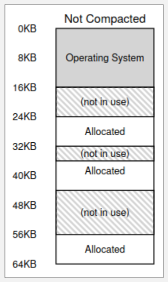
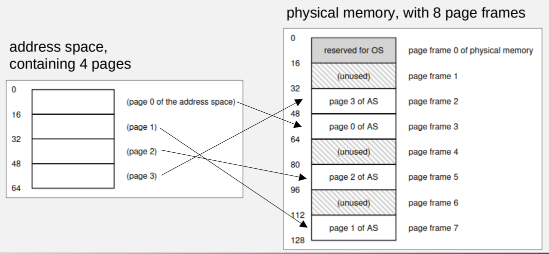
Drawbacks and Limitations – Segmentation

- simplifying assumptions
  - ~~address space~~ segment occupies contiguous chunk of physical memory
  - ~~address space~~ segment fits entirely within physical memory
  - ~~address spaces are all the same size~~
- *external fragmentation*
  - memory gets chopped up into a bunch of little pieces
    - there may be enough total space available for a new process or to grow a segment, but not in one chunk
- a large but sparse heap must still reside entirely in memory
  - unused space tied up with a process



Paging

- dividing memory into variable-sized pieces leads to problems with fragmentation
- instead, divide memory into fixed-sized *pages*
  - view physical memory as an array of fixed-sized *page frames*
    - each page frame can contain one virtual page



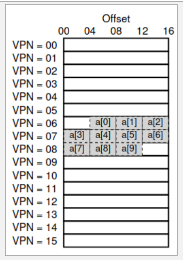
Recap: Paging Too Slow

- the extra memory access needed to look up the VPN→PFN translation in the page table is too slow
  - doubles the number of memory accesses required
- solution – utilize caching
  - add a *translation-lookaside buffer* (TLB) to the MMU
    - stores page table entries
  - hardware first looks up VPN in the TLB
    - on a TLB miss, hardware (for a hardware-managed TLB) or software (for a software-managed TLB) looks up the PTE in the page table, stores the entry in the TLB, and retries the operation
  - ideally, most of the time the desired PTE is in the TLB so the extra memory access is avoided

Example

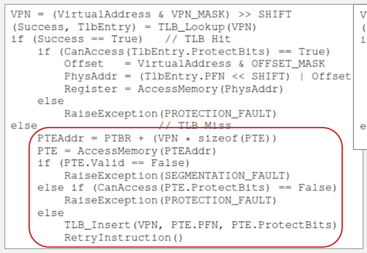
- accessing an array
  - consider only the array accesses

```
int i, sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```



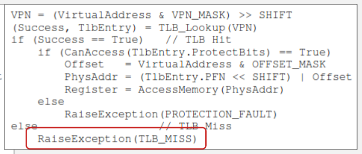
access a[0] – miss, load VPN 6 entry into TLB
access a[1] – hit
access a[2] – hit
access a[3] – miss, load VPN 7 entry into TLB
access a[4] – hit
access a[5] – hit
access a[6] – hit
access a[7] – miss, load VPN 8 entry into TLB
access a[8] – hit
access a[9] – hit

Recap: Handling TLB Misses



```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                  // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
    else
        TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
        RetryInstruction()
```

- hardware-managed TLB (CISC)
  - TLB miss is handled entirely by the hardware
  - PTBR stores the location of the page table in memory
  - hardware knows layout of page table to be able to look up the entry for a particular VPN
    - e.g. linear page table – index into an array



