CS 61C: Great Ideas in Computer Architecture (Machine Structures) Memory Management

Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/fa10

Review

- C has three pools of data memory (+ code memory)
  - Static storage: global variable storage, basically permanent, entire program run
  - The Stack: local variable storage, parameters, return address
  - *The Heap (dynamic storage): `malloc()/calloc()` grabs space from here, `free()/cfree()` returns it, `realloc()` resizes space*
- Common (Dynamic) Memory Problems
  - Using uninitialized values
  - Accessing memory beyond your allocated region
  - Improper use of free/realloc by messing with the pointer handle returned by malloc/calloc
  - Memory leaks: mismatched malloc/free pairs

Agenda

- Review
- More Peer Review on Malloc/Free Memory Managment
- Administrivia
- Memory Protection in Hardware
- Virtual Memory
- Translation Lookaside Buffer
- Paging to/from Disk (if we get that far)
- Another View of Virtual Memory (
- Summary

What is wrong with this code?

- What is wrong with this code?

```
int* init_array(int *ptr, int new_size) {
  ptr = realloc(ptr, new_size*sizeof(int));
  memset(ptr, 0, new_size*sizeof(int));
  return ptr;
}

int* fill_fibonacci(int *fib, int size) {
  int i;
  init_array(fib, size);
  /* fib[0] = 0; */ fib[1] = 1;
  for (i=2; i<size; i++)
   fib[i] = fib[i-1] + fib[i-2];
  return fib;
}
```

Using Memory You Don't Own

- Improper matched usage of mem handles

```
int* init_array(int *ptr, int new_size) {
  ptr = realloc(ptr, new_size*sizeof(int));
  memset(ptr, 0, new_size*sizeof(int));
  return ptr;
}

int* fill_fibonacci(int *fib, int size) {
  int i;
  /* oops, forgot: fib = */ init_array(fib, size);
  /* fib[0] = 0; */ fib[1] = 1;
  for (i=2; i<size; i++)
   fib[i] = fib[i-1] + fib[i-2];
  return fib;
}
```

Remember: `realloc` may move entire block

What if array is moved to new location?

What is wrong with this code?

- What is wrong with this code?

```
void StringManipulate() {
  const char *name = "Safety Critical";
  char *str = malloc(10);
  strncpy(str, name, 10);
  str[10] = '\0';
  printf("%s\n", str);
}
```


Using Memory You Haven't Allocated

- Reference beyond array bounds

```
void StringManipulate() {
  const char *name = "Safety Critical";
  char *str = malloc(10);
  strncpy(str, name, 10);
  str[10] = '\0';
  /* Write Beyond Array Bounds */
  printf("%s\n", str);
  /* Read Beyond Array Bounds */
}
```


What is wrong with this code?

- What is wrong with this code?

```
int *pi;
void foo() {
  pi = malloc(8*sizeof(int));
  …
  free(pi);
}

void main() {
  pi = malloc(4*sizeof(int));
  foo();
}
```

Faulty Heap Management

- Memory leak: *more mallocs than frees*

```
int *pi;
void foo() {
  pi = malloc(8*sizeof(int));
  /* Allocate memory for pi */
  /* Oops, leaked the old memory pointed to by pi */
  …
  free(pi); /* foo() is done with pi, so free it */
}

void main() {
  pi = malloc(4*sizeof(int));
  foo(); /* Memory leak: foo leaks it */
}
```


What is wrong with this code?

- What is wrong with this code?

```
int *plk = NULL;
void genPLK() {
  plk = malloc(2 * sizeof(int));
  … … …
  plk++;
}
```


Faulty Heap Management

- Potential memory leak – handle has been changed, do you still have copy of it that can correctly be used in a later free?

```
int *plk = NULL;
void genPLK() {
  plk = malloc(2 * sizeof(int));
  … … …
  plk++; /* Potential leak: pointer variable
  incremented past beginning of block! */
}
```

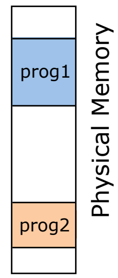

Program Model So Far

- Only 1 program running on the computer
  - The addresses in the program are exactly the physical memory addresses
- How can we prevent the program from accidentally clobbering itself?
- How can we run multiple programs with accidentally stepping on same addresses?
- How can we protect programs from clobbering each other?

