

Memory Management

Classifications of information stored in memory:

1. Role in programming language: instructions (specify op code and operands), variables (information that changes as program runs), constants (information that never changes).
2. Changeability: read-only (code, constants), read and write (variables).
Why is this important?
3. Addresses vs. data (e.g., A vs. $A[0]$).
Why is this important?

When is its space allocated (binding time)?

Static: compile time, link time, load time.

Unpredictability: how much memory? (recursive procedures, number of processes)

Dynamic: Generate the physical address dynamically during every reference.

Two views of address space (physical and logical)

Two basic operations in dynamic storage management:

allocate

free

Two organizations:

stack (push, pop), simple structure efficient implementation.

heap (free list, bit map, garbage collection)
(see Knuth volume 1).

Division of a process' memory:

When a process is running, its memory is divided up into areas called *segments*. In UNIX, each process has three segments: code, data, stack.

- Why distinguish between different segments of memory? Separate read-only code from read-write data.
- What if two processes?
- Where does OS go?

Division of responsibility between various portions of system:

- Compiler: generates object file. Information in an object file is incomplete, since one file may reference some things defined in another.
- Linker: combines object files into a complete and self-sufficient object file.
- Operating system: loads object files in the secondary storage into memory, allows processes to share memory, provides facilities for processes to get memory after they've started running.
- Run-time library: together with OS, provides dynamic allocation routines (e.g., *calloc* and *free*).

Sharing Memory

Recall: Where does OS go? What if two processes?

In a uniprogramming system:

Highest memory holds OS.

Process is allocated memory starting at 0, up to the OS area.

When loading a process, just bring it in at 0.

In a multiprogramming system:

Goals: transparency (processes are not aware of the fact that the memory is shared), safety (processes mustn't be able to corrupt each other), efficiency (CPU and memory shouldn't be degraded badly by sharing).

Issues:

How to divide up the memory into regions?

How to allocate regions among processes?

How to protect each user's processes?

Assumption: A process is allocated in contiguous regions (one segment for each process).

Fixed size with fixed boundaries

Division:

The memory is divided into regions of fixed size with fixed boundaries.

Allocation:

Each region contains exactly one process.

Protection:

Static relocation (fixed boundaries)

Relocation register (base register).

Example

Load A[1] into \$16

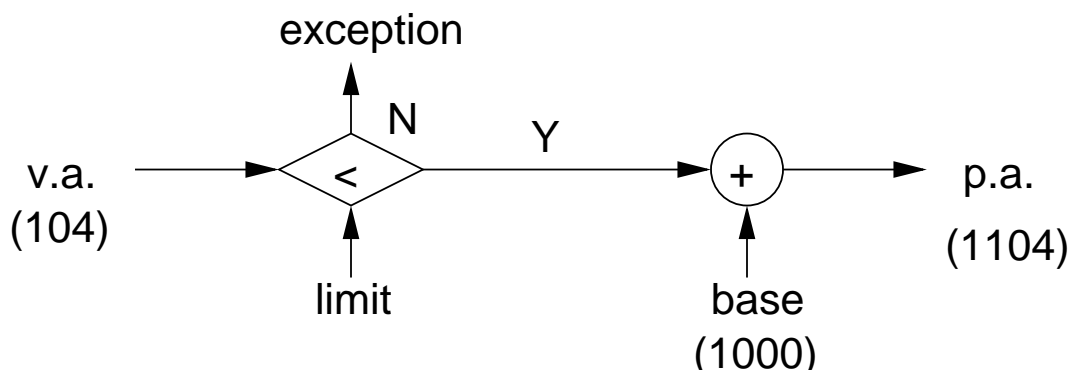
The address of A (100) is in \$4

```
lw $16, 4($4)
```

Each process is associated with the base address of the region allocated to it.

Hardware support

When a process is switched in, the base address is loaded into the relocation register.



Variable size regions

Division:

Memory is divided into variable size regions according to processes.

Allocation:

best-fit, worst-fit, first-fit.

Protection:

Boundary registers or base register + limit.

Problem:

processes cannot share codes.

External fragmentation v.s. internal fragmentation.

Can we scatter the regions of a process in the memory?

Why is this necessary? Processes can share segments.

Problems must be solved:

- generating addresses
- protecting users

Two approaches: paging, segmentation.

Paging

Division:

The memory is divided into fixed size (512-8K) regions (pages).

Allocation:

The system keeps a list of free pages (e.g., bit map).

