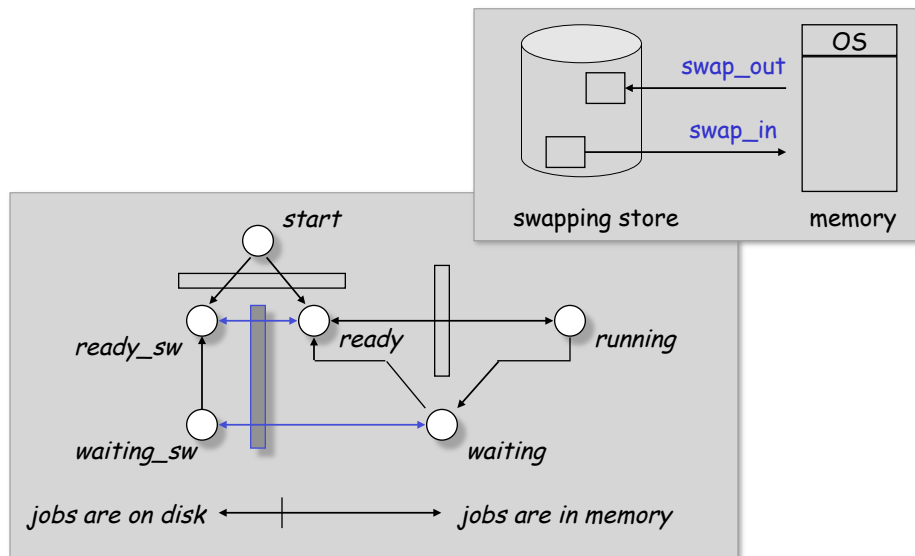# Memory Management

- Logical vs. physical address space
- Fragmentation
- Paging
- Segmentation

An expensive way to run multiple processes: Swapping

OS

swap_out

swap_in

swapping store   memory

start

ready_sw   ready   running

waiting_sw   waiting

jobs are on disk ◄─────┼─────► jobs are in memory

Memory Management

Observations:

1. Process needs at least **CPU** and **memory** to run.
2. CPU context switching is relatively **cheap**.
3. Swapping memory in/out from/to disk is **expensive**.

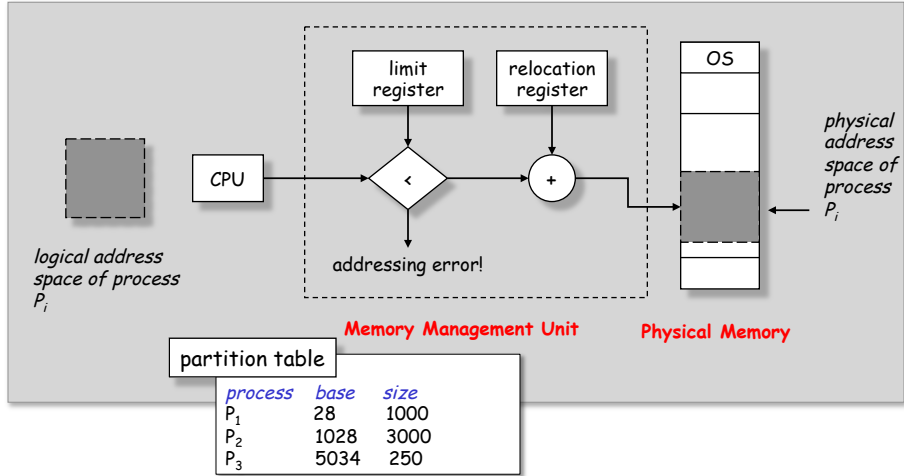So, we need to **subdivide** memory to accommodate **multiple processes**!

Q: How do we **manage** this memory?

