EssCS - Topic 3

Introduction to Operating Systems

Lecture 12, 19.11.2024

Nuša Zidarič

Recap and Motivation

- to improve CPU utilization: CPU is shared (reused) by different processes

  $\Rightarrow$ we need many processes in the main memory

  $\Rightarrow$ main memory is shared too

  $\Rightarrow$ memory management algorithms!

- most memory management algorithms need hardware support:

  MMU (Memory Management Unit)

Recap and Motivation

- Hardware perspective (Topic 2)
  - PC holds the address for the next insn to be fetched from I$
  - if load-store: PC generates an address for access to D$

- CPU can directly access: RegFile and main memory (regardless of mem. hierarchy)
- what happens if CPU needs something that is not contained in the main memory ?

- (history) compatibility as motivation:
  - IBM 370: a family of computers with one software suite
  - how to run same program on systems with differently sized memory ?
  - instead of rewriting the code: memory always looks like 2^n memory words (regardless of the actual memory size)
    $\Rightarrow$ virtual memory ($\mathrm{V_M}$) with virtual addresses ($\mathrm{V_A}$): length of $\mathrm{V_A}$ is n bits

NOTE: we will not differentiate *logical* and *virtual* address

Virtual Memory

- CPU uses virtual address
- CPU is always referencing the last level of the hierarchy - beyond main memory!

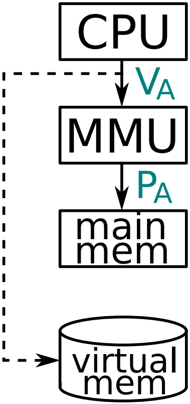
- main memory will see a sequence of addresses
- transfers between the main memory and virtual memory are invisible to the programmer - virtual

- secondary memory: virtual memory ($\mathrm{V_M}$) with virtual addresses ($\mathrm{V_A}$): n bit address
- main memory: physical memory ($\mathrm{P_M}$) with physical addresses ($\mathrm{P_A}$): m bit address
- typically: $m << n$ (physical address $<<$ virtual address)

- modern CPUs work with virtual addresses

Virtual Memory

- secondary memory: virtual memory ($\mathrm{V_M}$) with
  virtual addresses ($\mathrm{V_A}$): n bit address
- main memory: physical memory ($\mathrm{P_M}$) with
  physical addresses ($\mathrm{P_A}$): m bit address
- typically: $m << n$
  (physical address $<<$ virtual address)
  $\Rightarrow$ need for translation $\mathrm{V_A} \leftrightarrow \mathrm{P_A}$
- each byte of main memory has its $\mathrm{V_A}$ and $\mathrm{P_A}$
- MMU (Memory Management Unit):
  - address translation:
    $\mathrm{P_A} = f(\mathrm{V_A})$
    $f : \mathrm{V_A}\,\text{space} \rightarrow \mathrm{P_A}\,\text{space}$
  - we want f to be dynamic: to adapt the properties of the process to ensure a high hit rate in main memory



Virtual Memory

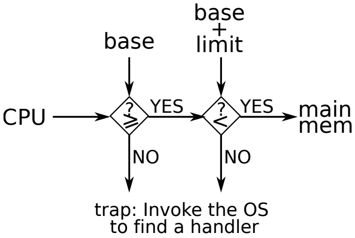
Benefits:

- more efficient use of main memory (“cache” for secondary memory)
  transfers between main mem. and virtual mem. only when needed
- each process thinks it has an uniform address space (simple memory management)
- protects the address space one process from being corrupted by other processes
  (processes can not read/write each-others memory)
- processes do not need to be aware of each-other

Challenges:

- translation must be fast! (need HW support - MMU)
- aliasing: two or more different $\mathrm{V_A}$'s can map to the same $\mathrm{P_A}$
- fragmentation