Generating addresses

logical address: (page number, displacement)

page map table (PMT):

page number \rightarrow base address

physical address \leftarrow base $+$ displacement

Protection:

Every translation goes through the PMT of the current process.

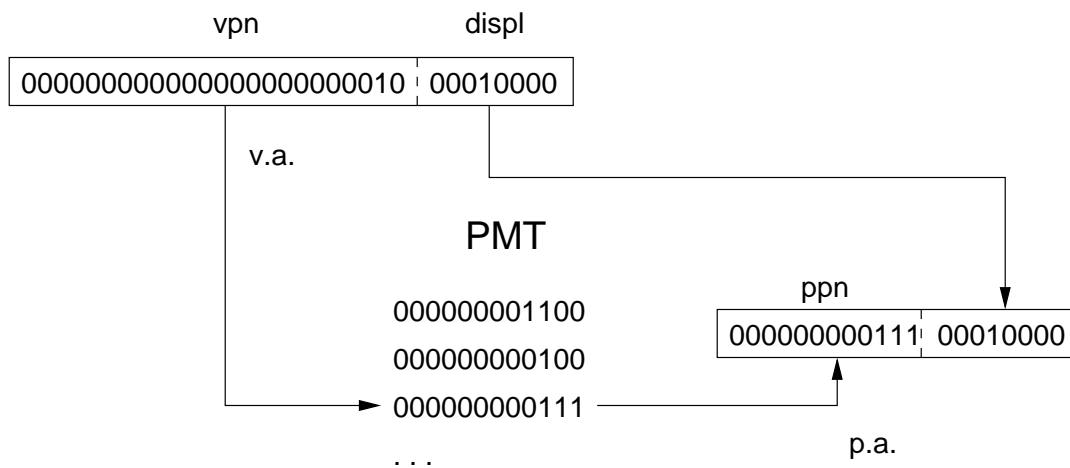
It is confined to one process.

Load program one page at a time.
Establish a PMT.
Keep the pointer to PMT in PCB.

Example

Virtual space: 4G, 32 bits
Memory: 1M, 20 bits
Page size: 256, 8 bits

Hardware support for paging.



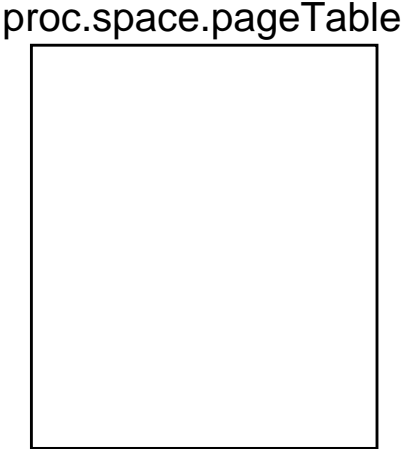
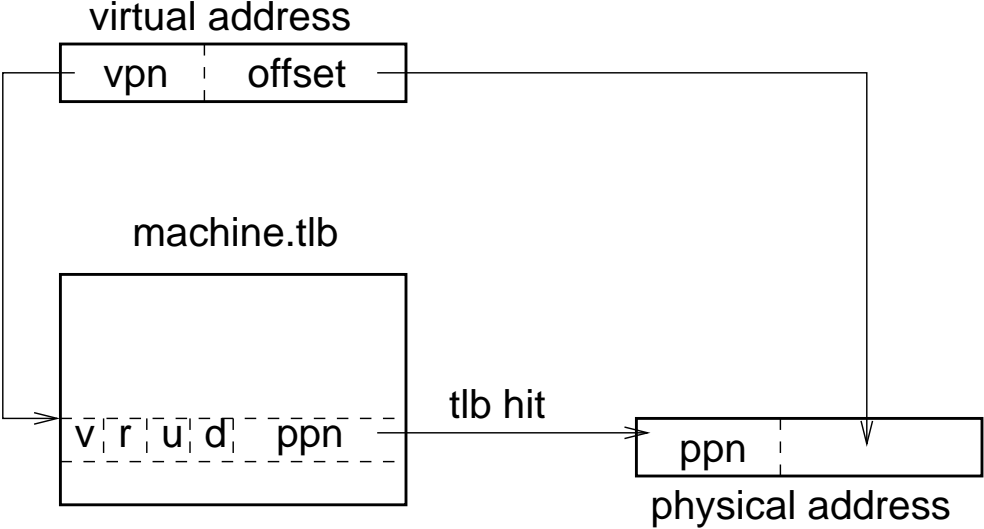
Paging eliminates external fragmentation.
Internal fragmentation exists.
Easy to make allocation and swapping.