Requirements for Memory Management

- **Relocation**
  - We do not know a priori where memory of process will reside.
- **Protection**
  - No uncontrolled references to memory locations of other processes.
  - Memory references must be checked at run-time.
- **Sharing**
  - Ability to share data portions and program text portions.
- Logical organization
  - Take advantage of semantics of use.
  - Data portions (read/write) *vs.* program text portions (read only).
- Memory hierarchy
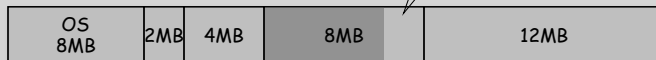 - RAM *vs.* secondary storage
  - Swapping

Logical *vs.* Physical Memory Space

- **Logical address**: address as seen by the process (i.e. as seen by the CPU).
- **Physical address**: address as seen by the memory.

logical address space of process P_i

limit register   relocation register

CPU $<$ addressing error! $+$

Memory Management Unit

OS

physical address space of process P_i

Physical Memory

partition table

process	base	size
P_1	28	1000
P_2	1028	3000
P_3	5034	250

Problem with simple Relocation: Fragmentation

Internal Fragmentation
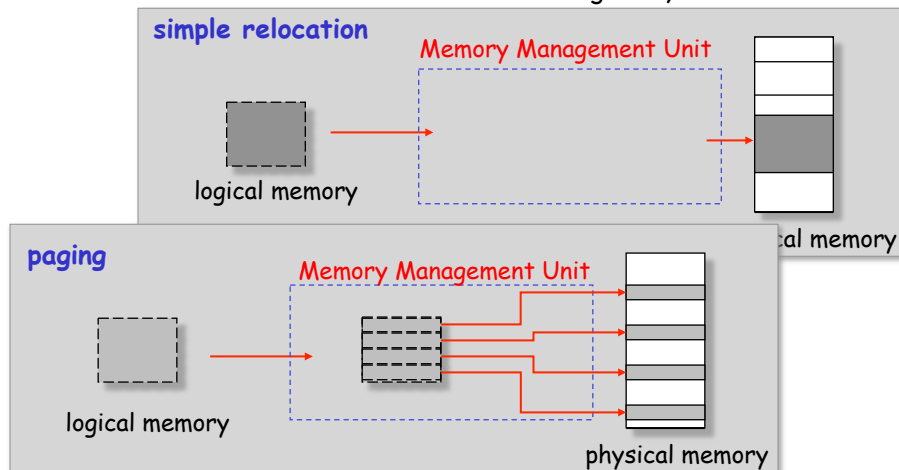
?!

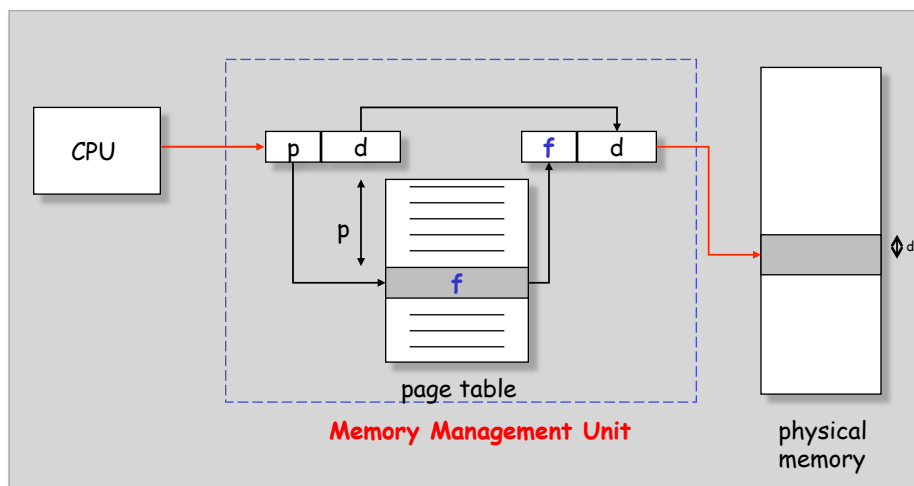| OS 8MB | 2MB | 4MB | 8MB | | 12MB |

External Fragmentation
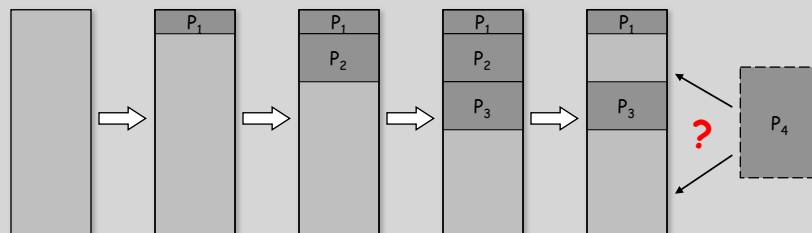
Paging

- Contiguous allocation causes (external) fragmentation.
- Solution: Partition memory blocks into smaller subblocks (pages) and allow them to be allocated non-contiguously.

