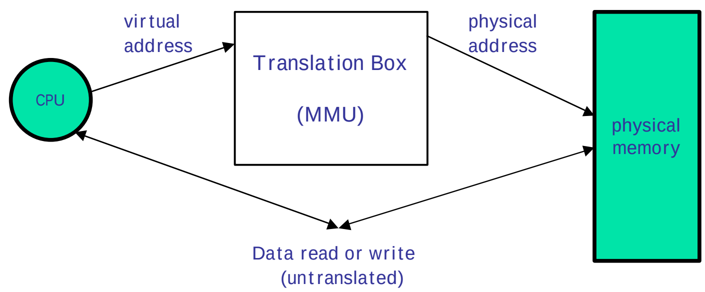
Address Translation

Arvind Krishnamurthy
Spring 2004

Address Translation Recap

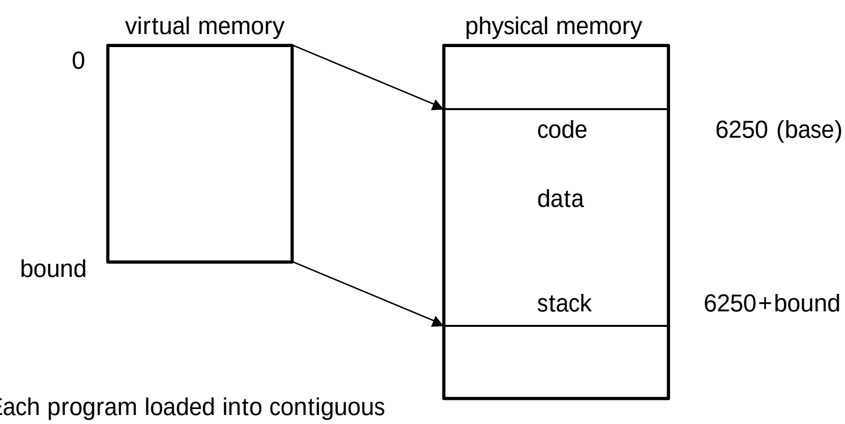
- Goal: memory protection
- Translate every memory reference to the actual physical address
- Programmable: relies on a memory address translation table
- On process switch, switch the translation table
- Install translations and let the program run
  - Who installs translations? Software
  - Not user level software → need to distinguish between user and kernel code → need for protected kernel mode
  - Hardware support for kernel mode: bit in a “processor status word”
  - When set, allows all kinds of protected operations
  - In kernel mode, all memory references are physical addresses

Memory Protection



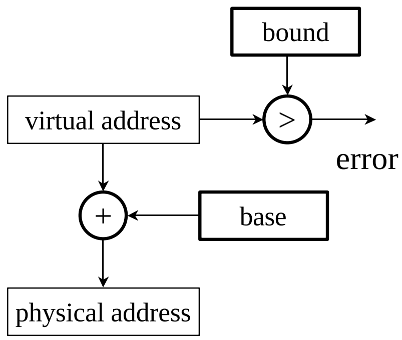
- This lecture: how to implement the translation? Should be fast
- Options
  - Base and Bounds
  - Segmentation
  - Paging
  - Multilevel translation

Base and Bounds



Each program loaded into contiguous regions of physical memory.

Base and Bounds (contd.)



- Built in Cray-1
- Hardware cost: two registers, adder, comparator ➔ fast
- On a context switch: save/restore base, bound registers
- What are the pros/cons of this approach?

Segmentation

- Motivation
  - separate the virtual address space into several segments so that we can share some of them if necessary
  - also allow holes in the address space
- A segment is a region of logically contiguous memory
- Main idea: generalize base and bounds by allowing a table of base&bound pairs

(assume 2 bit segment ID, 12 bit segment offset)

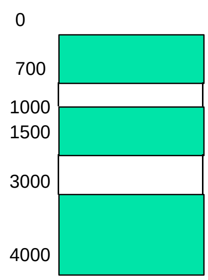
virtual segment #		physical segment start	segment size
code	(00)	0x4000	0x700
data	(01)	0x0000	0x500
-	(10)	0	0
stack	(11)	0x2000	0x1000

