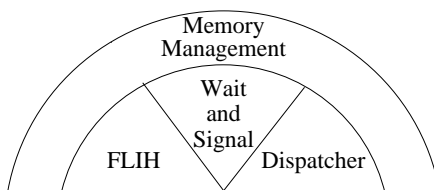


- ref: [Tanenbaum, sect 6.1, 7.4], [O'H&Bryant, ch 7], [Null&Lobur, sect 6.5, 8.2, 8.3]
- memory management objectives
- OS layers: where memory management fits in
- address maps
- paging (recall M2 lecture)
- segmentation
- compilation and linking; ELF object files (recall C4 lecture; also M5)
- structure and loading of ELF executable files

Memory Management Objectives

1. relocation
 - load to different memory location; dynamically move processes in memory
2. protection
 - write/read(?) memory of other processes
3. sharing (code, data)
 - sometimes desirable
4. logical organisation
 - same address space (i.e. range from 0...) for each process / source file(?)
5. physical organisation: multilevel storage (memory hierarchy)



MM and OS Layers:

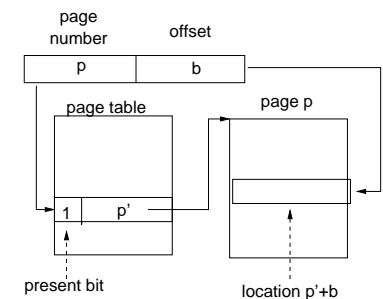
- a conceptually simple and elegant device to achieve all previous objectives
- distinguishes (virtual) program addresses from those of (physical) memory locations into which they are (currently) mapped
 - $f: VA \rightarrow PA$
- the programmer sees, and programs for, a virtual memory whose characteristics differ from those of the real memory
- a preliminary example: mapping by a simple offset
 - Base Register (B): constant offset to add to memory references
 - ◆ $f(a) = B + a$
 - also need a Limit Register (L): specify maximum valid offset
 - ◆ if $a < 0$ or $f(a) > L$, memory violation
 - mapping is simple (linear), but Objectives 3 (sharing) and 5 (virtual address space constrained by physical memory) not met

Paging

- provides a virtual memory space that is greater than physical memory
 - a program address is divided into two parts $a = \begin{matrix} p \\ b \end{matrix}$
 - high order address bits $p = a/P$ gives the page number
 - low order address bits $b = a \% P$ gives the page offset
- where P is the page size (e.g. on PeANUt, $P = 32$ cells) $f(a) = p' + b$

issues:

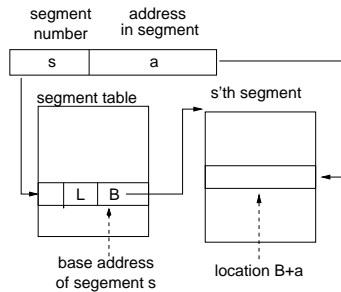
- page size
 - tradeoff reducing overhead with loading unused memory
- page replacement policy (e.g. LRU analysis)
- avoiding thrashing
- protection: R/W/X permissions in page table (&MMU)



Segmentation

- idea: distinguish memory areas by their intended use (or origin)
 - e.g. code: execute only (?); some (static) data is read-only; also heap and stack
- program addresses of the form $a' = \boxed{s \ a}$, where:
 - the segment number s is used to index the segment table
 - each segment has a limit $L(s)$:
 - if $a < 0$ or $f(a) > L(s)$, memory (segmentation!) violation

$$f'(a') = B + a$$



- may be combined with paging: $f(f'(a')) = f(B+a) = p+b$

Linking and Loading: Motivation

- why study this?
 - helps you build large program systems
 - understand linker errors (multiply defined global variables)
 - avoid dangerous program errors
 - know difference between global/local, and static variables
 - scoping rules
 - illustrates systems concepts: virtual memory, paging, memory mapping
 - know about shared libraries and dynamic linking
- example: what goes on here?

```

/* mswap.c */
void swap();
int buf[2]={1,2}; /* swap.c */

int main(){
    swap();
    return 0;
}

void swap() {
    int temp;
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
    
```

Paging and Segmentation

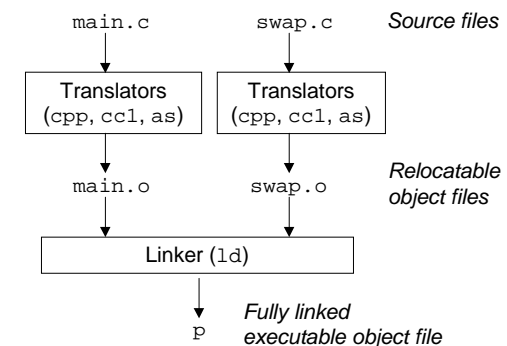
- comparison:

consideration	paging	segmentation
programmer aware	no	yes (from A.L.)
number of linear address spaces	1	many
virtual address space can exceed physical	yes	yes
variable sized 'units' easily handled	no	yes
why invented	to simulate large memories	to enhance protection, 'encapsulate' memory

- on real machines:
 - UltraSparc IIIc: demand paging, page size of 8, 64, 512 and 4096 KB
 - Pentium IV: demand paging, segmentation with paging, (normal) page size 4KB

Static Linking

- compile with `gcc -Wall -O2 -o p mswap.c swap.c` (main.c = mswap.c)



- symbol resolution: associate each symbol reference with exactly one (relocatable) address
- relocation: add offsets to compiler generated addresses that start from 0
 - this must be done at every memory location with an address reference!

