

Week 5.2

Virtual memory

School of Information Technology and Electrical Engineering
The University of Queensland

Coming up

- | Lectures - Today
 - Memory Management
 - | Virtual Memory
- | Tuesday
 - debugging and Ass2
- | Friday week
 - User-space memory management.
 - | Memory bugs

Recall from Week One

- | Operating Systems provide *abstractions* to
 - make computer hardware easier to use
 - | for user, programmer, system administrator...
 - manage hardware resources
- | Example abstractions

Abstraction	Resource
Virtual Memory	Memory
Processes	CPU time+
Sockets, etc	Network
File systems	Disk space

Objectives of Memory Management

- | In a system with multiple running processes, program can not know code/data addresses before execution
 - Need to support **relocation**
- | Program logical addresses might not be the same as physical memory addresses
 - Need to support address **translation**
- | Processes should not (in general) be permitted to access memory allocated to other processes
 - Need to support **protection**

Objectives of Memory Management (cont.)

- | Sometimes, multiple processes should be able to access the same memory
 - Need to support memory **sharing**
- | Processes can dynamically change the amount of memory they need to access
 - Need to support **allocation**
- | Processes may need more memory than a machine physically has
 - Need to support **paging (aka swapping)**
 - | data is moved between primary storage (memory) and secondary storage (disk)

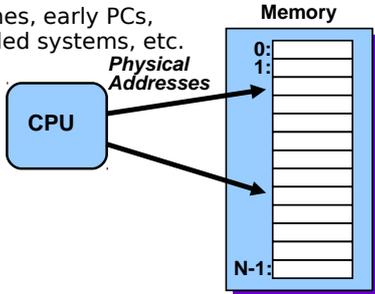
Virtual Memory

- | **Virtual memory** is an abstraction that helps in the management of memory
 - Processes see a *virtual memory* that is different from the physical memory
 - | e.g.
 - different size (smaller or larger)
 - different addresses
 - Can be used to support the objectives listed on prior slides

A System with Physical Memory Only

Examples:

- most Cray machines, early PCs, nearly all embedded systems, etc.



Addresses generated by the CPU correspond directly to bytes in physical memory

Paging

One of several methods of implementing virtual memory

Divide physical memory into fixed-sized blocks called **page frames**

- size is power of 2, usually between 512 bytes and 8192 bytes

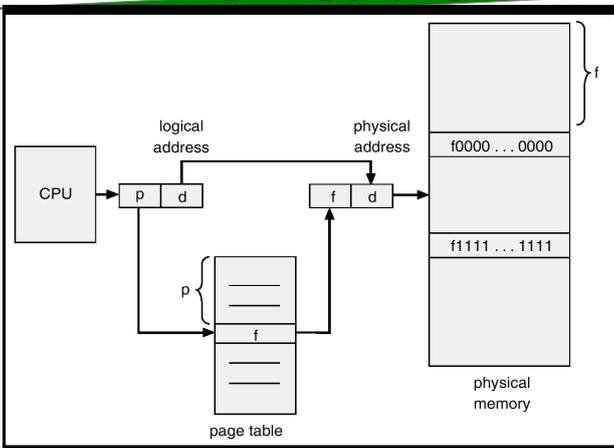
Divide logical memory into blocks of same size called **pages**

Keep track of all free page frames

To run a program of size n pages, need to find n free frames and load program

Set up a page table to translate logical to physical addresses

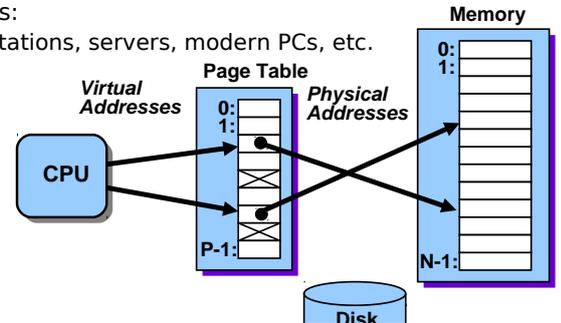
Address Translation Architecture



A System with Virtual Memory

Examples:

- workstations, servers, modern PCs, etc.



