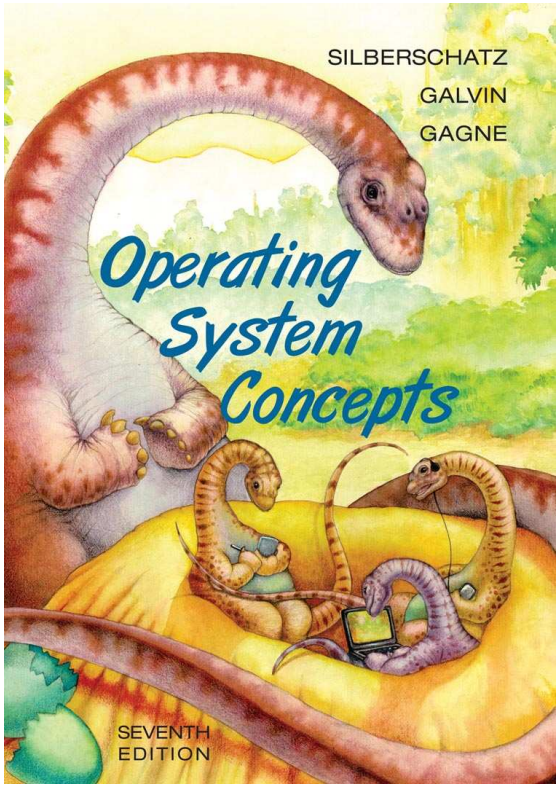
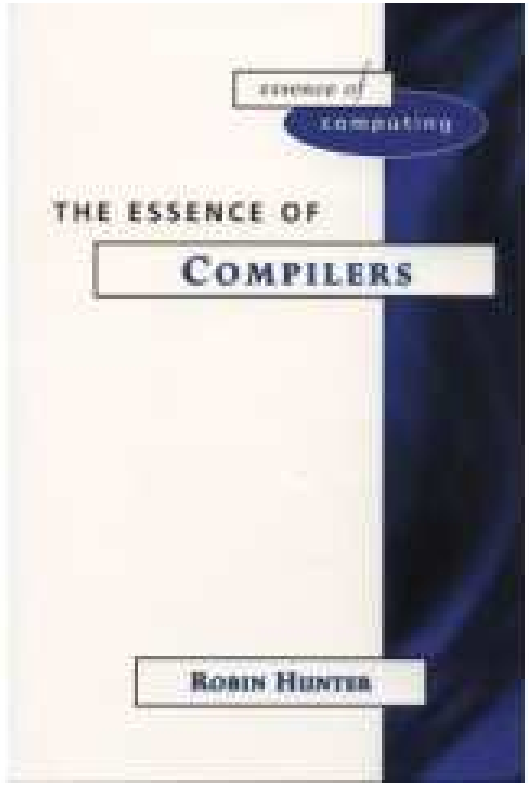
Memory Management

Frédéric Haziza <daz@it.uu.se>

Department of Computer Systems
Uppsala University

Spring 2007





Operating Systems

- Process Management
- Memory Management
- Storage Management

Compilers

- Compiling process & Lexical analysis
- Parsing
- Semantic & Code generation

Recall – Memory Management

- Which processes and data to move in and out memory
- Allocating and deallocating memory space as needed
- Keeping track of which parts of memory are currently being used and by whom

Outline

Typical:

1. Fetch instruction form MEM
2. Decode instruction
3. Fetch eventual operands from MEM
4. Execute instruction
5. Store eventual results to MEM

But for the MEM: stream of addresses.

