Memory

General purpose computers typically contain 4-128GB of volatile Random Access Memory (RAM)

Single Process Resident in RAM without Operating System

- Many small embedded systems run without an operating system.
- Single program running, typically written in C, perhaps with some assembler.
- Devices (sensors, switches, ...) often wired at particular address.
- E.g., motor speed can be set by storing a byte at 0x100400.
- Program accesses (any) RAM directly.
- Development and debugging tricky.
  - Might be done by sending ASCII values bit by bit on a single wire.
- Widely used for simple micro-controllers.
- Parallelism and exploiting multiple-core CPUs problematic.
Single Process Resident in RAM with Operating System

- Operating systems need (simple) hardware support.
- Part of RAM (kernel space) must be accessible only in a privileged mode.
- System call enables privileged mode and passes execution to operating system code in kernel space.
- Privileged mode disabled when system call returns.
- Privileged mode could be implemented by a bit in a special register.
- If only one process resident in RAM at any time - switching between processes is slow.
- Operating system must write out all RAM used by old process to disk (or flash) and read all memory of new process from disk.
- OK for some uses, but inefficient in general.
- Little used in modern computing.

Multi Processes Resident in RAM without Virtual Memory

- If multiple processes to be resident in RAM operating system can swap execution between them quickly.
- RAM belonging to other processes & operating system operating system must be protected.
- Hardware support can limit process accesses to particular segment (region) of RAM.
- BUT program may be loaded anywhere in RAM to run.
- Breaks instructions which use absolute addresses, e.g.: lw, sw, jr.
- Either programs can’t use absolute memory addresses (relocatable code).
- Or code has to be modified (relocated) before it is run - not possible for all code!
- Major limitation - much better if programs can assume always have same address space.
- Little used in modern computing.

Virtual Memory

- Big idea - disconnect address processes use from actual RAM address.
- Operating system translates (virtual) address a process uses to an physical (actual) RAM address.
- Convenient for programming/compilers - each process has same virtual view of RAM.
- Can have multiple processes to be in RAM, allowing fast switching.
- Can load part of processes into RAM on demand.
- Provides a mechanism to share memory between processes.
- Address to fetch every instruction to be executed must be translated.
- Address for load/store instructions (e.g. lw, sw) must be translated.
- Translation needs to be really fast - needs to be largely implemented in hardware (silicon).
Virtual Memory with One Memory Segment Per Process

Consider a scenario with multiple processes loaded in memory:

<table>
<thead>
<tr>
<th>[0] proc1 memory</th>
<th>unused</th>
<th>proc3 memory</th>
<th>proc4 memory</th>
<th>unused</th>
<th>proc6 memory</th>
</tr>
</thead>
</table>

- Every process is in a contiguous section of RAM, starting at address base finishing at address limit.
- Each process sees its own address space as [0 .. size - 1]
- Process can be loaded anywhere in memory without change.
- Process accessing memory address a is translated to a + base and checked that a + base is < limit to ensure process only access its memory
- Easy to implement in hardware.

Virtual Memory with One Memory Segment Per Process

Consider the same scenario, but now we want to add a new process.

<table>
<thead>
<tr>
<th>[0] proc7 memory</th>
<th>[max-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>proc1 memory</td>
<td>unused</td>
</tr>
</tbody>
</table>

- The new process doesn’t fit in any of the unused slots (fragmentation).
  - Need to move other processes to make a single large slot

<table>
<thead>
<tr>
<th>[0] proc1 memory</th>
<th>proc4 memory</th>
<th>proc3 memory</th>
<th>proc7 memory</th>
<th>unused</th>
<th>proc6 memory</th>
</tr>
</thead>
</table>

- Slow if RAM heavily used.
- Does not allow sharing or loading on demand.
- Limits process address space to size of RAM.
- Little used in modern computing.

Virtual Memory with Multiple Memory Segments Per Process

Idea: split process memory over multiple parts of physical memory.