simple relocation

Memory Management Unit

logical memory

physical memory

paging

Memory Management Unit

logical memory

physical memory

Basic Operations in Paging Hardware

CPU

| p | d |

| f | d |

p

| f |

page table

Memory Management Unit

physical memory

d

Example: PDP-11 (16-bit address, 8kB pages)

Internal Fragmentation in Paging

- Example:

page size 4kB

logical memory
13300B

physical memory

4084 bytes
wasted!
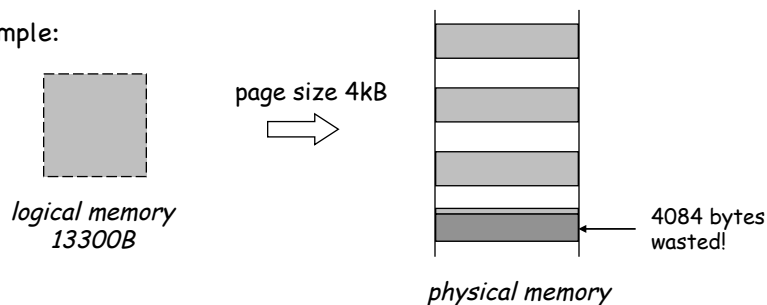
- Last frame allocated may not be completely full.
- Average internal fragmentation per block is typically half frame size.
- Large frames vs. small frames:
  - Large frames cause more fragmentation.
  - Small frames cause more overhead (page table size, disk I/O)

Implementation of Page Table

- Page table involved in every access to memory. Speed very important.
- Page table in registers?
  - Example: 1MB logical address space, 2kB page size; needs a page table with 512 entries!
- Page table in memory?
  - Only keep a page table base register that points to location of page table.
  - Each access to memory would require two accesses to memory!
- Cache portions of page table in registers?
  - Use translation lookaside buffers (TLBs): typically a few dozens entries.
  - Hit ratio: Percentage of time an entry is found. Hit ratio must be high in order to minimize overhead.

Hierarchical (Multilevel) Paging

- Problem: Page tables can become <u>very</u> large! (e.g. 32-bit address space?)
- Solution: Page the page table itself! (e.g. page directory vs. page table)
- Two-level paging:
  - Example: 32 bit logical address, page size 4kB

| page directory $_{(10)}$	page table $_{(10)}$	offset$_{(12)}$

page table base register
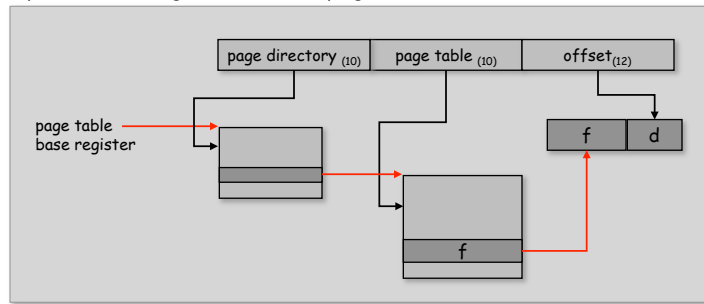
- Three-level paging (SPARC), four-level paging (68030), ...
- AMD64 (48-bit virtual addresses) has 4 levels.
- Even deeper for 64 bit address spaces (5 to 6 levels)

Variations: Inverted Page Table

proc id	page no	offset

	process id	page no
0		
1		
2		
3		
...		
n		

3	offset
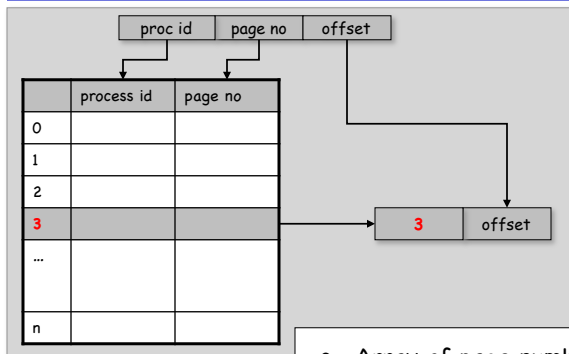
- Array of page numbers indexed by frame number.
  - page lookup: search for matching frame entry
- Size scales with physical memory.
- Single table for system (not per process)
- Used in early virt. memory systems, such as the Atlas computer.
- Not practical today. (Why?)