Address Translation: Hardware converts virtual addresses to physical addresses via OS-managed lookup table (**page table**)

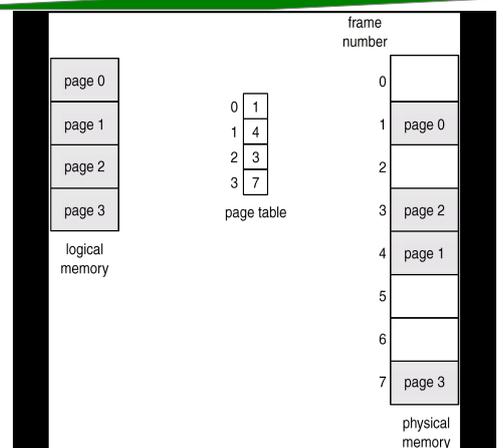
Address Translation Scheme

Address generated by CPU is divided into:

- Page number (p)** - used as an index into a page table which contains base address of each page in physical memory
- Page offset (d)** - combined with base address to define the physical memory address that is sent to the memory unit



Paging Example

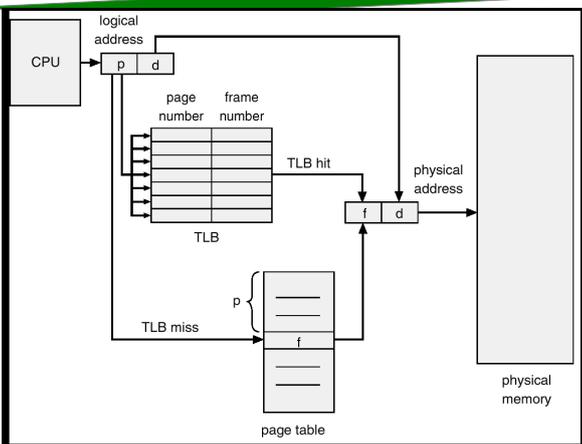


BREAK

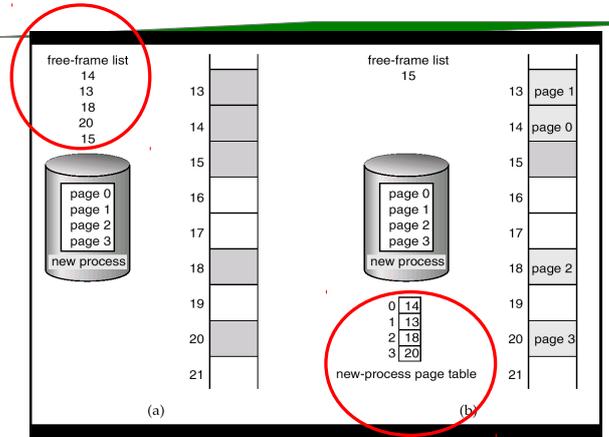
Implementation of Page Table

- Page table is kept in main memory
- In this scheme every data/instruction access requires two memory accesses
 - One for the page table and one for the data/instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or **translation look-aside buffers (TLBs)**

Paging Hardware With TLB



Free Frames



Before allocation After allocation 14

Aside: Associative Memory

- Associative memory allows parallel search

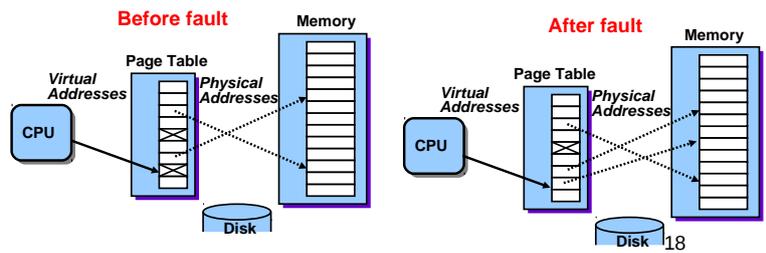
Page #	Frame #

Address translation (A' , A'')

- If A' is in associative register, get frame # out.
- Otherwise get frame # from page table in memory

Page Faults

- What if an object is on disk rather than in memory?
 - Page table entry indicates virtual address not in memory
 - OS exception handler invoked to move data from disk into memory
 - current process suspended, others can resume
 - OS has full control over placement, etc.