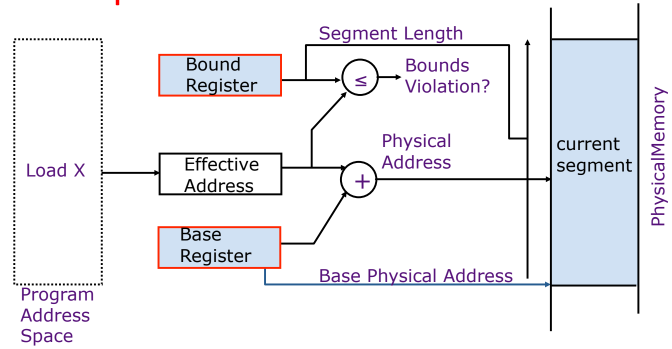
Dynamic Address Translation

Location-independent programs
- Programming and storage management ease
- ⇒ need for a *base register*

Protection
- Independent programs should not affect each other inadvertently
- ⇒ need for a *bound register*

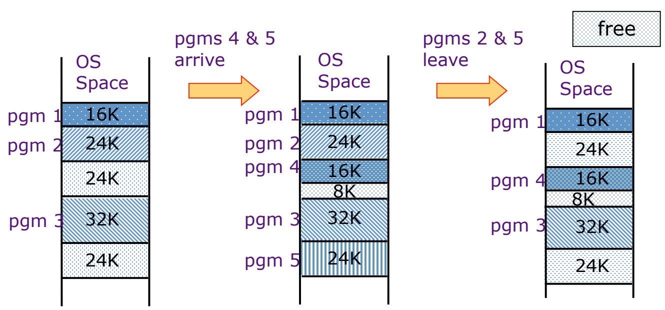


Simple Base and Bound Translation



Base and bounds registers are visible/accessible to programmer

Programs Sharing Memory



Why multiple programs?
- **Run others while waiting for I/O**

What prevents programs from accessing each other's data?

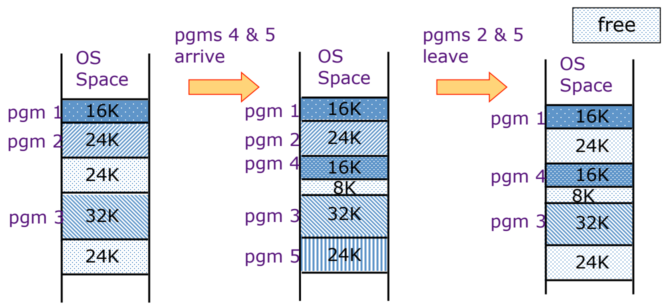
Need to Restrict Base and Bounds

- Want only Operating System to be able to change Base and Bound Registers
- Processors needs different *modes* of execution

1. *User mode*: can use Base and Bound Registers, but cannot change them
2. *Supervisor mode*: can use and change Base and Bound Registers

- Also need Mode Bit (0=User, 1=Supervisor) to determine processor mode
- Also need way for program in User Mode to invoke operating system in Supervisor Mode, and vice versa

Programs Sharing Memory



As programs come and go, the storage is "fragmented". Therefore, at some stage programs have to be moved around to compact the storage. Easy way to do this?

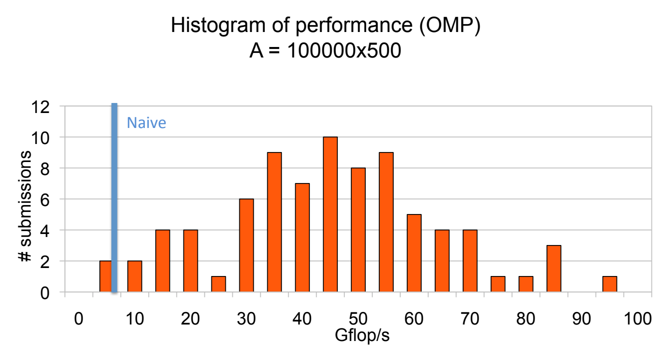
Administrivia

- Following lectures go into more depth on topics you've already seen while you work on projects
  - 3 on Protection, Traps, Virtual Memory, TLB, Virtual Machines
  - 1 on Economics of Cloud Computing
  - 1.5 on Anatomy of a Modern Microprocessor (Nehalem + Sandy Bridge, latest microarchitecture from Intel)

