dedicated systems.
- logical == physical
- User can have complete control. Commensurably, the operating system has none.

DEFINITION OF PARTITIONS:

- Division of physical memory into fixed sized regions. (Allows addresses spaces to be distinct = one user can't muck with another user, or the system.)

- The number of partitions determines the level of multiprogramming. Partition is given to a process when it's scheduled.

- Protection around each partition determined by

    bounds (upper, lower)
    base / limit.

  - These limits are done in hardware.

RESIDENT MONITOR:

- Primitive Operating System.

- Usually in low memory where interrupt vectors are placed.

- Must check each memory reference against fence (fixed or variable) in hardware or register. If user generated address < fence, then illegal.

- User program starts at fence -> fixed for duration of execution. Then user code has fence address built in. But only works for static-sized monitor.

- If monitor can change in size, start user at high end and move back, OR use fence as base register that requires address binding at execution time. Add base register to every generated user address.

- Isolate user from physical address space using logical address space.

- Concept of "mapping addresses". **<<< FIGURE 8.6 >>>**

MULTIPLE-PARTITION ALLOCATION

JOB SCHEDULING

- Must take into account who wants to run, the memory needs, and partition availability. (This is a combination of short/medium term scheduling.)

- Sequence of events:

    1. In an empty memory slot, load a program
    2. THEN it can compete for CPU time.
    3. Upon job completion, the partition becomes available.

- Can determine memory size required (either user specified or "automatically").

DYNAMIC STORAGE

- (Variable sized holes in memory allocated on need.)

- Operating System keeps table of this memory - space allocated based on table.

- Adjacent freed space merged to get largest holes - buddy system.
  <<< FIGURE 8.8>>>

    First fit - allocate the first hole that's big enough.
    Best fit - allocate smallest hole that's big enough.
    Worst fit - allocate largest hole.

    (First fit is fastest, worst fit has lowest memory utilization.)

- Avoid small holes (**external fragmentation**). This occurs when there are many small pieces of free memory.

- What should be the minimum size allocated, allocated in what chunk size?

- Want to also avoid **internal fragmentation.** This is when memory is handed out in some fixed way (power of 2 for instance) and requesting program doesn't use it all.

LONG TERM SCHEDULING

- If a job doesn't fit in memory, the scheduler can

  wait for memory
  skip to next job and see if it fits.

- What are the pros and cons of each of these?

- There's little or no internal fragmentation, but a great deal of external fragmentation. Look again at **<<< FIGURE 8.8 >>>**

COMPACTION

- Trying to move free memory to one large block. **<<< FIGURE 8.10 >>>**

- Only possible if programs linked with dynamic relocation (base and limit.)

- There are many ways to move programs in memory. **<<< FIGURE 8.11 >>>**

- Swapping: if using static relocation, code/data must return to same place. But if dynamic, can reenter at more advantageous memory.

PAGING:

Permits a program's memory to be physically noncontiguous so it can be allocated from wherever available. This avoids fragmentation and compaction.

HARDWARE

- An address is determined by:

  page number (index into table) + offset
  ---> mapping into --->
  base address (from table) + offset.

  <<< FIGURE 8.12 >>>
  Frames = physical blocks
  Pages = logical blocks

- Size of frames/pages is defined by hardware (power of 2 to ease calculations)

EXAMPLE: **<<< FIGURE 8.14 >>>**

IMPLEMENTATION OF THE PAGE TABLE

- A 32 bit machine can address 4 gigabytes which is 4 million pages (at 1024 bytes/page). WHO says how big a page is, anyway?

- Could use dedicated registers (OK only with small tables.)

- Could use a register pointing to table in memory (slow access.)

- Cache or associative memory (TLB = Translation Lookaside Buffer): simultaneous search is fast and uses only a few registers**.<<<FIGURE 8.16>>>**

  Issues include:

  key and value
  hit rate 90 - 98% with 100 registers
  add entry if not found

  Effective access time = %fast * time_fast + %slow * time_slow

  Relevant times:
  20 nanoseconds to search associative memory – the TLB.
  200 nanoseconds to access memory and bring it into TLB for next time.

  Calculate time of access:
  hit = 1 search + 1 memory reference
  miss = 1 search + 1 mem reference(of page table) + 1 mem reference.

SHARED PAGES

- Data occupying one physical page, but pointed to by multiple logical pages.
- Useful for common code - must be write protected. (NO writeable data mixed with code.) **<<< FIGURE 8.21 >>>**
- Shared pages are extremely useful for read/write communication between processes.

INVERTED PAGE TABLE:

- One entry for each real page of memory; entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Essential when you need to do work on the page and must find out what process owns it.
- Use hash table to limit the search to one - or at most a few - page table entries.

PROTECTION

- Bits associated with page tables.
- Can have read, write, execute, valid bits
- Valid bit says page isn't in address space.
- Write to write-protected page causes a fault. Touching an invalid page causes a fault.

ADDRESS MAPPING

- Allows physical memory larger than logical memory.
- Useful on 32 bit machines with more than 32-bit addressable words of memory.
- The operating system keeps a frame containing descriptions of physical pages: if allocated, then to which logical page in which process.

MULTILEVEL PAGE TABLE

- A means of using page tables for large address spaces. Best seen with an example **<<< FIGURE 8.18 >>>**

SEGMENTATION:

USER'S VIEW OF MEMORY

- A programmer views a process consisting of unordered segments with various purposes. This view is more useful than thinking of a linear array of words. We really don't care at what address a segment is located.
- Typical segments include

  global variables
  procedure call stack
  code for each function
  local variables for each
  large data structures

- Logical address = segment name (number) + offset
- Memory is addressed by both segment and offset.

HARDWARE

- Must map a dyad into one-dimensional address. **<<< FIGURES 8.23, 8.24 >>>**
- base / limit pairs in a segment table.

PROTECTION AND SHARING

- Addresses are associated with a logical unit (like data, code, etc.) so protection is easy.
- Can do bounds checking on arrays
- Sharing specified at a logical level, a segment has an attribute called "shareable".
- Can share some code but not all - for instance a common library of subroutines.

<<< FIGURE 8.25 >>>

FRAGMENTATION

- Use variable allocation since segment lengths vary.
- Again have issue of fragmentation; Smaller segments means less fragmentation. Can use compaction since segments are relocatable.

PAGED SEGMENTATION

- Combination of paging and segmentation.

address	=	frame at (page table base for segment
	+	offset into page table)
	+	offset into memory

- Look at example of Intel architecture. **<<< FIGURE 8.28 >>>**