Variations: Hashed Page Table

page number	offset

hash function

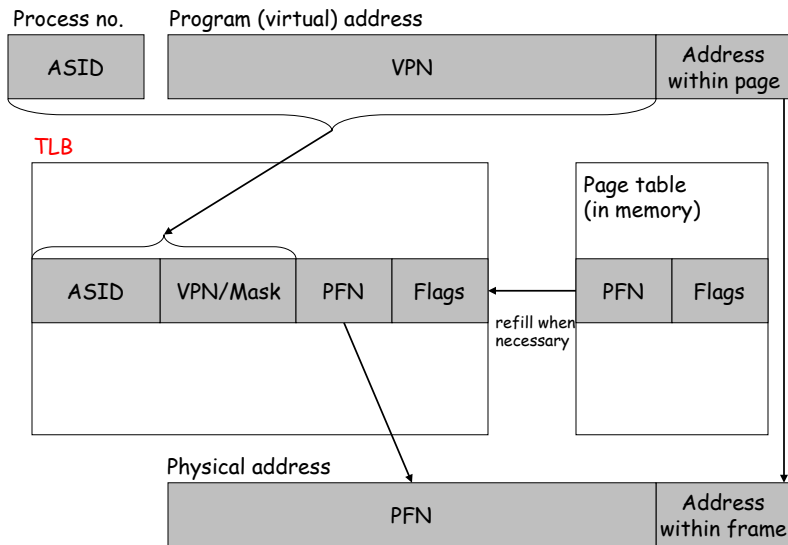| proc id	page no	frame no	chain
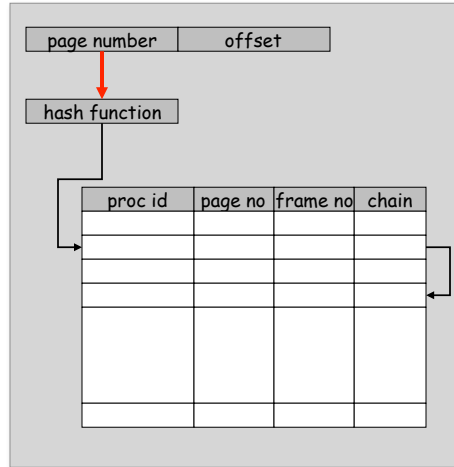
- Used by many 64bit architectures:
  - IBM POWER
  - HP PA-RISC
  - Itanium
- Scales with physical memory
- One table for whole system

cons:
- How about collisions?
- Difficult to share memory between processes

Software-loaded TLBs: Paging - MIPS Style

Process no.

Program (virtual) address

ASID	VPN	Address within page

TLB

ASID	VPN/Mask	PFN	Flags

Page table (in memory)

PFN	Flags

refill when necessary

Physical address

| PFN	Address within frame

Recap: Memory Translation -- "VAX style"

1. Split virtual address
2. Concatenate more-significant bits with Process ASID to form page address.
3. Look in the TLB to see if we find translation entry for page.
4. If YES, take high-order physical address bits.
   - (Extra bits stored with PFN control the access to frame.)
5. If NO, system must locate page entry in main-memory-resident page table, load it into TLB, and start again.

Memory Translation -- MIPS Style

- In principle: Do the same as VAX, but with as little hardware as possible.
- Apart from register with ASID, the MMU is just a TLB.
- The rest is all implemented in software!
- When TLB cannot translate an address, a special exception (TLB refill) is raised.
- Note: This is easy in principle, but tricky to do efficiently.