Where do we keep the PMT?
Main memory (slow)

Keep part of PMT in fast memory (cache):
TLB (translation Lookaside buffer).

With TLB, all CPU sees is TLB.
When a TLB miss occurs, exception handler is called.

Translation look-aside buffer (TLB)



Segmentation

Division:

Memory is divided into to variable size regions (segments) according to programmer's view.

Allocation:

System keeps a list of holes in the memory.

Generating addresses:

logical address (segment number, offset)

segment table: segment number \rightarrow base, limit

physical address: $\text{base} + \text{offset}$ (if \leq limit)

Protection: Similar to paging.

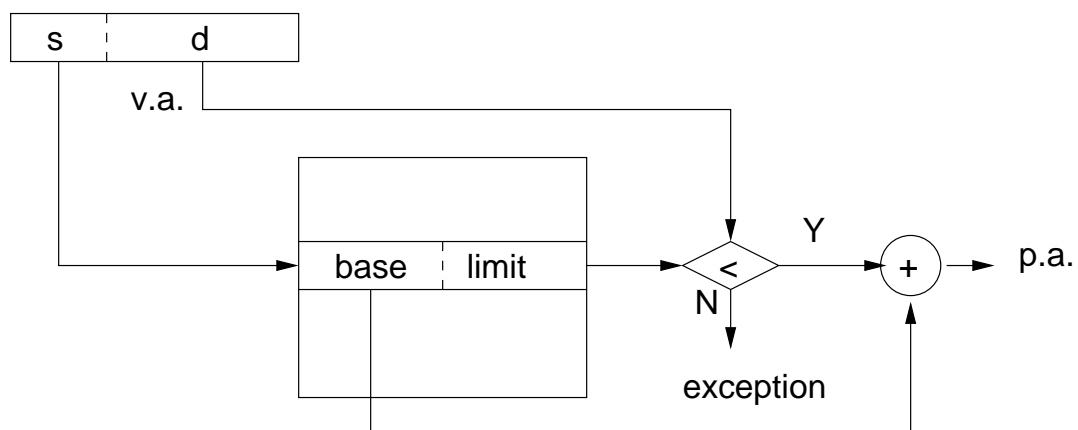
In addition, segmentation easily provides access restrictions on segments. (Read only for code segment.)

Load program one segment (code/data) at a time.

Establish a segment table. Each entry contains (base, limit).

Keep the pointer to the segment table in PCB.

Hardware support for segmentation:



Virtual Memory

Why should we load all pages of a process in the main memory? (some are never used some are rarely used, 90/10 rule)

Goal: create the illusion of a disk as fast as main memory.

Issues to be discussed:

1. When is a page brought in memory? (demand paging)
2. How do we know whether a page is in memory? (valid-bit)
3. Why should we always rewrite a page when it has to be replaced? (dirty-bit)
4. How do we replace a page in memory when it is necessary?

Replacement algorithms

FIFO, LIFO, LFU, LRU

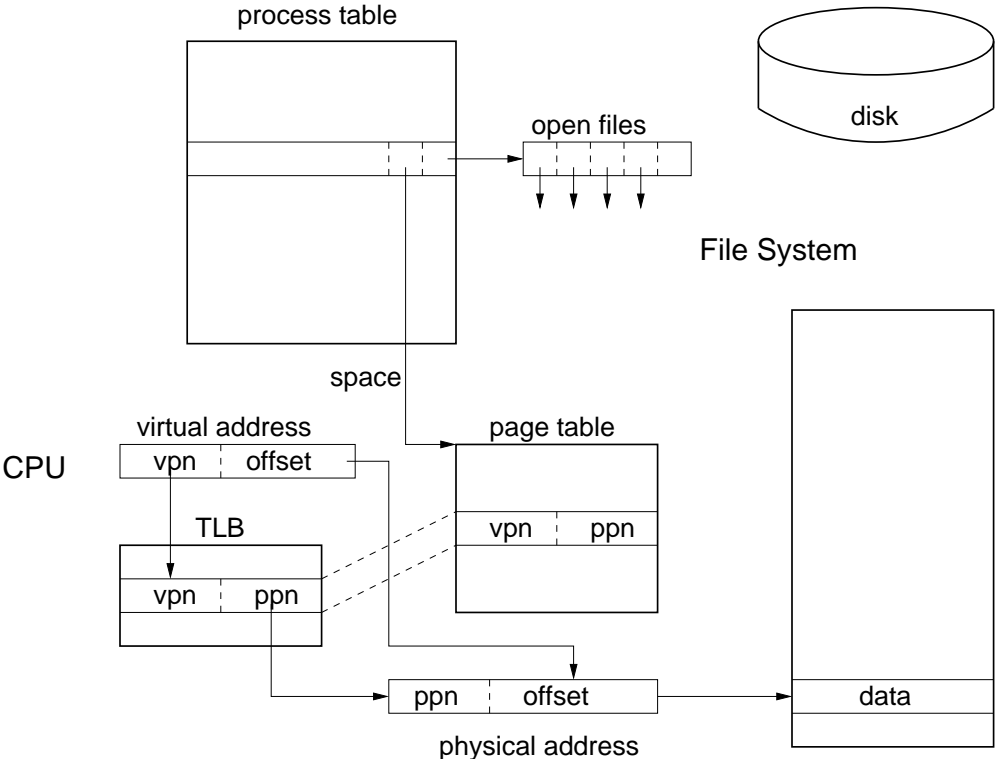
Approximation of LRU (clock algorithm):