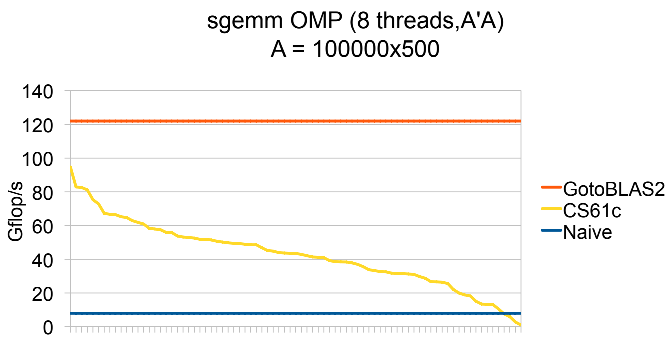
Administrivia

- Project 4: Single Cycle Processor in Logicsim
  - Due Part 2 due Saturday 11/27
  - Face-to-Face : Go to Lab on Thursday 12/2
- Extra Credit: Fastest Project 3 (due 11/29)
- Final Review: Mon Dec 6, 2-5PM (10 Evans)
- Final: Mon Dec 13 8AM-11AM (220 Hearst Gym)
  - Like midterm: T/F, M/C, short answers
  - Whole Course: readings, lectures, projects, labs, hmwks
  - Emphasize 2nd half of 61C + midterm mistakes

Project 3 Results



Project 3 Results



Avoid Memory Fragmentation?

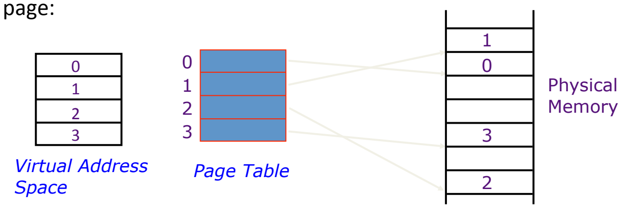
- Divide memory address space into equal sized blocks, called *pages*
  - Traditionally 4 KB or 8 KB
- Use a level of indirection to map program addresses into memory addresses
  - 1 indirection mapping per page
- Table of mappings called a *page table*

Paged Memory Systems

- Processor-generated address can be split into:

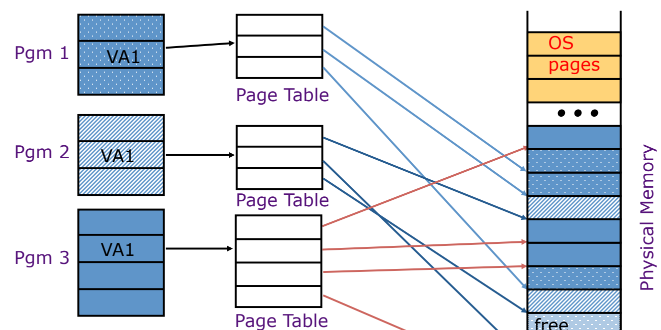
page number	offset
20 bits	12 bits

- A *page table* contains the physical address of the base of each page:



Page tables make it possible to store the pages of a program non-contiguously.

Separate Address Space per Program



- Each program has a page table
- Page table contains an entry for each program page

Paging Terminology

- Program addresses called *virtual addresses*
  - Space of all virtual addresses called *virtual memory*
- Memory addresses called *physical addresses*
  - Space of all physical addresses called *physical memory*

Processes and Virtual Memory

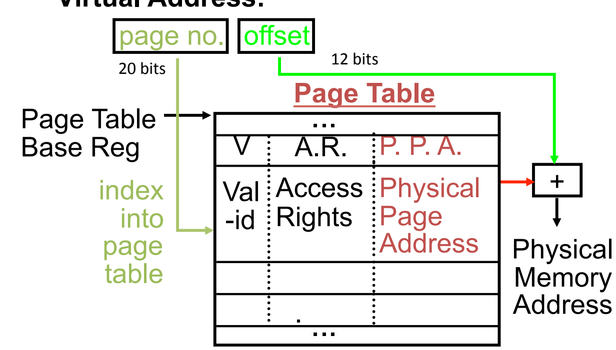
- Allow multiple *processes* to simultaneously occupy memory and provide protection – don't let one program read/write memory from another
  - Each has own PC, stack, heap
  - Like Threads, except separate address spaces
- Address space – give each program the *illusion* that it has its own private memory
  - Suppose code starts at address 0x40000000. But different processes have different code, both residing at the same address. So each program has a different view of memory.