Segmentation example

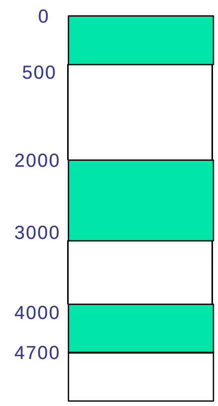
(assume 2 bit segment ID, 12 bit segment offset)

v-segment #	p-segment start	segment size
code (00)	0x4000	0x700
data (01)	0x0000	0x500
- (10)	0	0
stack (11)	0x2000	0x1000

virtual memory



physical memory



Segmentation example (cont'd)

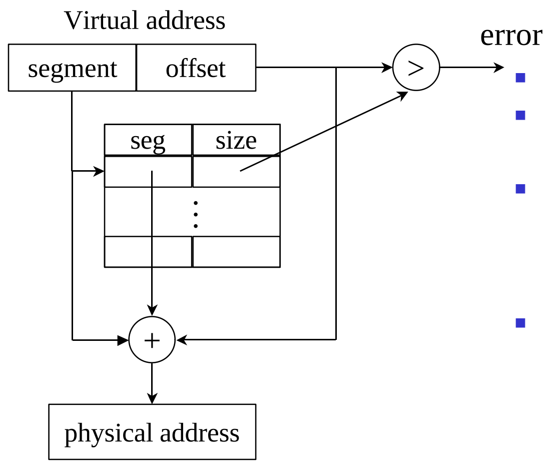
Virtual memory for strlen(x)

```
Main:   240      store 1108, r2
        244      store pc+8, r31
        248      jump 360
        24c
        ...
strlen: 360      loadbyte (r2), r3
        ...
        420      jump (r31)
        ...

   x:   1108     a b c \0
        ...
```

physical memory for strlen(x)

```
x:      108      a b c \0
        ...

Main:   4240     store 1108, r2
        4244     store pc+8, r31
        4248     jump 360
        424c
        ...
strlen: 4360     loadbyte (r2), r3
        ...
        4420     jump (r31)
        ...
```

Segmentation Implementation



- Have a table of (seg, size)
- Protection: each entry has
  - (nil,read,write)
- On a context switch: save/restore the table or a pointer to the table in kernel memory
- Have we addressed all of the requirements with this approach?

Object file format

- Notice:
  - Segmentation table performs the task of runtime relocation
  - Loader's task is simple; linker still needs to perform static relocation
- Standard file format: ELF, COFF

  type "man a.out" to see detail
  - magic number
  - the header information
  - a list of segments:

    (a) size needed for BSS segment (uninitialized variables)

    (b) data segment (with initialized global and static variables)

    (c) text segment (including executable instructions)
  - optional relocation information
  - optional symbol table and line number information

Object file format (cont'd)

```c
char chArray[40];
static double x;
int y = 13;
static long z = 2001;

main () {
    int i = 3, j, *ip;

    ip = malloc(sizeof(i));
    chArray[5] = i;
    y = 2.0 * z;
}
```

Runtime segments:

BSS segment: chArray, x
data segment: y, z
code segment: all the machine instructions
stack segment: local variables
heap segment: dynamic memory allocation

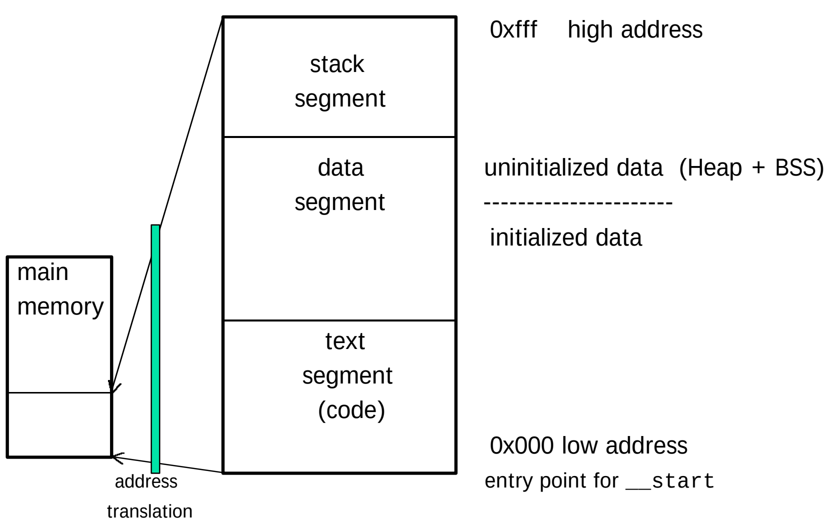
The NOFF object file contains:

code segment
data segment with initial values (initData)
BSS segment with size only (uninitData)