MIPS TLB Entry Fields

input

VPN	ASID	G

output

PFN	Flags N D V
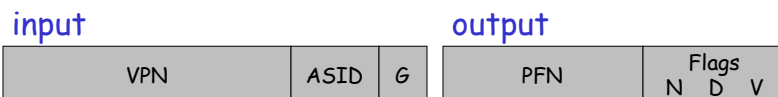
- **VPN**: higher order bits of virtual address
- **ASID**: identifies the address space
- **G**: if set, disables the matching with the ASID

- **PFN**: Physical frame number
- **N**: 0 – cacheable, 1 – noncacheable
- **D**: write-control bit (set to 1 if writeable)
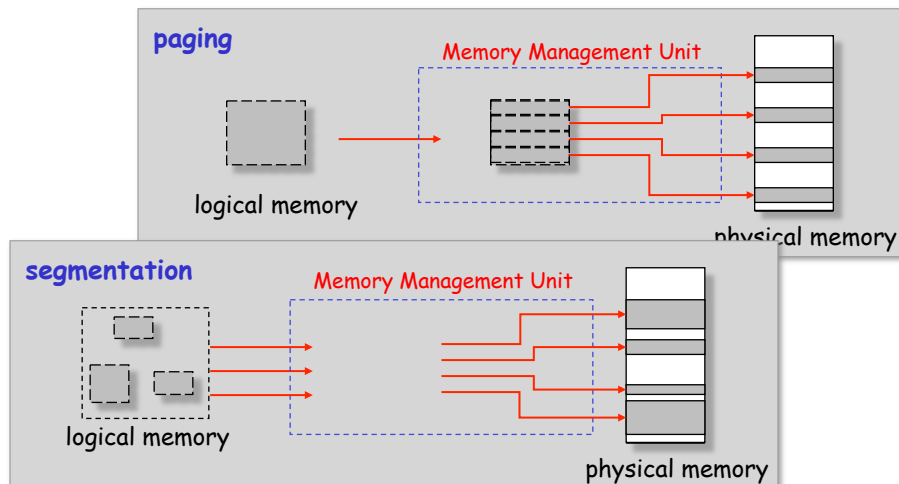- **V**: valid bit