No difference between data and instructions

Hardware: CPU - MEM - IO

Registers+MEM $\Longleftrightarrow$ CPU

No instruction with disk address
$\Rightarrow$ All must be first brought to MEM

Speed considerations

- Register: 1 cycle (1 ns)
- Mem (via Bus & transactions): 10 - 100 cycles (100 ms)

$\Rightarrow$ CPU must stall (if no instruction scheduling, cf DARK2)

$\Rightarrow$ Cache

Speed, ok, but Need of Protection

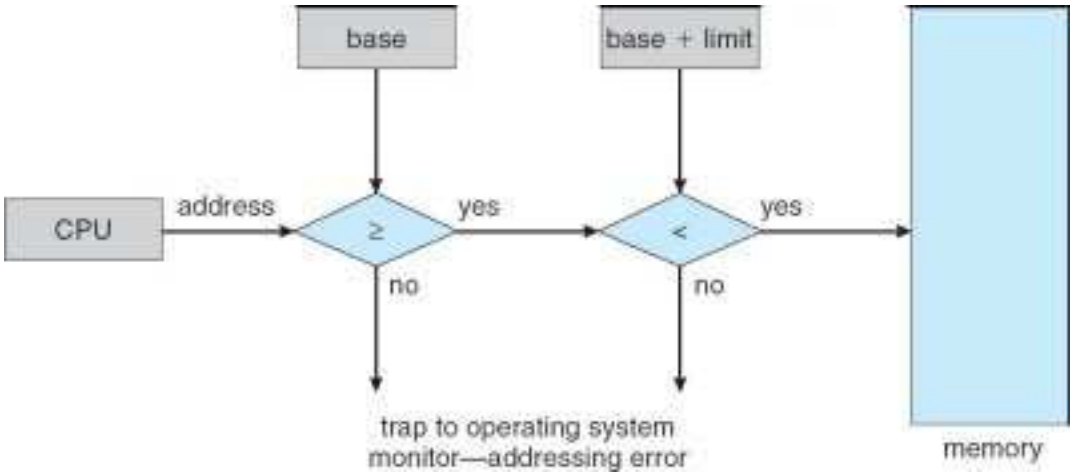
Speed, ok, but need for protection
(User process shouldn't interfer with each others)

Idea

Each process has a separate memory space,
i.e. a range of valid addresses

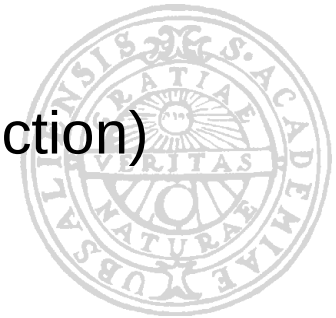
Simple example

Example: 2 registers - base & limit



HW checks *every* address. If problem, trap.

- Prevents user from touching others user space or OS space.
- Base and limit placed in PCB.
- Registers loaded by OS only (since privileged instruction)
- $\Rightarrow$ OS = *"superpower"*



How do we bring a program in MEM?

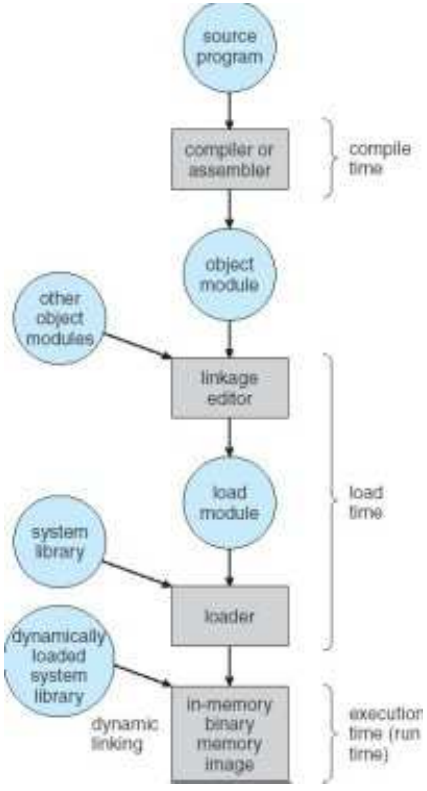
Programs are typically on disk as executable files and must be brought to MEM.

Input Queue

Processes on disk waiting to be brought into MEM

Cf. analogy with the ready queue.