1. when reference a page, mark the use (reference) bit.
2. when replacing a page, sweep the clock hand. If the use bit is marked, reset it to unmarked and continue until find an unmarked use bit. (Second chance.)

Need an inverted page table. (Note. Processes may share a page.)

How many pages should be kept in memory?
Too many jobs, memory is overcommitted.
(What do humans do?)

Putting process and memory together.



Thrashing and working set model

Thrashing: a process is spending more time paging than executing. Memory is as slow as disk.

Working set model

- basis: locality
- working set window (WSW) (a time frame)
- working set (WS) (a set of pages referenced in the time frame)
- working set size (WSS) (number of pages in WS)

Page replacement can be determined by working set model.

Working set model can prevent thrashing.

The collection of active processes is called the balance set.

Working set + balance set can prevent thrashing.

- Keep the sum of working sets of all runnable processes less than memory size.
- Divide runnable processes up into two groups: active and inactive.
- Keep the balance set up to date.

Examples

System 370: paged segmentation

virtual address space: 24 bits

segment no: 4 bits

page no: 8 bits

offset: 12 bits

physical address space: 24 bits

segment table entry:

page table address (real): 24

page table size (number of pages)

protection (R, RW, 0)

page table entry:

page address (real): 12 bits → 2 bytes

Note: byte addressable

Example. (all numbers in hexadecimal)

segment table:

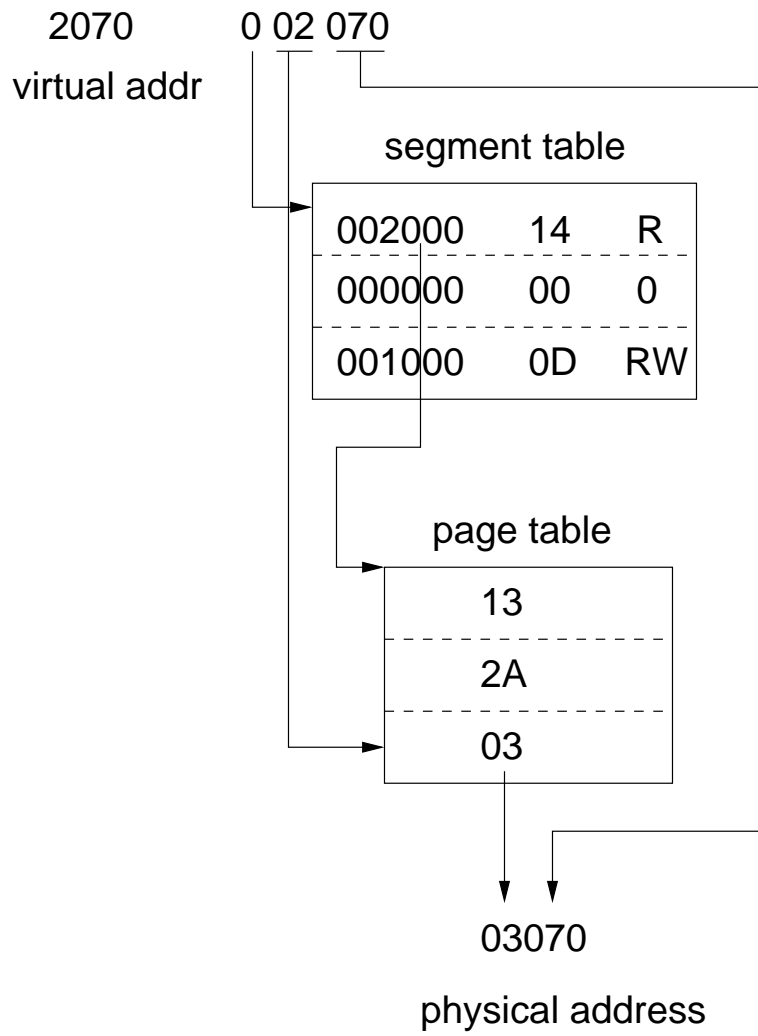
002000	14	R
000000	00	0
001000	0D	RW

At location 2000: 13, 2A, 3 (each value is 2 bytes long)