Recap: Process, address space, memory protection

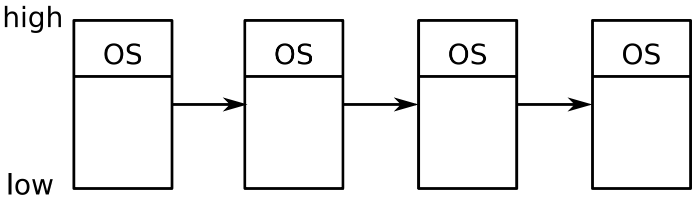


- when process is executed on a CPU it generates a sequence of addresses:
  memory protection[1]: base address and a limit
- CPU generates $\mathrm{V_A} \Rightarrow$ MMU $\Rightarrow \mathrm{P_A}$ to access main memory $\Rightarrow$ "relocatable" address[2]:
  CPU is running a user program and generates addresses 0,. . . , max
  corresponding $\mathrm{P_A}$: R+0,. . . , R+max (base=R)
  recall: translation invisible to the programmer !

[1] implemented in hardware!
[2] simplified

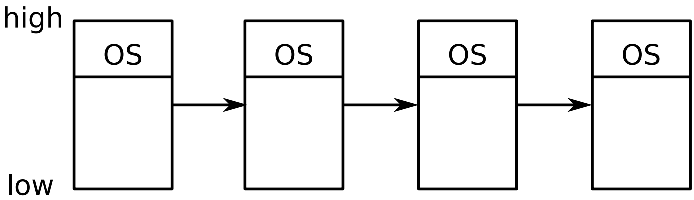
Contiguous memory allocation

- several user processes in main memory at the same time: how to allocate memory ?
- OS keeps a table with free/used memory locations



Contiguous memory allocation

- several user processes in main memory at the same time: how to allocate memory ?
- OS keeps a table with free/used memory locations



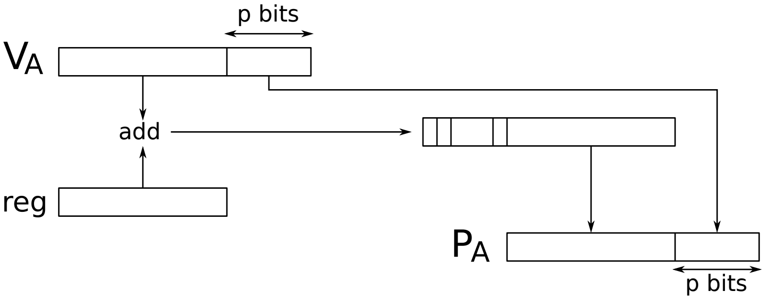
- OS must check if a new process fits into free space
- Dynamic storage allocation:
  - first fit: allocate first free “hole”
  - best fit: allocate smallest free “hole” that is big enough
  - worst fit: allocate largest “hole”
- (external) fragmentation → we will show one scheme that “solves” this problem

Paging

- we divide memory into fix-sized units and can now allocate memory in those units (several units per process, no need to be contiguous in phys. mem.)
- possible internal fragmentation:
  worst case scenario: T+1 units, where last unit only contains 1 byte
- fix-sized units
  - physical memory is divided into frames
  - virtual memory is divided into pages
- frames and pages are of same size and each page can map into any frame!
- paging address translation: using page table (PT) stored in physical memory
- page table contains descriptors (generated by the OS when when transferring the page into physical memory)
- each process has its own page table and CPU has a special register containing the address of the page table
- each descriptor contains permissions set by the OS (r,w,x)
  ⇒ paging solves the problem with aliases and offers (some) memory protection

Paging

- V_A consists of virtual page number (VPN) and offset (p-bits for offset)
- P_A consists of frame number (FN) and offset (p-bits for offset)
- descriptor has the following flags:
  - V for "descriptor valid"
  - P for "page exists"
  - r,w,x permissions (trap is violated, OS invokes correct handler)
  - C for "changed" (dirty bit)
  - FN to generate the P_A



Paging

- on page fault (P=0) OS will choose a handler that will:
  - if dirty: write-back ($\mathrm{P_M} \rightarrow \mathrm{V_M}$)
  - choose a new page (replacement strategy - e.g., Least Recently Used - LRU)
  - transfer new page $\mathrm{V_M} \rightarrow \mathrm{P_M}$ and create descriptor
  - restart the insn
- if too many page faults $\Rightarrow$ thrashing $\mathrm{V_M} \leftrightarrow \mathrm{P_M}$