Protection via Page Table

- Access Rights checked on every access to see if allowed
  - Read: can read, but not write page
  - Read/Write: read or write data on page
  - Execute: Can fetch instructions from page
- Valid = Valid page table entry

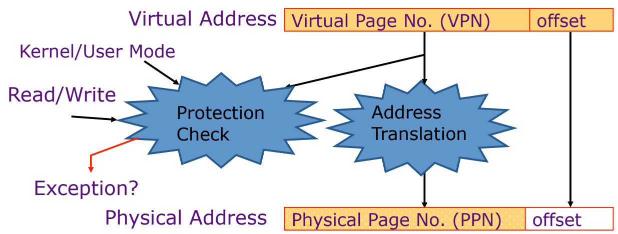
In More Depth on Page Table

Virtual Address:



Page Table located in physical memory

Address Translation & Protection



- Every instruction and data access needs address translation and protection checks

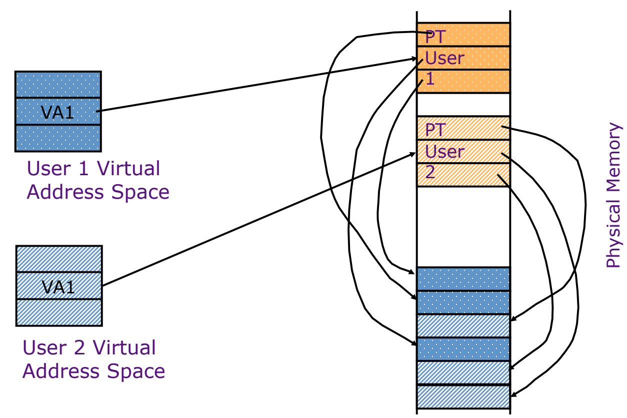
Analogy

- Book title like virtual address
- Library of Congress call number like physical address
- Card catalogue like page table, mapping from book title to call #
- On card, available for 2-hour in library use (vs. 2-week checkout) like access rights

Where Should Page Tables Reside?

- Space required by the page tables is proportional to the address space, number of users, ...
  - ⇒ Space requirement is large
  - ⇒ Too expensive to keep in registers
- Idea: Keep page tables in the main memory
  - needs one reference to retrieve the page base address and another to access the data word
    - ⇒ *doubles the number of memory references!*

Page Tables in Physical Memory



How can offer Virtual Memory but avoid doubling memory accesses?

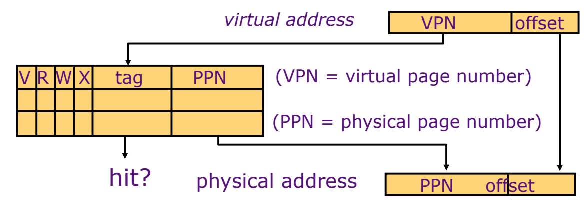
- Caches suggest that there is temporal and spatial locality of data
- But locality of data really means locality of addresses of data
- Also locality of translations of virtual page addresses into physical page addresses?
- For historical reasons, called *Translation Lookaside Buffer* (*TLB*)
  – More accurate name is *Page Table Address Cache*

Translation Lookaside Buffers

Address translation is very expensive!
Each reference becomes 2 memory accesses

Solution: *Cache translations in TLB*

TLB hit ⇒ *Single Cycle Translation*
TLB miss ⇒ *Access Page-Table to refill*



TLB Design

- Typically 32-128 entries, usually fully associative
  – Each entry maps a large page, hence less spatial locality across pages
- A memory management unit (MMU) walks the page tables and reloads the TLB

"And In Conclusion"

- Can separate Memory Management into orthogonal functions:
  – *Translation* (mapping of virtual address to physical address)
  – *Protection* (permission to access word in memory)
  – But most modern systems provide support for all functions with single page-based system
- All desktops/servers have full demand-paged virtual memory
  – Portability between machines with different memory sizes
  – Protection between multiple users or multiple tasks
  – Share small physical memory among active tasks
  – Simplifies implementation of some OS features
- Hardware support: User/Supervisor Mode, invoke Supervisor Mode, TLB, Page Table Register

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS152, CS252