Translate the following addresses from virtual to physical:

2070 read (3070)

210014 write (bounds violation)



VAX-11/780: Paged Virtual Memory

address space: 32 bits (4G)

3G-4G: unused

2G-3G: system segment, bit 31=1

1G-2G: process segment P1, bit 30=1

0G-1G: process segment P0, bit 30=0

page size: 512 bytes (small)

To save page table space

two level paging (recursive):

system page table (physical memory)

process page table (system segment)

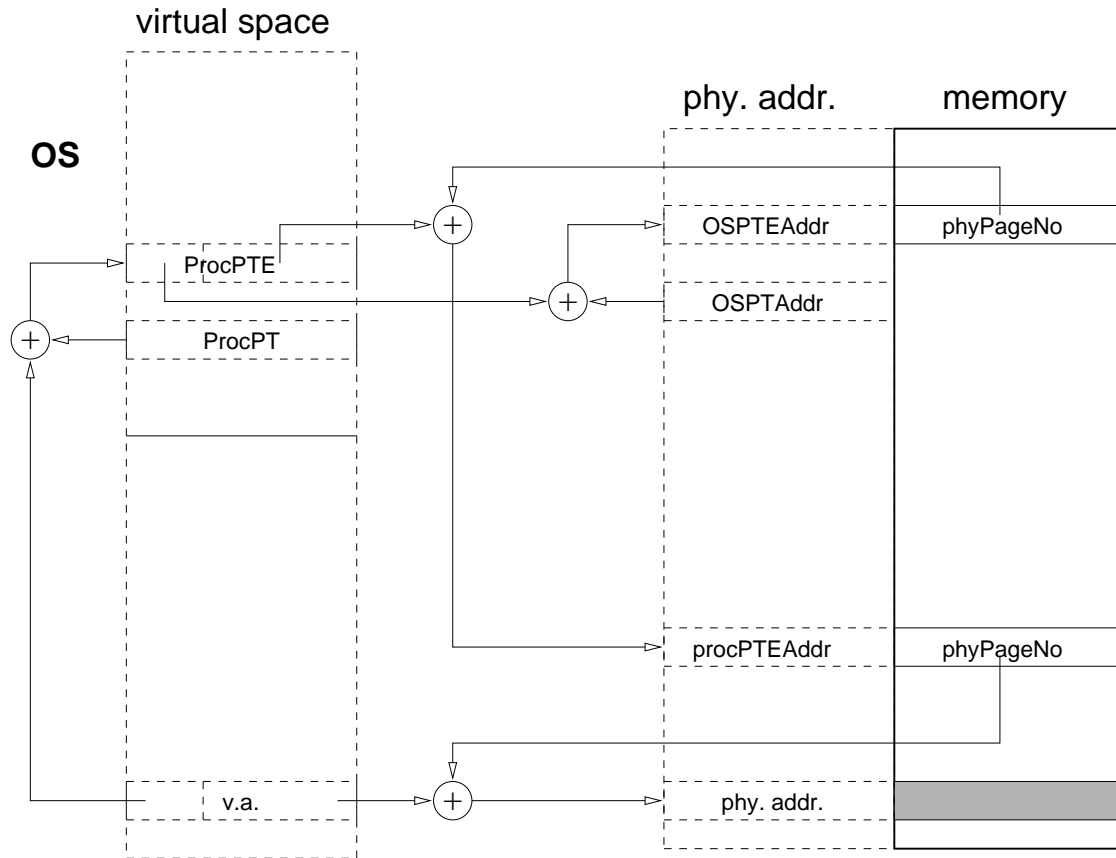
PTE includes:

M—modify bit

V—valid bit

PROT—four protection bits

TLB: two-way-set-associative



Two Level Paging

Some parameters:

Hit time	1 clock cycle
Miss penalty	22 clock cycles
Miss rate	1% - 2%
Cache size	128 PTEs