Page Replacement Algorithms

- On a page fault, which page should be removed to make space for incoming page?
 - Optimal** approach - choose page not needed until furthest in future
 - Impossible to predict the future!
- Many algorithms possible
 - NRU** - Not recently used
 - FIFO** - First-in, First-out
 - LRU** - Least Recently Used
 - NFU** - Not Frequently Used
 - Clock**
 - Working Set**
 - Working Set Clock**
- Not a topic for this course

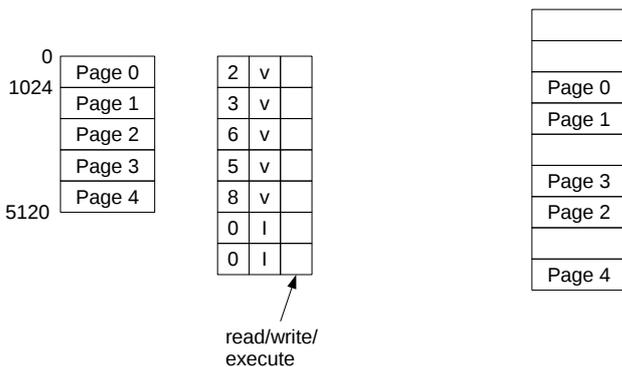
19

Memory Protection

- Memory protection implemented by associating protection bit with each frame
- Valid-invalid bit attached to each entry in the page table:
 - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
 - "invalid" indicates that the page is not in the process' logical address space

20

Memory protection (ctd)



21

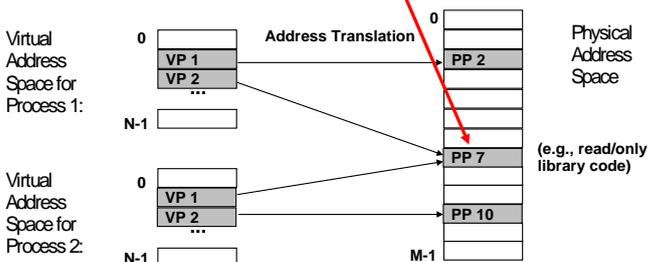
Segmentation Faults

- Segmentation Fault arises when
 - Accessing invalid memory page
 - Trying to write to a read-only page

22

Separate Virtual Address Spaces

- Each process has its own virtual address space
 - separate page tables
 - some pages may be shared



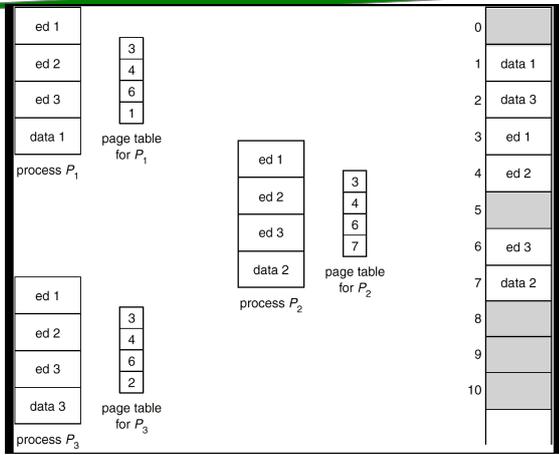
23

Shared Pages

- Shared code
 - One copy of read-only code shared among processes (i.e., text editors, compilers, window systems)
 - Shared code must appear in same location in the virtual address space of all processes.
- Private code and data
 - Each process keeps a separate copy of the code and data
 - The pages for the private code and data can appear anywhere in the virtual address space
- Data can be shared also
 - Pages can be at different locations in the virtual address spaces of each process

24

Shared Pages Example



Example

- Given multi-level page table structure as described on previous slide, how much memory is needed for the page table(s) for a process using 512MB of memory?
- What about for a process using 1MB of memory?

