Memory Management

Sanzheng Qiao

Department of Computing and Software

January, 2013

Classifications of information

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?

Allocation

When is its space allocated (binding time)?

Static: compile time, link time, load time.
Unpredictability: how much memory? (recursive procedures, number of processes)

Allocation

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 and free
Two organizations:
stack (push, pop), simple structure efficient implementation.
heap (free list, bit map, garbage collection) (see Knuth volume 1).

Memory division

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.

Memory division

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

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*, *free*, *new*).

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?

Fixed size regions

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

Fixed size regions

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.

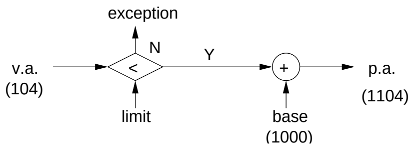
Protection: Static relocation (fixed boundaries)
Relocation register (base register).

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.

Memory allocation

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

Memory allocation

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).

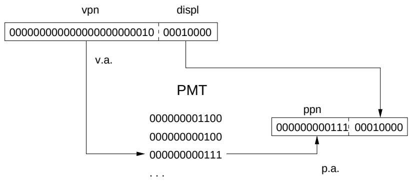
Allocation:
The system keeps a list of free pages (e.g., bit map).

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.

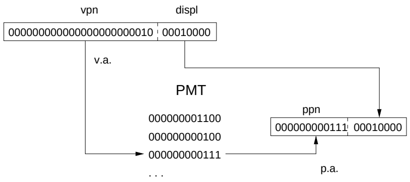
Paging

Example
Virtual space: 4G, 32 bits
Memory: 1M, 20 bits
Page size: 256, 8 bits
Hardware support for paging.



Paging

Example
Virtual space: 4G, 32 bits
Memory: 1M, 20 bits
Page size: 256, 8 bits
Hardware support for paging.



During a context switch, change the current PMT to the PMT of the process being switched in.

Example: Nachos 4.02

Pure paging.

Page size 128 (machine/machine.h)

PMT `pageTable` entry `TranslationEntry` structure (machine/translate.h)

- virtual page number
- physical page number
- valid: Is the translation valid?
- use: Set every time the page is referenced or modified
- dirty: Set every time the page is modified

Example: Nachos 4.02

Address space (userprog/AddrSpace)

- An array of translation entries (PMT)
- Number of pages

In the thread structure

if the address space is null, it is a thread in the kernel space

if the address space is not null, it is a (Nachos) process, associated with a user program

Example: Nachos 4.02

Loading a user program from (Nachos) disk into (Nachos) memory, by calling the file system.

Two views:

- Program: A set of (virtual pages, virtual addresses)
- File system: A file (Nachos object file, infile addresses)

Example: Nachos 4.02

Nachos object file file-header (userprog/noff.h)

- Nachos object file magic number
- Code segment
  - virtual address, infile address, size
- Initial data segment
  - virtual address, infile address, size
- Uninitialized data segment
  - virtual address, infile address, size

Total address space size:
code size + init data size + uninit data size + stack size

Example: Nachos 4.02

Loading a user program from (Nachos) disk into (Nachos) memory, by calling the file system.

userprog/addrspace.cc

Load

1. Read the file header
2. Check the magic number
3. Calculate address space size (number of pages)
4. Copy in code and initial data segments

Assuming linear mapping (uniprogramming),
virtual page number $=$ physical page number

Paging

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

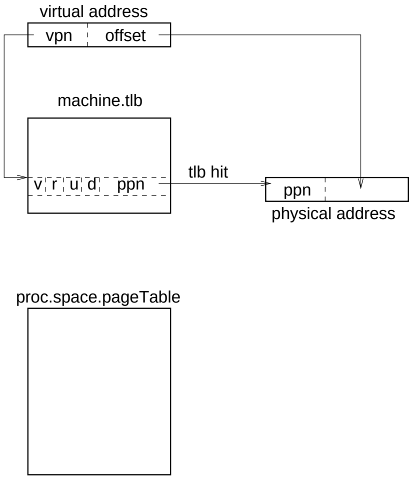
Where do we keep the PMT?

Main memory (slow)

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

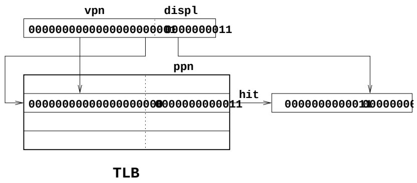
With TLB, all CPU sees is TLB.
During a context switch, set all TLB entries invalid.