Translation steps



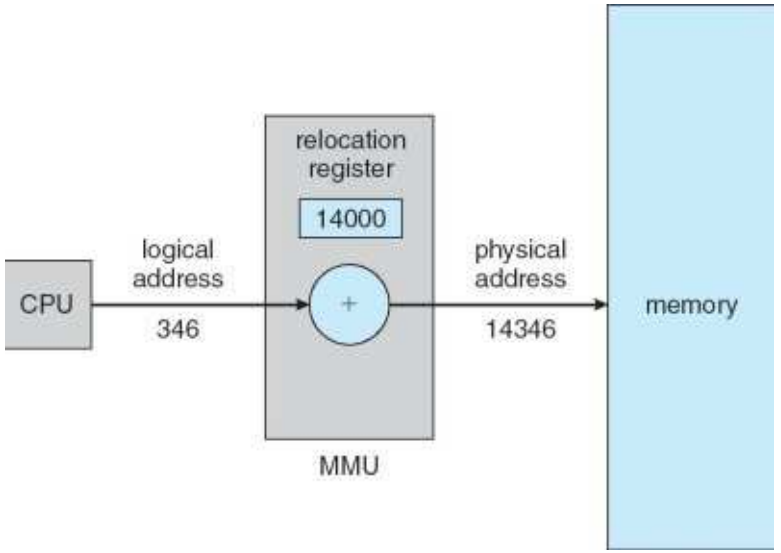
Each step: more specific translation

If translation at compile-time: absolute code

If translation as run-time: done by MMU

Example

Relocation register (base register)



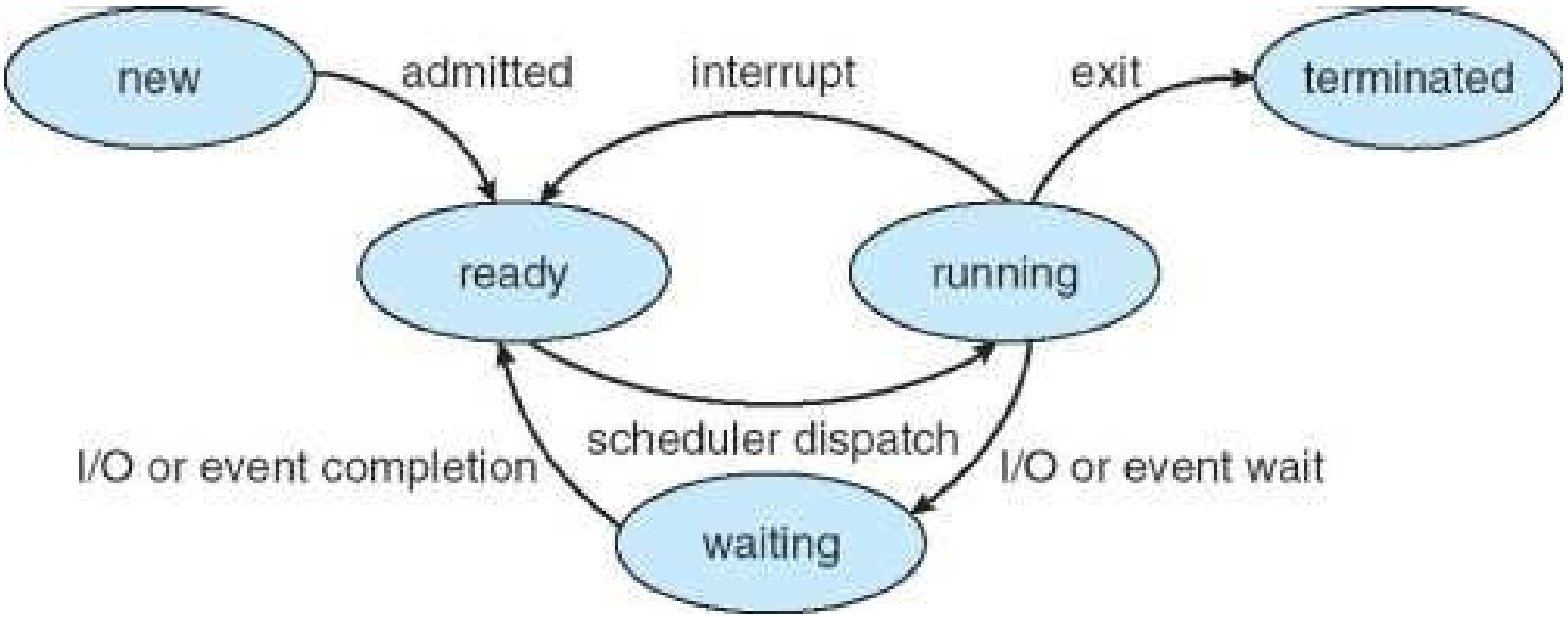
Used to relocate addresses after OS space (protection for that latter)