```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                  // TLB Miss
    RaiseException(TLB_MISS)
```

- software-managed TLB (RISC)
  - hardware raises an exception on a TLB miss
  - control transfers to a trap handler
    - trap handler looks up the VPN in the page table and updates the TLB (via a privileged instruction)
  - on return from trap, the hardware retries the instruction

Software-Managed TLB

- handling return-from-trap
  - normally execution resumes with the instruction *after* the trap into the OS
  - for a TLB miss, should retry the instruction that *caused* the trap
    - hardware must save a different PC when TLB miss causes the trap
- must prevent infinite chains of TLB misses
  - executing the trap handler involves loading instructions from memory...which could cause additional TLB misses if address translation is needed
  - solution strategies
    - avoid address translation: store TLB miss handlers in a reserved location in physical memory – physical address is known, so address translation is not needed
    - ensure TLB hit: reserve some TLB entries for *wired translations* that are always valid

TLB Details

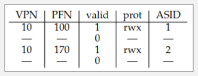
- caches must be small in order to be fast
  - TLBs typically have only 32, 64, or 128 entries
- TLBs are commonly *fully associative*
  - all locations are searched in parallel to find the desired translation

TLB Details

- typical TLB entry VPN | PFN | other bits
  - *valid bit* – whether or not entry holds a valid translation
    - not the same as the valid bit in a PTE!
      - PTE valid bit indicates if VPN is currently mapped to a PFN
  - *protection bits* – determine how a page can be accessed (read, write, execute)
  - *address-space identifier* (ASID) – to enable sharing of TLB between processes
  - *dirty bit* – identifying pages that have been written to

Handling Context Switches

- VPN→PFN translations are only valid for a particular process
  - TLB must be updated on context switches
- simple option: flush TLB on context switch
  - via privileged instruction for software-based TLB
  - when PTBR is changed for hardware-based TLB
  - can be implemented by setting the valid bit to invalid
  - flushing ensures no out-of-date translations, but results in large numbers of TLB misses immediately after a context switch
- alternative: use an *address-space identifier* (ASID)
  - shorter version of PID to enable sharing of TLB between processes
  - requires a register to store current ASID



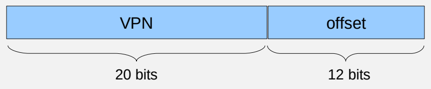
VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

TLB Replacement Policies

- the new entry brought in on a TLB miss usually *replaces* an existing one
- performance relies on TLB misses being rare
- common replacement policies
  - least-recently-used (LRU)
    - seeks to take advantage of locality – a page not used for a while probably isn't going to be the next one used
    - can often perform well, but worst case (every TLB access is a miss) if the program loops over ≥ n+1 pages with a TLB of size n
  - random
    - simple
    - randomness mitigates worst case behavior by spreading it out over all cases
      - instead of certain cases being reliably bad, bad performance is possible for any case with low probability

Too Big

- page tables as described so far are too large to be stored in registers
  - 32-bit address space (4GB memory) with 4KB pages means a 12-bit offset and a 20-bit VPN



- 4KB page means 4096 addresses
  - 12 bits = 2^{12} = 4096
- 20 bits = 2^{20} = 1,048,576 page table entries
  - 32 bit physical addresses means 4 bytes per entry = 4MB per process

- 4MB per process is too much memory
  - even worse with 64-bit address spaces

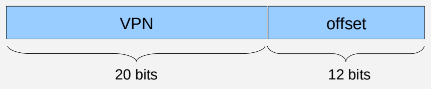
Too Big

Strategies for dealing with too big –

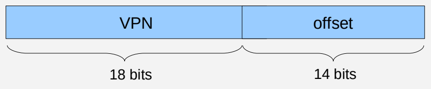
- bigger pages
- combine paging and segments
- multi-level page tables
- inverted page tables

Bigger Pages

- bigger pages means fewer PTEs for a given size of address space
  - for a 32-bit address space (4GB memory)…
  - …4KB pages means a 12-bit offset and a 20-bit VPN
    - 20 bits = 2^{20} = 1,048,576 page table entries
      - 32 bit physical addresses means 4 bytes per entry = 4MB per process



  - …16KB pages means a 14-bit offset and a 18-bit VPN
    - 18 bits = 2^{18} = 262,144 page table entries
      - 32 bit physical addresses means 4 bytes per entry = 1MB per process



Bigger Pages

- bigger pages mean more *internal fragmentation* due to unused space within pages

- many architectures support multiple page sizes to increase TLB performance
  - default is 4KB or 8KB pages but applications can request larger pages (e.g. 4MB)
  - allows databases or systems with large memory usage to fit an entire data structure into a single page to increase TLB performance
  - makes OS memory management more complex