Programs and Processes

The Virtual Address Space

A typical process VAS space includes:
- user regions in the lower half
  - V->P mappings specific to each process
  - accessible to user or kernel code
- kernel regions in upper half
  - shared by all processes
  - accessible only to kernel code
- **Nachos**: process virtual address space includes only user portions.
  - mappings change on each process switch

A VAS for a private address space system (e.g., Unix) executing on a typical 32-bit architecture.
The Program and the Process VAS

Header “magic number” indicates type of image.

Section table an array of (offset, len, startVA)

May be removed after final link step and strip.

BSS “Block Started by Symbol” (uninitialized static data)

Args/env copied in by kernel on exec.

Linking 101

May be removed after final link step and strip.
Virtual Memory Illustrated

executable file

header
text
idata
wdata
symbol table, etc.
program sections

virtual memory (big)

text
data

physical memory (small)

user stack
args/env
kernel

virtual-to-physical translations

pageout/eviction

page fetch

physical page frames

physical memory (small)

virtual memory (big)

virtual memory

data

executable file

BSS

program sections

user stack

args/env

kernel

MIPS instructions

executed by SPIM

Nachos: A Peek Under the Hood

shell

cp

user space

MIPS instructions
executed by SPIM

SPIM

MIPS emulator

ExceptionHandler()

Nachos kernel

Machine::Run()

fetch/execute
examine/deposit

SaveState/RestoreState
examine/deposit

process page tables

Machine object

page table

registers

memory

R0
SP
PC
The User-Mode Context for Nachos

boolean Machine::Translate(uva, alignment, &kva)
Translate user virtual address to a kernel memory address, checking access and alignment.

Memory Management and Protection
**Memory Management 101**

*Once upon a time...*memory was called “core”, and programs ("jobs") were loaded and executed one by one.

- load image in contiguous physical memory
  - start execution at a known physical location
  - allocate space in high memory for stack and data
- address text and data using physical addresses
  - prelink executables for known start address
- run to completion

---

**Memory and Multiprogramming**

One day, IBM decided to load multiple jobs in memory at once.

- improve utilization of that expensive CPU
- improve system throughput

*Problem 1*: how do programs address their memory space?

*Problem 2*: how does the OS protect memory from other programs?
Base and Bound Registers

*Goal*: isolate jobs from one another, and from their placement in the machine memory.

- addresses are offsets from the job’s *base address*
  - stored in a machine *base register*
  - machine computes *effective address* on each reference
    - initialized by OS when job is loaded
- machine checks each offset against job size
  - placed by OS in a *bound register*

Base and Bound: Pros and Cons

*Pro*:
- each job is physically contiguous
- simple hardware and software
- no need for load-time relocation of linked addresses
- OS may swap or move jobs as it sees fit

*Con*:
- memory allocation is a royal pain
- job size is limited by available memory
Variable Partitioning

Variable partitioning is the strategy of parking differently sized cars along a street with no marked parking space dividers.

Fixed Partitioning

Fixed partitioning incurs wastage due to internal fragmentation.
The Storage Allocation Problem

- fixed partitioning leads to *internal fragmentation*
- variable partitioning leads to *external fragmentation*
  which partition to choose? *first fit, best fit, worst fit, next fit?*
  these strategies don’t help much
- external fragmentation can be fixed by:
  - compaction (e.g., *copying garbage collection*)
  - coalescing (e.g., *buddy system*)
- these issues arise in *heap managers*
  e.g., runtime support for C++ *new* and *delete*

Managing Storage with Pages or Blocks

*Idea:* allow *noncontiguous* allocation in fixed blocks.
- partition each (file, memory) into *blocks* of $2^{**N}$ bytes
- partition storage into *slots* of size $2^{**N}$ bytes
  - blocks are often called *logical blocks or pages*
  - slots are often called *physical blocks or frames*

Paged allocation simplifies storage management:
- allocate a slot for each block independently
- slots are reusable and interchangeable
  - no need to search for a “good” slot; any free one will do
- no external fragmentation; low internal fragmentation
Translating the Logical Address Space

**Problem:** the system must locate the slot for each block on-the-fly as programs reference their data.

Applications name data through a *logical address space* that isolates them from the details of how storage is allocated.

Translate addresses indirectly through a *logical-physical map.*

The map $M$ is a function that maps a *logical block* number in the address space to a *physical slot* number in storage.

$$\text{slot\_index} = \text{Map}(\text{logical\_address} \gg N)$$

Block offset (low-order $N$ bits of the address) is unchanged.

$$\text{offset} = \text{logical\_address} \& ((2^{**N}) - 1)$$

$$\text{physical\_address} = (\text{slot\_index} \ll N) + \text{offset}$$

Examples of Logical-to-Physical Maps

1. **files:** inode block map
   - logical-physical map is part of the file metadata (*inode*)
     - map grows dynamically; file’s byte length is stored in the inode
   - the block map is stored on disk and cached in memory
   - block size is a power-of-two multiple of the disk sector size

2. **virtual memory:** page tables
   - $\text{virtual address} = \text{virtual page number} + \text{offset}$
   - page table is a collection of *page table entries (ptes)*
   - each valid pte maps a virtual page to a *page frame number*
Virtual Memory Illustrated

Virtual Address Translation

**Example:** typical 32-bit architecture with 8KB pages.

Virtual address translation maps a *virtual page number* (VPN) to a *physical page frame number* (PFN): the rest is easy.

Deliver exception to OS if translation is not valid and accessible in requested mode.

physical address { PFN + offset }
## Safe System Calls

Kernel must validate arguments to system calls.

- **scalar arguments**
  - must be valid values, e.g., syscall number, open file number

- **buffer arguments**
  - [addr, addr+len-1] must be mapped in current space and accessible from user mode with requested permission

- **string arguments**
  - must be null-terminated and [addr, addr+strlen(addr)-1] must be mapped and accessible with requested permission

- **all arguments**
  - should be copied into the kernel so they cannot be modified by a different user thread after validation

## Handling a System Call (Revisited)

1. Decode and validate by-value arguments.
2. Validate by-reference (pointer) IN arguments.
   - Validate user pointers and copy into kernel memory with `copyin`.
   - Kernel has access to user memory: syscall executes in process context with user space addresses mapped.
3. Call internal routines that implement the service.
   - Copy into user memory with `copyout`.
5. Set up registers with return value(s); return to trap handler.
Nachos: A Peek Under the Hood

SPIM
MIPS emulator

ExceptionHandler()

Nachos
kernel

Machine::Run()

fetch/execute
examine/deposit

SaveState/RestoreState
examine/deposit

Machine::Translate(uva, alignment, &kva)

boolean Machine::Translate(uva, alignment, &kva)

Translate user virtual address to a kernel memory address, checking access and alignment.

The User-Mode Context for Nachos