Overview

Let’s explore how the OS virtualizes memory.

This section should help us answer the following question:

- **How can the OS represent a private, possibly huge address space for multiple running processes that all share memory on top of a single physical memory?**

Introduction

Early machines didn't provide much **memory abstraction** for users. The machine's physical memory looked a lot like the illustration to the left.

The OS was:
* A set of routines (a library) in memory
* Beginning at physical address 0 in this example.
* One running program (a process)
* This consumed the rest of memory (starting at physical address 64k in this example).

Multiprogramming and Time Sharing

Machines were expensive, so people began to share them more.

Multiprogramming became the prevalent, in which multiple processes could run at the same time, and the OS would switch between them, like when performing an I/O operation.

This improved **CPU efficiency** and efficiency gains were critical back when machines cost hundreds of thousands or even millions of dollars!

Interactivity became important because multiple users could be utilizing a system at the same time, each looking for a prompt answer from their current work, so the era of **time sharing** was born.

To implement **time sharing**, you could:
* Run one process for a short period, giving it complete access to all memory,
* Then stop it,
* Save its state to disk (including physical memory),
* Load another process's state, and
* Run it for a short time.

Unfortunately, this strategy is too slow, especially as memory increases.

While saving and restoring register-level state is quick, saving the complete contents of memory to disk is very slow.

The OS can then efficiently perform **time sharing** by leaving processes in memory while switching between them like in the graphic to the left.

In the diagram, three processes (A, B, and C) share $512KB$ of physical memory. On a single CPU, the OS runs one of the processes (say A), leaving B and C in the ready queue.

As time sharing became more common, the operating system had to adapt. Allowing many processes to share memory makes protection a priority; you don't want one process to be able to read or worse, write over another's memory!

How can the operating system handle time sharing effectively?

The Address Space

The **address space** is the part of the computer operating system where a program's view of memory is stored, and it is what the running application sees. It provides the OS with an abstraction of physical memory.

A process' **address space** contains all of the program's memory state, including the **code**. The program uses a **stack** to monitor its position in the function call chain, allocate variables, and pass parameters. Finally, the **heap** is used for dynamically-allocated, user-managed memory, like when using a `malloc()` call.

The graphic to the left shows a small, **16kb address space**. The program code occupies the top 1KB of the address space. Given that code is static (and easy to store in memory), we can put it at the top of the address space and not worry about it.

The **heap** and the **stack** are the two address space areas that can increase (or shrink) while the program runs. We put them at opposing ends of the address space so they can grow in opposite directions. The heap starts at 1KB and increases downward (for example, when a user calls `malloc()`); the stack starts at 16KB and rises upward (say when a user makes a procedure call).

Fill in the blanks to complete the statements below.

Virtualizing Memory

How can the OS build this abstraction of a private, potentially massive address space for several running processes (all sharing memory) on top of a single, physical memory?

When we talk about the address space, we're talking about the OS's **abstraction** of the address for the running program.

We call this **virtualizing memory**, because the running program *believes* it's been put into memory at a specific address (say 0) and has a potentially very large address space (say 32-bits or 64-bits), but the truth is quite different.

For example, when process A in the graphic below tries to do a load at **virtual address** 0, the OS, in cooperation with some hardware support, has to make sure that the load is sent to physical address 320KB rather than physical address 0 itself (where A is loaded into memory).

This is the key to memory virtualization.

Where's my Program?

EVERY ADDRESS YOU SEE IS VIRTUAL

Have you ever written a C program and printed a pointer? The large (usually hexadecimal) number that you see as a result is the **virtual address**.

The location of your program's code is also a virtual address if you can print it. In fact, as a user-level software writer, every address you see is a virtual address.

Only the OS knows where these instructions and data values are located in the machine's physical memory. It's important to remember that the addresses you see on screen are only virtual and essentially an illusion. Only the OS (and the hardware) knows the true ones.

The program to the left, (`va.c`), prints out:

- The locations of the `main()` routine (where code lives),
- The value of a heap-allocated value returned from `malloc()`, and
- The location of an integer on the stack

Copy and paste the command below to compile the code.

```
gcc -o va va.c -Wall
```

Then type `./va` in the terminal to run it.

From this, we see that:
* Code comes first in the address space
* The heap is next
* The stack is all the way at the other end of this large virtual space.

All of these addresses are virtual, and will be translated by the OS and hardware in order to fetch values from their true physical locations.

Goals

The OS's job is to virtualize memory. To make sure the OS does so, we need specific goals.

- **Transparency** - The OS should implement virtual memory in a way that the program cannot see.
  - It should not be aware that memory is virtualized, but should operate as if it had its own private physical memory.
  - The OS (and hardware) provides the illusion by multiplexing memory among multiple jobs.
- **Efficiency** - The OS should optimize virtualization for time (i.e., not slowing down programs) and space (i.e., not using too much memory for structures needed to support virtualization).
  - To achieve time-efficient virtualization, the OS will need hardware characteristics like TLBs (which we will learn about later).
- **Protection** - The OS should protect processes against each other and from itself.
  - When a process loads, stores, or fetches instructions, it shouldn't be able to access or change the memory contents of other processes or the OS (or anything outside its address space). Protection lets us guarantee process isolation and security from the horrors of other malfunctioning or malicious processes.

Fill in the blank to complete the statements below.

Summary

- **Virtual memory (VM)** is a major OS subsystem.
- The **VM** system provides programs with the **abstraction**, or illusion of a large, limited, private address space in which to store their instructions and data.
- The OS will convert each of these virtual memory references into **physical addresses** that may be given to the physical memory to retrieve the desired information.
- The OS' goals are to provide, **transparency**, **efficiency**, and **protection** for processes to effectively virtualize memory.

Lab 1

Let's explore some of the many useful tools for examining **virtual memory usage** on Linux-based systems.

1. Let's first look at the `free` tool by typing `man free` into the terminal and checking out its manual page.

2. Run `free` using the `-m` flag to show memory totals in megabytes.

You should be able to see:
* How much total memory is in your system
* How much is being used, and
* How much is free

Take some time to explore other flags and see how they relate (`-kilo`, `-b`, `-mega`, etc.)

Lab 2

Write a program, `memory-user.c`. This program should:
* Take one command line argument: The number of megabytes of memory it will use.

- When it's run, it should allocate an array and constantly loop through the array, accessing each entry.
- The program should do this indefinitely.

For example, you would umallse the code below to run your program for 16 megabytes of memory.

```
gcc -o memory-user memory-user.c -Wall
./memory-user 16
```

- While running this program in one terminal, open a different terminal window and run the `free` tool to observe the memory usage totals.
- `kill` the `memory-user` program and see how the memory usage changes.
- Run the `memory-user` program with different amounts of memory usage and run free to see how your system is affected.