0 -----------------------> increasing offset

Header	text (code)	initialized data

Address space in Nachos

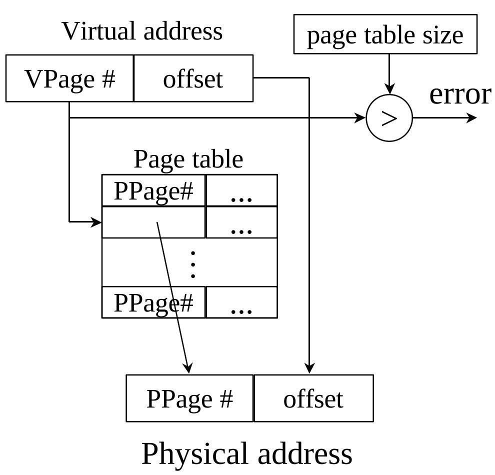


Paging

- Motivations
  - both branch & bounds and segmentation still require fancy memory management (e.g., first fit, best fit, re-shuffling to coalesce free fragments if no single free space is big enough for a new segment)
  - can we find something simple and easy

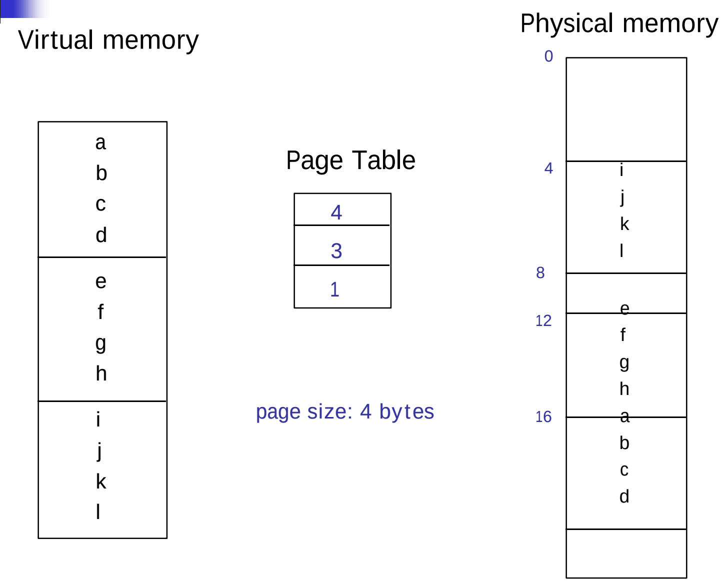
- Solution
  - allocate physical memory in terms of fixed size chunks of memory, or **pages**.
  - Simpler because it allows use of a bitmap: 00111110000001100
    - each bit represents one page of physical memory
    - 1 means allocated, 0 means unallocated

Paging (contd.)



- Use a page table to translate
- Context switch: similar to the segmentation scheme
- Question: What should be the page size?
- What are the pros/cons of this scheme?

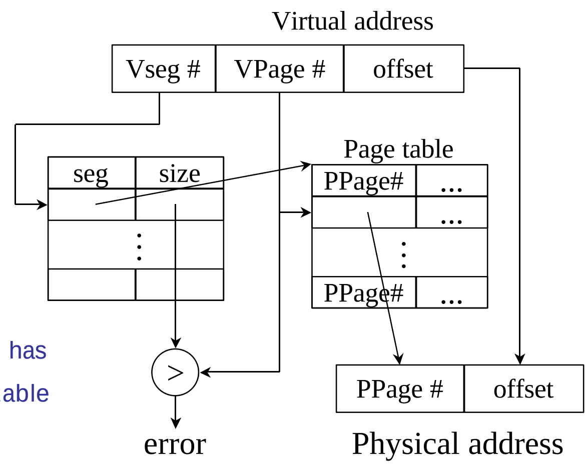
Paging example



How many PTEs do we need ?

- Worst case for 32-bit address machine
  - # of processes $\times$ 2^{20} (if page size is 4096 = 2^{12} bytes)
- What about 64-bit address machine?
  - # of processes $\times$ 2^{52}
- Question: how do we solve the huge page-table size problem?

