For multiprogramming, must have multiple processes
Each process must be in memory in order to execute
Possibilities
- One process in memory. **Swapping**: Swap it out to disk, swap in a new one

Internal Fragmentation—Fragmented memory within allocated memory blocks

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>process 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a+3n</td>
<td>process 2</td>
<td>process 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a+2n</td>
<td></td>
<td></td>
<td>physical memory</td>
<td></td>
</tr>
<tr>
<td>a+n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

External Fragmentation—Fragmented memory between allocated memory blocks

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>process 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>process 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>process 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>physical memory</td>
</tr>
</tbody>
</table>
Placement Algorithms

Need to allocate memory block of size $n$

First-fit
- Find first block $\geq n$

Best-fit
- Find smallest open memory block ($\leq 2n$)

Worst-fit
- Find largest open memory block ($\leq 2n$)

Next-fit
- Starting with last open block considered, find next block $\geq n$

Buddy system
- Round $n$ up to a power of 2. Free memory is kept in powers of 2. If no block of size $n$, split a larger chunk in half until size $n$ is found.
- When a block is freed, if its buddy (of the same size) is free, merge them together (recursively)

Address Binding

When to map instructions and data references to actual memory locations

Link time
- Absolute addressing. Works if well-known address at which user programs are loaded

Load time
- Linker-loader relocates code as it loads it into memory
  - Change all data references to take into account where loaded
- Position-independent code.
  - On Mac OS 9, subroutine calls are PC-relative; data references are relative to A5 register. Linker sets A5 register at load time

Execution time
- Code uses logical addresses which are converted to physical addresses
- Hardware: relocation register

Process Protection

How to protect the memory of a process (or the OS) from other running processes?
- Hardware solution: base (aka relocation) and limit registers
  - Disadvantage: can’t easily share memory between processes
- Software:
  - Tagged data (e.g., Smalltalk, Lisp). No raw pointers
  - Virtual machine

Within a Processes Address Space
Fragmentation

External Fragmentation
- Space wasted between allocated memory blocks

Internal Fragmentation
- Space wasted within allocated memory blocks

Solutions to external fragmentation
- Compaction
  - Move blocks around dynamically to make free space contiguous
  - Only possible if relocation is dynamic at execution time
- Non-contiguous
  - Don’t require all of the memory of a process to be contiguous
    - Segmentation
    - Paging

Solutions to internal fragmentation
- Don’t use fixed-size blocks

Segmentation

Provides multiple address spaces
- Handy for separate data/stack/code
- Good for sharing code/data between processes
- Easy way to specify protection
  - No execute on stack!
- Address is (segment_number offset within segment)
- Need segment base/segment limit registers (array of registers)
- Programmer (or compiler) must specify different segments

Pros
- Easy to share segments between processes
- Segment-specific protection
- Programmer/Compiler/Linker must be aware of different segments

Cons
- Still must worry about external fragmentation: each segment must have contiguous physical memory

Segmentation Example

8-bit address:
- 2 bits segment
- 6 bits offset

Example code
- 0x04: load 0x19, r0
- 0x06: push r0, sp
- 0x08: add sp, #2
Swapping

Swapping moves the memory of a segment (or entire process) out to disk:
- Process can’t run without its memory
- On context switch in, memory is loaded back from memory

Slow since can be lots of data

Paging

Map contiguous virtual address space to non-contiguous physical address space:
- Idea:
  - Break virtual address space into fixed-size pages (aka virtual pages)
  - Map those fixed-size pages into same-sized physical frames (aka physical pages)

Implementing Paging

Page table:
- Index into table is page number.
- Each entry is a page table entry:
  - Frame number
  - Valid/invalid bit
  - Permissions
  - ...

Usually done in hardware (Memory Management Unit—MMU):
- Page-Table Base Register and Page-Table Limit Register
  - Must be saved/restored on a context-switch
- Free list of frames
- Page-table per process (or per segment)

convertToPhysicalAddress(physicalAddress) {
    pageNumber = upper bits of physicalAddress
    if pageNumber out of range, generate exception (aka fault)
    if pageTable[pageNumber].invalid generate page fault
    offset = lower bits of physicalAddress
    upper bits of virtualAddress=pageTable[pageNumber].frameNumber
    lower bits of virtualAddress = offset
}

Advantages:
- No external fragmentation
- Unallocated memory can be allocated to any process
- Transparent to programmer/linker/compiler
- Can put individual frames out to disk (virtual memory)

Disadvantages:
- Translating from virtual to physical address requires an additional memory access
- Unused pages in a process still require page table entries
- Internal fragmentation
  - On average, 1/2 frame size per segment
Relationship between physical and virtual addresses