<table>
<thead>
<tr>
<th>[0] proc7 memory</th>
<th>[max-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>proc1 memory</td>
<td>unused</td>
</tr>
</tbody>
</table>

becomes

<table>
<thead>
<tr>
<th>[0] proc1 memory</th>
<th>proc7 memory1</th>
<th>proc3 memory</th>
<th>proc4 memory</th>
<th>proc7 memory2</th>
<th>unused</th>
<th>proc6 memory</th>
</tr>
</thead>
</table>

| [0] [a-1]        | [a] [p7size-1] |
Implications for splitting process memory across physical memory

- Each chunk of process address space has its own base
- Each chunk of process address space has its own size
- Each chunk of process address space has its own memory location

Need a table of process/address information to manage this

With arbitrary sized memory segments hardware support is difficult
A side-effect of this type of virtual → physical address mapping:
- don’t need to load all of process’s pages up-front
- start with a small memory “footprint” (e.g. main + stack top)
- load new process address pages into memory as needed
- grow up to the size of the (available) physical memory

The strategy of...
- dividing process memory space into fixed-size pages
- on-demand loading of process pages into physical memory

is what is generally meant by virtual memory.

---

**Virtual Memory**

4096 bytes is a common pages/frame size, but sizes 512 to 262144 bytes used

With 4GB memory, would have ≈ 1 million × 4KB frames

Each frame can hold one page of process address space

Leads to a memory layout like this (with L total pages of physical memory):

```plaintext
[0]  [1]  [2]  [3]  ...  [L-1]

proc1 page5  proc7 page1  proc1 page0  proc1 page1  free  proc4 page1  proc7 page3
```

When a process completes, all of its frames are released for re-use

---

**Virtual Memory - Loading Pages**

Consider a new process commencing execution...
- initially has zero pages loaded
- load page containing code for main()
- load page for main()’s stack frame
- load other pages when process references address within page

Do we ever need to load all process pages at once?
Virtual Memory - Working Sets

From observations of running programs ...

- In any given window of time, process typically access only a small subset of their pages
- Often called *locality of reference*
- Subset of pages called the *working set*

Implications:

- If each process has a relatively small working set, can hold pages for many active processes in memory at the same time
- If only need to hold some of process’s pages in memory, process address space can be larger than physical memory

Virtual Memory - Loading Pages

We say that we “load” pages into physical memory

But where are they loaded from?

- Code is loaded from the executable file stored on disk into read-only pages
- Some data (e.g. C strings) also loaded into read-only pages
- Initialised data (C global/static variables) also loaded from executable file
- Pages for uninitialised data (heap, stack) are zero-ed
  - Prevents information leaking from other processes
  - Results in uninitialised local (stack) variables often containing 0

Virtual Memory - Loading Pages

We can imagine that a process’s address space ...

- Exists on disk for the duration of the process’s execution
- And only some parts of it are in memory at any given time

<table>
<thead>
<tr>
<th>Process Address Space</th>
<th>[0]</th>
<th>page0</th>
<th>page1</th>
<th>page2</th>
<th>page3</th>
<th>page4</th>
<th>...</th>
<th>pageK-2</th>
<th>pageK-1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Physical Memory</th>
<th>[0]</th>
<th>page4</th>
<th>...</th>
<th>page0</th>
<th>page1</th>
<th>...</th>
<th>pageK-1</th>
<th>...</th>
<th>pageK-2</th>
<th>...</th>
<th>[L-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>t=4</td>
<td></td>
<td></td>
<td></td>
<td>t=1</td>
<td>t=3</td>
<td></td>
<td>t=1</td>
<td></td>
<td>t=1</td>
<td></td>
<td>t=2</td>
</tr>
</tbody>
</table>

Transferring pages between disk↔memory is very expensive

- Need to ensure minimal reading from / writing to disk
Virtual Memory - Handling Page Faults

An access to a page which is not-loaded in RAM is called a page fault.

Where do we load it in RAM?

First need to check for a free frame

- need a way of quickly identifying free frames
- commonly handled via a free list

What if there are currently no free page frames, possibilities:

- suspend the requesting process until a page is freed
- replace one of the currently loaded/used pages

Suspending requires the operating system to

- mark the process as unable to run until page available
- switch to running another process
- mark the process as able to run when page available

Virtual Memory - Read-only Pages

Virtual memory allows sharing of read-only pages (e.g. for library code)

- several processes include same frame in virtual address space
- allows all running programs to use same pages for e.g. C library code (printf)

Memory Management Hardware

Address translation is very important/frequent

- provide specialised hardware (MMU) to do it efficiently
- sometimes located on CPU chip, sometimes separate