Segmentation with paging



Each segment has its own page table

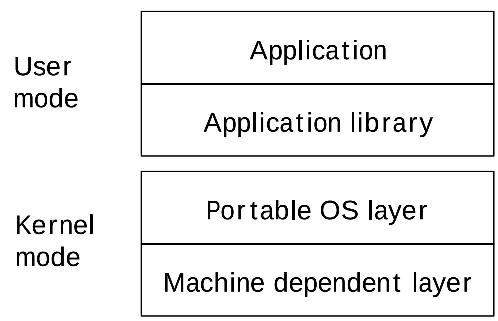
Paged Page Tables

- So far, page tables have to be allocated linearly in memory
- Can we page them?
  - That is, can we replace page table pointers with virtual addresses
  - Implication: they can be swapped
- Put page tables in a special segment that is translated but not accessible to user programs (part of program's virtual address space)
- Page table for this segment alone is in physical memory
- Segment table contains page table pointers that are virtual for some segments, but physical for some others (used in MIPS and HPs)

Assignment 2: Overview

- Objectives
  - understand how system call really works
  - understand how to support multiple address spaces
- Problems
  - implement a set of system calls
    - Exec, join on processes
    - Create, open, read, write, close on files
    - Fork, yield for threads (optional – extra credit)
  - implement multiprogramming
    - use bitmap to find unused main memory
    - setup the page table (translation is no longer identity)
    - data copying between user and kernel
  - support argument passing for “exec”
    - support exec of “prog arg1 arg2” instead of exec prog
    - should be easy

Traditional OS Structure



- How does Nachos’s structure fit into this model?
  - Nachos is the portable OS layer – it simulates the hardware and machine-dependent layer, and it simulates the execution of user programs running on top
    - Can still use debugger, printf, etc.
    - Can run normal UNIX programs concurrently with Nachos
  - Could run Nachos on real hardware by writing a machine-dependent layer

Assignment 2: Overview (cont'd)

- Nachos execution overview:
  - user program (written in C): `halt.c`
  - gcc cross compiler compiling halt.c into MIPS binary code

```
decstation-ultrix/bin/gcc halt.c start.s -o halt.coff
coff2noff halt.coff  halt
```

Here, `halt.coff` is like the standard "a.out" file;
"`halt`" is a simplified version of "`halt.coff`" designed for Nachos

  - nachos loads and runs the user code (**`exec`** or `progtest.cc`)
    - initializing an address space
    - set up the page table (mapping address space to physical memory)
    - zero-ing all memory cells
    - copy all segments in "noff" file (e.g., `halt`) into main memory
    - call the MIPS simulator to run the user code

Assignment 2: user C program `halt.c`

```
#include "syscall.h"

int
main()
{
    Halt();
    /* not reached */
}
```

- Note: we don't use any standard C libraries (because they wouldn't work with the Nachos kernel)

Assignment 2: user C program `shell.c`

```
#include "syscall.h"

int main()
{
    SpaceId newProc;
    OpenFileId input = ConsoleInput;
    OpenFileId output = ConsoleOutput;
    char prompt[2], ch, buffer[60];
    int i;

    prompt[0] = '-';
    prompt[1] = '-';
    ......
```

```
......
while( 1 )
  {  Write(prompt, 2, output);

     i = 0;
     do {
        Read(&buffer[i], 1, input);
     } while( buffer[i++] != '\n' );

     buffer[--i] = '\0';

     if( i > 0 ) {
           newProc = Exec(buffer);
           Join(newProc);
     }
  }
}
```

The assembly stub file: `start.s`

```
#include "syscall.h"
        .text
        .align  2
/* ---------- a stub to main() ------- */
      .globl __start
      .ent     __start
__start:          /* must start at address 0 */
      jal      main
      move   $4,$0
      jal      Exit
                  /* if we return from main, exit(0) */
      .end __start
```

```
/* ------- System call stub for Halt ----- */
      .globl Halt
      .ent     Halt
Halt:
      addiu $2,$0,SC_Halt
      syscall
      j        $31
      .end Halt
/* ------- System call stub for Exit ------ */
      .globl Exit
      .ent     Exit
Exit:
      addiu $2,$0,SC_Exit
      syscall
      j        $31
      .end Exit
..............................
```

```
gcc halt.c start.s -o halt.coff
```