MIPS Translation Process

1. CPU generates a program (virtual) address on a instruction fetch, a load, or a store.
2. The 12 low-end bits are separated off.
3. Case 1: TLB matches key:
   1. Matching entry is selected, and PFN is glued to low-order bits of the program address.
   2. **Valid?:** The V and D bits are checked. If problem, raise exception, and set BadVAddr register with offending program address.
   3. **Cached?:** IF C bit is set, the CPU looks in the cache for a copy of the physical location's data. If C bit is cleared, it neither looks in nor refills the cache.
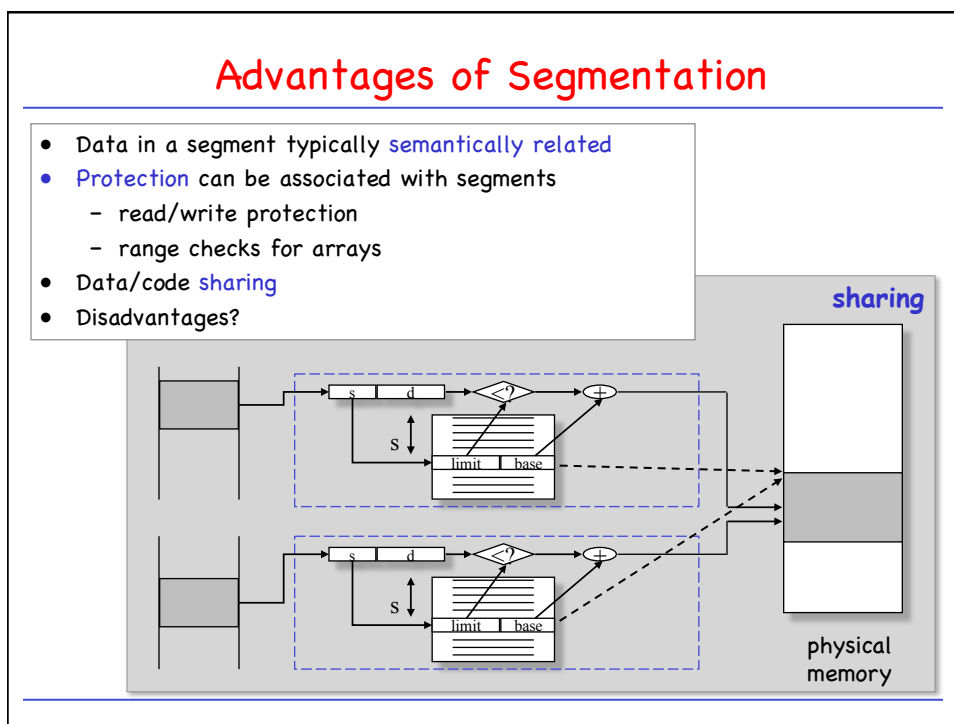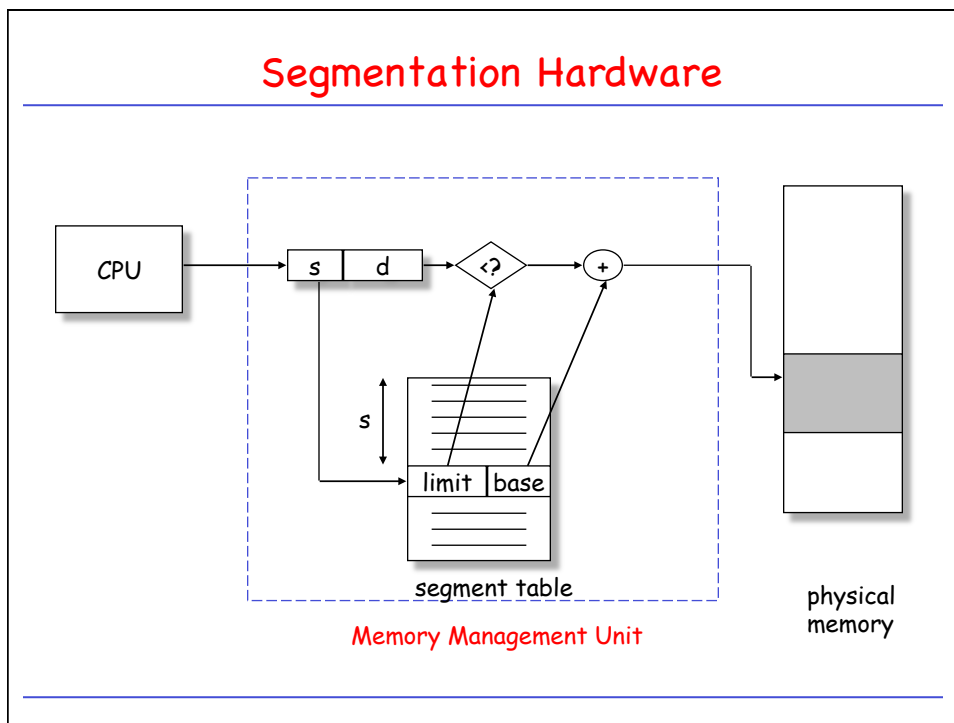4. Case 2: TLB does not match: TLB Refill Exception (see next page)

TLB Refill Exception

- Figure out if this was a correct translation. If not, trap to handling of address errors.

- If translation correct, construct TLB entry.

- If TLB already full, select an entry to discard.
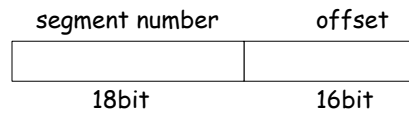
- Write the new entry into the TLB.

Segmentation

- Users think of memory in terms of segments (data, code, stack, objects, ...)
- Data within a segment typically has uniform access restrictions.

paging

Memory Management Unit

logical memory

physical memory

segmentation

Memory Management Unit

logical memory

physical memory

Segmentation Hardware

segment table

Memory Management Unit

physical memory

Advantages of Segmentation

- Data in a segment typically semantically related
- Protection can be associated with segments
  - read/write protection
  - range checks for arrays
- Data/code sharing
- Disadvantages?

sharing

physical memory

Solution: Paged Segmentation

- Example: MULTICS

segment number	offset
18bit	16bit

Problem: 64kW segments -> external fragmentation!

Solution: **Page the segments**.

segment number	page#	page offset
18bit	6bit	10bit
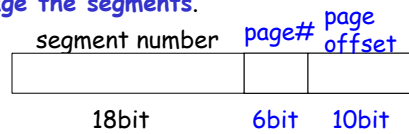
Problem: need 2^18 segment entries in segment table!

Solution: **Page the segment table**.

page#	page offset	page#	page offset
8bit	10bit	6bit	10bit