Remember

- 512MB = 2^9 MB = $2^9 \times 2^{10} \times 2^{10}$ bytes
- 1MB = 2^{20} bytes

Exercise

- Consider a system with 36-bit physical memory addresses and 32-bit virtual memory addresses. Pages are of size 8kB (2^{13} bytes) and page table entries are 4 bytes each.
- ...

You have 2 minutes

Multi-Level Page Tables

Example:

- 4kB (2^{12} bytes) page size
- 32-bit address space
- 4-byte page table entry

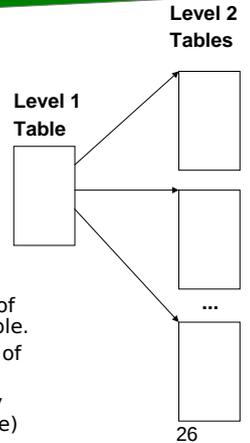
Problem:

- Would need a 4 MB page table!
- $2^{20} * 4$ bytes

Common solution

- multi-level page tables
- e.g. 2-level table (Intel processors)
 - Level 1 table: 1024 entries, each of which points to a Level 2 page table.
 - Level 2 table: 1024 entries, each of which points to a page

Note - page tables at each level typically occupy one full page (4kB in this example)



Exercise

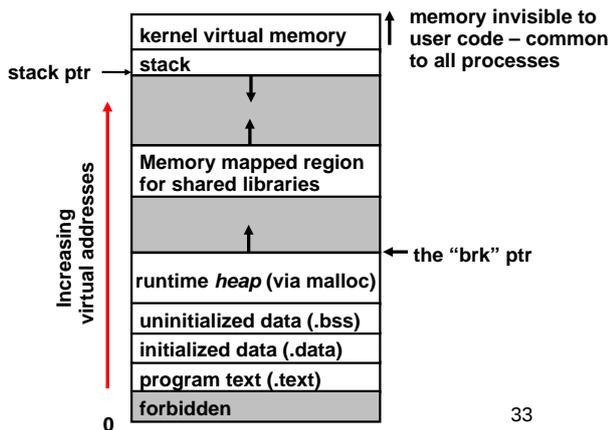
- Consider a system with 36-bit physical memory addresses and 32-bit virtual memory addresses. Pages are of size 8kB (2^{13} bytes) and page table entries are 4 bytes each.
- If the system uses single level page tables, how much memory is used by the page table for a process?
- If the system uses two-level page tables, how much memory is used by the page tables for a process using 1GB (2^{30} bytes)?

You have 2 minutes

Other Issues

- Hashed page tables
- Inverted page tables
- Segmentation
- Ideal size of pages
- Not topics for this course

Typical Process Memory Usage



Process memory image (UNIX / Linux)

User-space vs Kernel

- The operating system controls the address ranges (pages) a process can use. It does not decide how that space is used.
- Management of those pages is the responsibility of the process.
 - Usually via standard libraries.

User-space Memory management

User-space vs Kernel

- The operating system controls the address ranges (pages) a process can use. It does not decide how that space is used.
- Management of those pages is the responsibility of the process.
 - Usually via standard libraries.

Dynamic Memory Allocation

Application
Dynamic Memory Allocator
Heap Memory

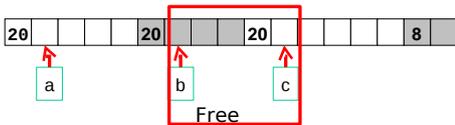
- Explicit vs. Implicit Memory Allocator
 - Explicit:** application allocates and frees space
 - E.g., malloc and free in C
 - Implicit:** application allocates, but does not free space
 - E.g. **garbage collection** in functional languages, scripting languages, and modern object oriented languages: Lisp, Java, Perl, Mathematica, ...
- Allocation
 - In both cases the memory allocator provides an abstraction of memory as a set of blocks
 - Doles out free memory blocks to application

Where Does malloc() get its Memory?

- System calls
 - brk()
 - sbrk()
- See manual pages on malloc
- See also end (section 3c)
- free() doesn't necessarily return memory to the operating system - may just keep track of it for future use by the application

37

Trashing the heap



- Now c[0] is in free space (as is the size of c).
- If this space is allocated to something else, free(c) could get very interesting.

