Virtual Memory

solves lots of problems. creates some, too!

As a program runs, it generates memory accesses.
The program's view of what addresses it may access is its address space.

For example,
size of address: 32 bits byte addressable

address space is

0 to $2^{32} - 1$

(4 Gbytes)

0x00000000 to 0xffffffff
Without virtual memory, our model:

\[ P \leftrightarrow \square \leftrightarrow \text{(textbook) DRAM} \leftrightarrow \text{disk} \]

main memory

physical memory

RAM

Memory access addresses generated by the running program are physical addresses.
Historically, for a 32-bit address space, physical memory was considerably **smaller** than the address space.

(WITHOUT VIRTUAL MEMORY) use **OVERLAYS**

divide program into blocks of object code called **OVERLAYS**

always resident

siblings of tree: one can be resident

O.S. has **OVERLAY MANAGER**

Program itself is aware of overlaps & calls manager functions to do switch.
Physical memory can still be smaller than address space.

AND

We want a timeshare OS
multiprogrammed OS

AND

We want to eliminate the ability of 1 program to trash another's memory (malicious or innocent bug).

addr

0

\begin{array}{|c|c|}
\hline
\text{OS} & \text{Assume P1 running.} \\
\hline
\text{P1 code} & \text{Assume } \%\text{eax has a large, negative value.} \\
\hline
\text{P2 data} & \text{movl } \$-1, (\%\text{eax}, \%\text{esp}) \\
\hline
\text{P1 stack} & \%\text{esp} \\
\hline
\end{array}

(want isolation)
AND

* (eventually) we want portions of memory to be purposely shared by programs.

For example, the object code of the C stdio library.

Virtual memory can do all these things.
We might: \textcolor{blue}{P12} as a program is to run, OS brings entire program into main memory.

When O.S. decides to run another program \textcolor{green}{P2}, it \textcolor{red}{swaps} out \textcolor{blue}{P1} (write to disk), and brings into main memory all of \textcolor{green}{P2}.

+ simple
+ isolation
- inefficient
- no sharing
- doesn't address problem of physical memory being smaller than address space needed.
Implementation called base & bounds

If physical memory is big enough, OS can have multiple programs in main mem at same time.
Addresses generated by the running program are virtual addresses. HW does a translation of virtual → physical.

HW maintains Base where the program starts in physical mem. Bounds size of program.

---

Diagram:

Virtual address

+ physical addr

OK

no

Bounds

Yes

fault!

addr is outside program's space
The contiguous allocation of memory of base + bounds makes it unworkable.

Better:

let each program have a set of variable-sized segments.
(how about code, data, stack, & heap?)

OS keeps a table of base + bounds pairs for each running program.
processor keeps
ST Base Reg  ST SizeReg
(to locate segment table)
Issues:

Where is the segment table?

How many memory accesses are there?
If there are a fixed number of segments, the segment table can be in dedicated registers.

The Pentium does this.....

CS
DS
SS
ES  (extra segment)
FS
GS

We do not use msbs of virtual address as segment #.

Which segment register is used is often implied.
Segmentation

+ variable sized segments means each can be the exact size it needs to be
- external fragmentation

```
P1  |  P2  |  P2  |  |
```

↑ nothing can fit here

+ multiple programs' segments can reside in memory at the same time (if there is enough space)
+ could implement sharing on a per-segment basis
+ a segment (of a program) that is never referenced can remain on the disk
- changing a segment's size is not easy
**paging**

divide memory into fixed-size chunks (pages).

OS does disk \(\rightarrow\) physical memory transfers of a page.

Called **swapping**.

\{It behaves similar to a cache, where physical memory is a cache of pages from the disk.\}

The textbook introduces the topic from this point of view.
A virtual address has a fixed size.

The choice of a page size determines the number of bits of offset within a page.

VA (Virtual Address)
A program has a fixed number of pages in its address space.

A page can end up in any page frame.

We need a mapping of which page frame has a page. Called page table.

Each element of page table is called Page Table Entry.
Consider page table size.

Per program:

\[ \#PTE = 2^{VA_{\text{size}} - \text{offset size}} \]

Example: 32-bit VA
4 Kbyte page size

\[ 2^{32-12} = 2^{20} = 1 \text{ M pages!} \]

If each PTE is 4 bytes (like Intel), it takes 4 Mbytes of memory for each page table.
Consider contents of a PTE.

→ Physical frame's base address (PFN)

→ Present bit whether this page is currently in physical memory or out on disk


→ Dirty bit

→ Reference bit (access bit) might be used to identify victim for swapping.
x86 PTE

<table>
<thead>
<tr>
<th></th>
<th>PFN</th>
<th>GT</th>
<th>PAT</th>
<th>Dirty</th>
<th>Access</th>
<th>PCD</th>
<th>PWT</th>
<th>U/S</th>
<th>R/W</th>
<th>Present</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

user/kernel

PWT, PCD, PAT, GT
determines caching details
Consider a single memory access.
We desperately need a cache of PTEs

Called
Translation
Lookaside
Buffer