Virtual address same size as physical address

Virtual address smaller than physical address

Virtual address bigger than physical address

Dealing with holes in virtual address space

With large address space (and small pages), page table can be very large
- 32-bit virtual addresses, 4KB page: $2^{20}$ page table entries per process

Multilevel page table
- Virtual address broken up into multiple page numbers: PT1 and PT2.
  - PT1 used as index into top-level page table to find second-level page table
  - PT2 used as index into second-level page table to find page table entry
- If parts of address space are unused, top-level page table can show second-level page table not present

Dealing with holes in virtual address space

Combination of segmentation/paging
- Top $n$ bits are segment number
  - Except some machines, current segment stored in separate register, not in address itself
- Next $m$ bits are page number
- Remainder are offset

Segmentation size: 500
Segment size: 200

Speed

Imagine memory access is 100ns

How long to access memory using paging?
- 100ns to access page table entry
- 100ns to access physical memory address

Using virtual addresses slows down by factor of 2

Solution
- Use a cache
  - Definition: copy on a faster medium of data that resides elsewhere
  - Examples:
    - Caching contents of files in memory
    - Caching contents of memory in off-chip (Level 2) cache
    - Caching contents of off-chip cache in on-chip (Level 1) cache
    - Caching contents of page-table entries
  - Kinds of cache:
    - Write-back (changes to data are made in cache, written to permanent storage later)
    - Write-through (changes to data are made in cache and permanent storage)
Translation Lookaside Buffer (TLB) Implemented in Hardware

- Cache to map virtual page numbers to page frame
  - Associative memory: HW looks up in all cache entries simultaneously
  - Usually not big: 64-128 entries
- TLB entry:
  - page number
  - Valid
  - Modified
  - Protection
  - Page frame
- If not present, do ordinary lookup, then evict entry from TLB and add new one
  - Evict which entry?
- Serial/Parallel lookup
  - Serial: First look in TLB. If not found, then look in page table
  - Parallel: Look in TLB and in page table in parallel. If not found in TLB, then page table lookup already in progress.

Software TLB Management

- MMU doesn’t handle page tables; software does
- On a TLB miss, generate a TLB fault and let OS deal with it
  - Search a larger memory cache.
  - Page containing cache must be in TLB for speed
  - If not in cache, search page table
  - Once page frame, etc. found, update TLB

Why not use hardware?

- Logic to search page table takes space on the die
- Spend die size alternatively:
  - Increase Memory cache
  - Reduce cost/power consumption

TLB Summary

Cost Example

- Direct memory access: 100ns
- Without TLB: 200ns (lookup in Page Table first)
- With TLB
  - Assume cost of TLB lookup is 10ns
  - Assume TLB hit rate is 90%
    - Serial lookup: Average cost = .9*10ns + .1*200ns = 119ns
    - Parallel lookup: Average cost = .9*10ns + .1*(200ns-10ns) = 118ns

Caches are very sensitive to:

- Hit rate
- Cost of cache miss

Note that TLB must be flushed on context switch

- Unless TLB entries include process ID

Inverted Page Tables

Traditional page tables: 1 entry/virtual page
Inverted page tables: 1 entry/physical frame of memory

Why? Size

- 64-bit virtual addresses, 4KB page 256MB of RAM per process. Inverted page table needs 65536 entries

Page Table Entry:

- Process ID
- Virtual page number
- Additional PTE info

Slow to search through table with 65536 entries

- Solution: Hash table. Key is virtual page number. Entry contains virtual page, process ID and page frame

Advantage:

- Page table memory is proportional to physical memory
  - Not logical address space
  - Not number of processes

Disadvantage:

- Hard to share memory between processes
Inverted Page Tables

Hash table
- Space: proportional to number of allocated memory frames
  - 1 entry in hash table for each allocated page

Segmentation vs. Paging

<table>
<thead>
<tr>
<th>Need the programmer be aware the technique is being used?</th>
<th>Segmentation</th>
<th>Paging</th>
</tr>
</thead>
<tbody>
<tr>
<td>How many linear address spaces are there?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Can the total address space exceed the size of phys. mem?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Can procedures and data be distinguished and separately protected?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Can tables whose size fluctuates be accommodated easily?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is sharing of procedures between users facilitated?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page Fault Handling

MMU generates Page Fault (protection violation or page not present).

Page fault handler must:
- Save registers
- Figure out virtual address that caused fault
  - Often in hardware register
- If writing to page after currently-allocated stack
  - Allocate free page to add to stack
  - Update page-table
  - Restart instruction for fauliting process
  - Must undo any partial effects
- Else, signal or kill process