Address a becomes $a + x$ but OS can vary in size (due to buffers,. . .) $\Rightarrow$ x in register (dynamic relocation) added to *all* addresses.

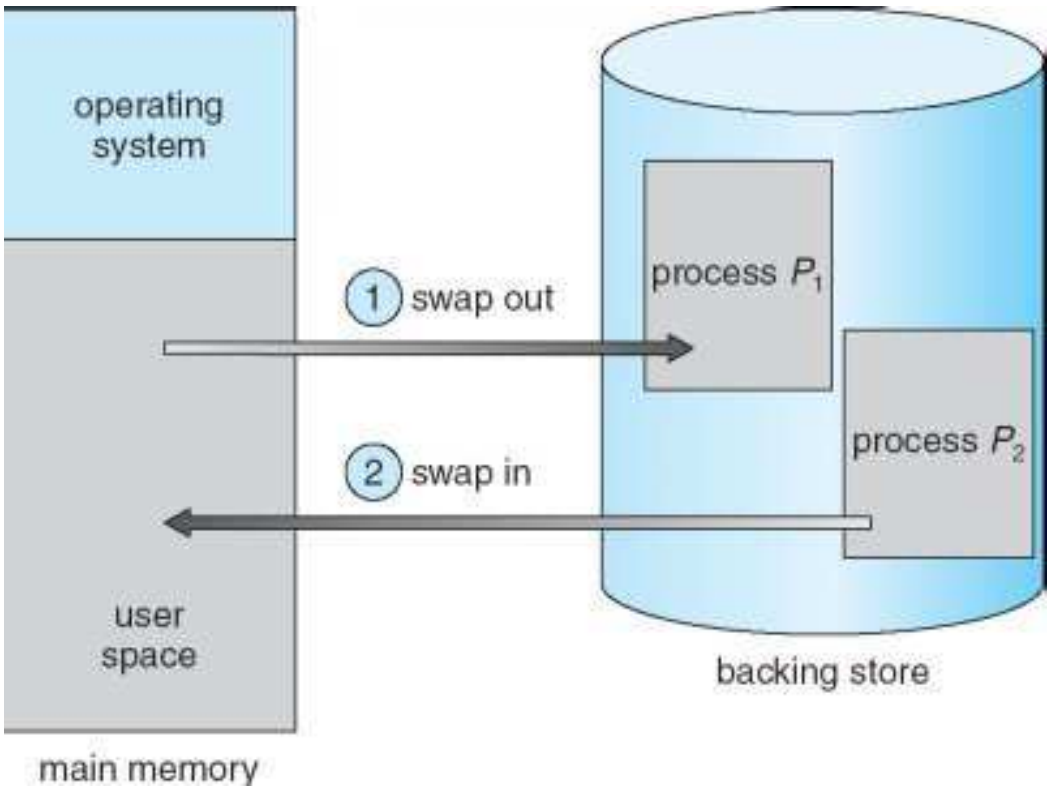
Saving Space

- Dynamic loading
  - Do we need to load everything? No, just when needed.
  - If routine is needed, check if loaded. If not, loads the routine, stored on disk as relocatable load format
  - Advantage: If never used, never loaded (example: error routines)
  - No special hardware support, up to programmers to use it smartly
- Dynamic linking
  - If static: Embrace system libs in all programs $\Rightarrow$ Programs are bigger
  - Stub (Description page 281)
  - Extension to library updates (bug fixes)

Saving More Space - Swapping



Saving More Space - Swapping



Saving More Space - Swapping example

Example

- User process: 10 MB
- Backing store: standard disk with transfer rate of 40 MB/s
- Latency: 8ms

$\Rightarrow$ Swap time = 258 ms

Swap in and out $\Rightarrow$ Total swap time $\approx$ 516 ms

If RR, quantum $\gg$ 0.516 s

Saving More Space - Swapping

Swap is slow $\Rightarrow$ Swap the necessary

i.e. what the process *IS* using and not *MIGHT BE* using.