Translation look-aside buffer (TLB)

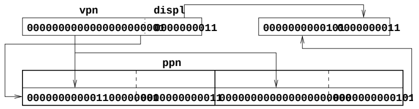


TLB organization

One-way-set-associative



Two-way-set-associative



Segmentation

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

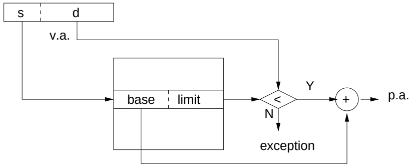
Allocation:
System keeps a list of holes in the memory.

Generating addresses:
logical address (segment number, offset)
segment table: segment number $\rightarrow$ base, limit
physical address: base $+$ offset (if $\leq$ limit)

Protection: Similar to paging.
In addition, segmentation easily provides access restrictions on segments. (Read only for code segment.)

Segmentation

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:



Demand paging and 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?

Demand paging

Bring a page into the memory when it is referenced.

Initially, set all PMT entries invalid.

When a page is not in the memory, raise a page fault exception.

Demand paging

Page fault exception handler

1. Get the bad virtual address that caused the page fault
2. Allocate a physical page
3. Call the file system to copy the page from the disk to the physical page in the memory. Note that a page may contain data from more than one segment.
4. Update the PMT
5. Re-execute the instruction

Demand paging

Page fault exception handler

1. Get the bad virtual address that caused the page fault
2. Allocate a physical page
3. Call the file system to copy the page from the disk to the physical page in the memory. Note that a page may contain data from more than one segment.
4. Update the PMT
5. Re-execute the instruction

One instruction may cause more than one page fault.

TLB miss

TLB miss handler

```
1. Get the bad virtual address that caused the TLB miss
2. if the PMT entry is valid
       Copy the PMT entry to the TLB
       Re-execute the instruction
   else
       Call page fault exception handler
```

Replacement algorithms

FIFO, LIFO, LFU, LRU

Replacement algorithms

FIFO, LIFO, LFU, LRU

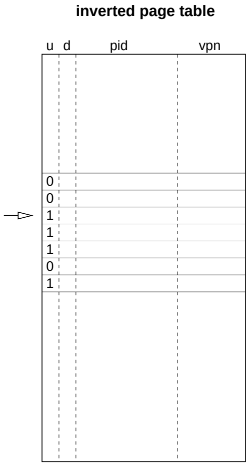
Approximation of LRU (clock algorithm):

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.)

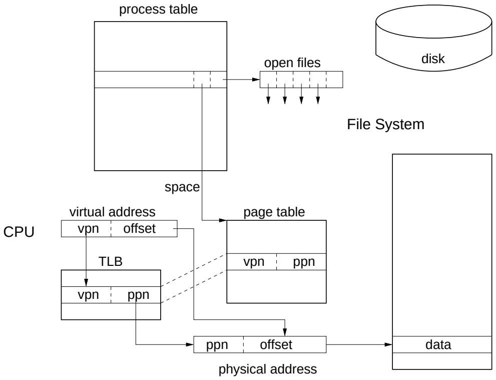
Need an inverted page table. (Note. Processes may share a page.)
How many pages should be kept in memory?
Too many jobs, memory is overcommitted. (What do humans do?)

Inverted page table

The operating system has a global page table mapping physical pages to virtual pages. Each entry is a pair (pid, vpn).



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.

Working set model

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 $\rightarrow$ 2 bytes
- Note: byte addressable

Examples

All numbers in hexadecimal

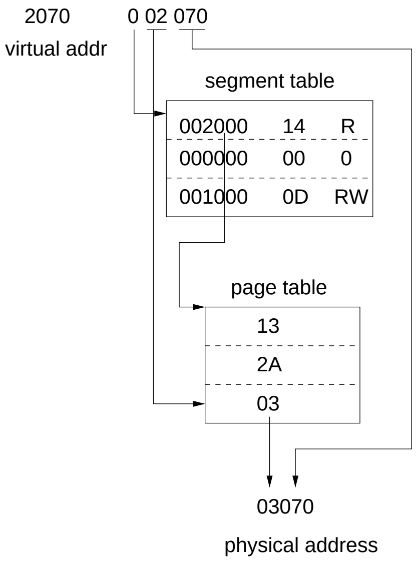
segment table:

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)

Examples



2070
virtual addr
0 02 070
segment table
002000 14 R
000000 00 0
001000 0D RW
page table
13
2A
03
03070
physical address

Examples

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)

Examples

To save page table space

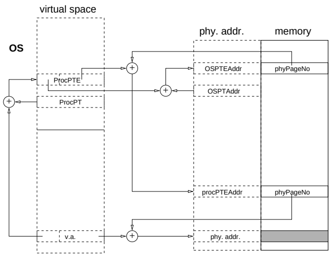
    two level paging (recursive):
    system page table (physical memory)
    process page table (system segment)

PTE includes:

    M–modify bit
    V–valid bit
    PROT–four protection bits

TLB: two-way-set-associative

Examples



Two Level Paging

Some parameters:

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