39

Dereferencing Bad Pointers

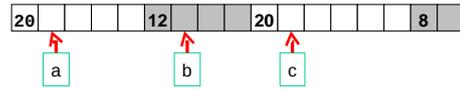
- The classic scanf bug

```
scanf("%d", val);
```

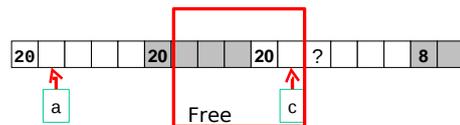
41

Trashing the heap

- Allocators may record audit information near the allocated memory. (For example the size of the allocation)



- Now consider a[4]=20; free(b);



38

Memory-Related Bugs

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

40

Reading Uninitialized Memory

- Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
    int *y = malloc(N*sizeof(int));
    int i, j;

    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            y[i] += A[i][j]*x[j];
        }
    }
    return y;
}
```

42

Overwriting Memory

- Allocating the (possibly) wrong sized object

```
int **p;
p = malloc(N*sizeof(int));
for (i=0; i<N; i++) {
    p[i] = malloc(M*sizeof(int));
}
```

43

Overwriting Memory

- Not checking max string size

```
char s[8];
get(s); // enter 1234567890
```

- Basis for classic buffer overflow attacks.

45

Overwriting Memory

- Misunderstanding pointer arithmetic

```
int* search(int* p, int val) {
    while (*p && *p!=val) {
        p+=sizeof(int);
    }
    return p;
}
```

47

Overwriting Memory

- Off-by-one error

```
int **p;
p = malloc(N*sizeof(int));
for (i=0; i<=N; i++) {
    p[i] = malloc(M*sizeof(int));
}
```

44

Overwriting Memory

- Referencing a pointer instead of the object it points to

```
void swap(int* a, int* b) {
    int c=a;
    a=b;
    c=c;
}
```

46

Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

```
int *foo () {
    int val;
    return &val;
}
```

48

Freeing Blocks Multiple Times

Nasty!

```
x = malloc(N*sizeof(int));
<manipulate x>
free(x);

y = malloc(M*sizeof(int));
<manipulate y>
free(x);
```

49

Failing to Free Blocks (Memory Leaks)

Slow, long-term killer!

```
foo() {
    int *x = malloc(N*sizeof(int));
    ...
    return;
}
```

51

Dealing With Memory Bugs

- ▮ Conventional debugger (gdb)
 - Good for finding bad pointer dereferences
 - Hard to detect the other memory bugs
- ▮ Debugging malloc (CSRI UToronto malloc)
 - Wrapper around conventional malloc
 - Detects memory bugs at malloc and free boundaries
 - Memory overwrites that corrupt heap structures
 - Some instances of freeing blocks multiple times
 - Memory leaks
 - Cannot detect all memory bugs
 - Overwrites into the middle of allocated blocks
 - Freeing block twice that has been reallocated in the interim
 - Referencing freed blocks

53

Referencing Freed Blocks

Evil!

```
x = malloc(N*sizeof(int));
<manipulate x>
free(x);
...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++) {
    y[i] = x[i]++;
}
```

50

Failing to Free Blocks (Memory Leaks)

Freeing only part of a data structure

```
struct list {
    int val;
    struct list* next;
};

void foo() {
    struct list* head=
        malloc(sizeof(struct list));
    head->val=0;
    head->next=0;
    <create and use the rest of the list>
    ...
    free(head);
}
```

52

Dealing With Memory Bugs (cont.)

- ▮ Check while executing:
 - valgrind (linux)
 - bcheck (agave)
- ▮ Garbage collection (Boehm-Weiser Conservative GC)
 - Let the system free blocks instead of the programmer

54

Memory Matters!

- | Memory is not unbounded
 - It must be allocated and managed
 - | malloc(), free() etc
 - Many applications are memory dominated
 - | Especially those based on complex, graph algorithms
- | Memory referencing bugs especially pernicious
 - Effects are distant in both time and space
- | Memory performance is not uniform
 - Cache and virtual memory effects can greatly affect program performance
 - Adapting program to characteristics of memory system can lead to major speed improvements