Other constraint: Process must be idle

If swapped out, MEM reclaimed to another process, so should not be waiting for IO

Solutions:

- Never swap a process with pending IO
- Execute IO into OS buffers only

  (Transfer buffers to process mem, when swapped in)

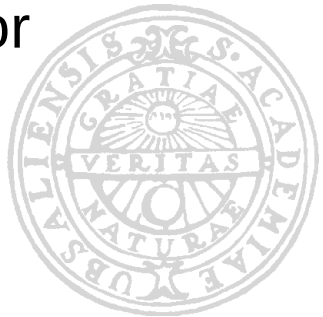
Saving Even More Space

What about the OS? Can we reclaim some of its space?

Recall the OS space is protected, for example, with the relocation register.

But can be used to *dynamically* allow OS size to change.

What to toss? OS can grow & shrink: OS buffers, code for device drivers, . . .
transcient code: comes & goes as needed



Fixed-sized partition

1 partition - 1 process

Split memory in fixed size partitions, give one to each process.

When processes terminate, memory is freed and can be reused by other processes.

Problem: Degree of multiprogramming is limited to the number of partitions

Variable-sized partition

Recall that processes waiting to be swapped in form the input queue (on disk).

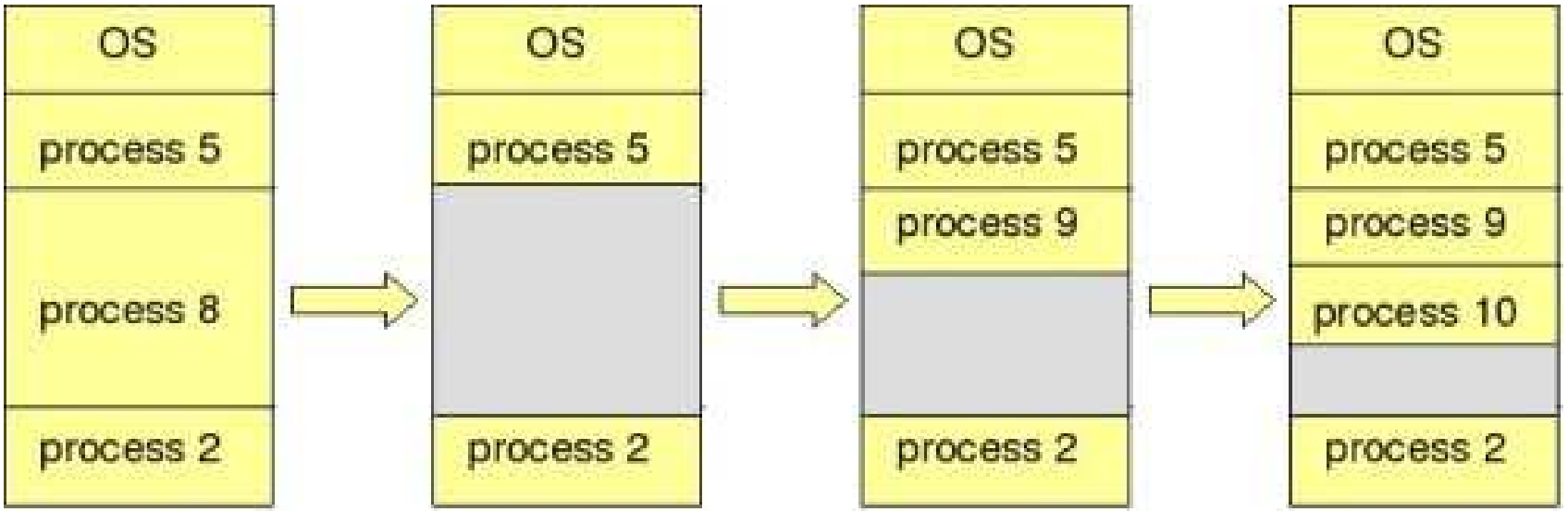
OS checks mem requirements for processes

Job Scheduling: Input queue is sorted with an algorithm (cf CPU scheduling)

Take the first one or skip down the list to find one where mem requirements can be met

Risk? Starvation

Variable-sized partition



Note: Memory is contiguous

Algorithms

First-fit

Allocate the *first* hole that is big enough

Searching stops as soon as hole found

Best-fit

Allocate the *smallest* hole that is big enough

$\Rightarrow$ smallest leftover hole

Worst-fit

Allocate the *largest* hole

Searching over the entire list (unless list is ordered)

First-fit & Best-fit $\geq$ Worst-fit. First-fit is faster than Best-fit.
But First-fit & Best-fit lead to external fragmentation

Fragmentation

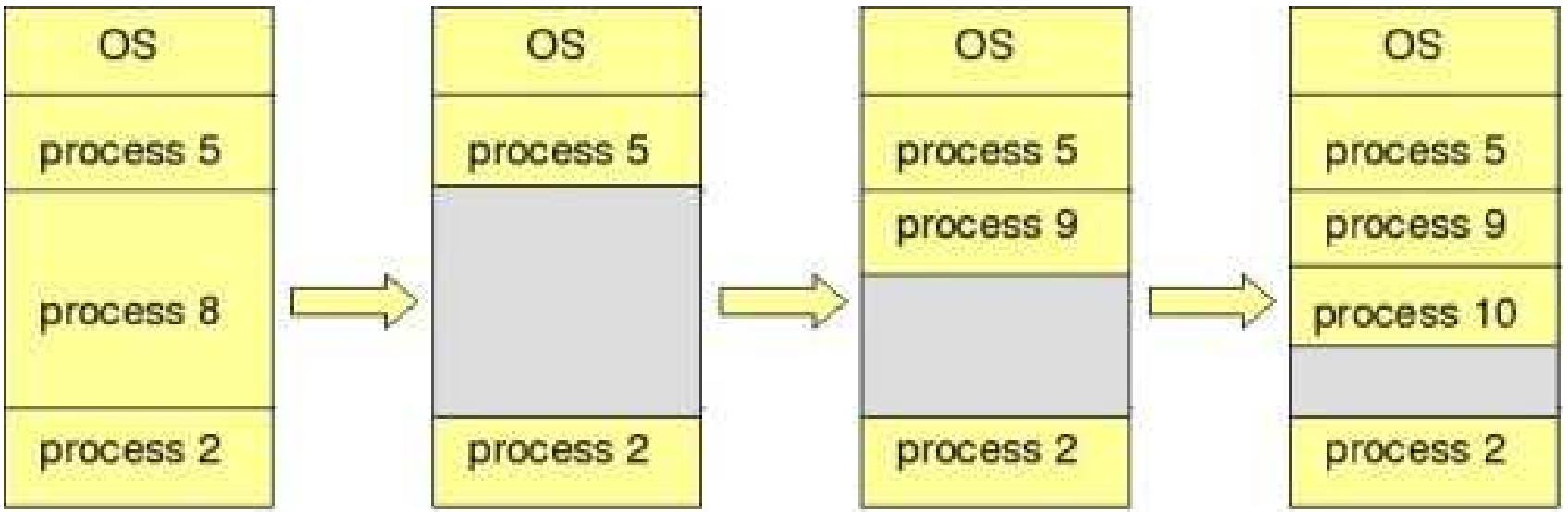
Internal

space within an allocated block of memory is unused
(the block is larger than necessary)

External

unallocated blocks of memory are unused because they are each too small, although the sum of their sizes would be usable.

Variable-sized partition



Note: Memory is contiguous

Dealing with Fragmentation

Avoid external fragmentation: **Compaction**(only if dynamic relocation)

Cost?

Permit non-contiguous address space $\Rightarrow$ Paging

Note: What about fragmentation of backing store when swapped out? Too slow to think about compaction!!