

Virtual Memory

CS241 Discussion Section

Week 10

4/09/07 – 4/15/07

Outline

Debugging memory allocation

Contiguous allocation and compaction

Paging and page replacement algorithms

Page tables, address translation

Debugging Memory Allocation

With LMP2, we have learned how to implement dynamic memory allocation similar to `malloc()/free()` in C.

Problem: it is very easy to go outside of the allocated space by accident, and errors often do not show up immediately.

Example:

```
int *ptr = (int *)malloc(100);  
for (i = 0; i < 100; i++)  
    ptr[i] = i;
```

What's wrong with this code? How can we help the programmer detect and avoid such errors?

Debugging Memory Allocation

Change `malloc()` and `free()` implementations to check for such errors.

`malloc()`: allocate extra space on both sides of the requested allocation, and fill this memory with a pattern (called *guard*)

`free()`: check that the guards have not been overwritten, otherwise indicate an error

Problem: Implement a Debugging MM

debug.c provides you with a very simple memory manager, similar to the one in LMP2.

Change myAlloc() and myFree() functions to check for out of bounds memory accesses

Note: similar, but more thorough and efficient techniques are used in practice.

See, for example, the *Electric Fence* memory debugger by Bruce Perens.

Memory Organization

How is program data is loaded and stored in memory?

Contiguous Allocation

Memory is allocated in monolithic segments
or *blocks*

Public enemy #1: external fragmentation

We can solve this by periodically rearranging the
contents of memory

Compaction

After numerous `malloc()` and `free()` calls, our memory will have many holes

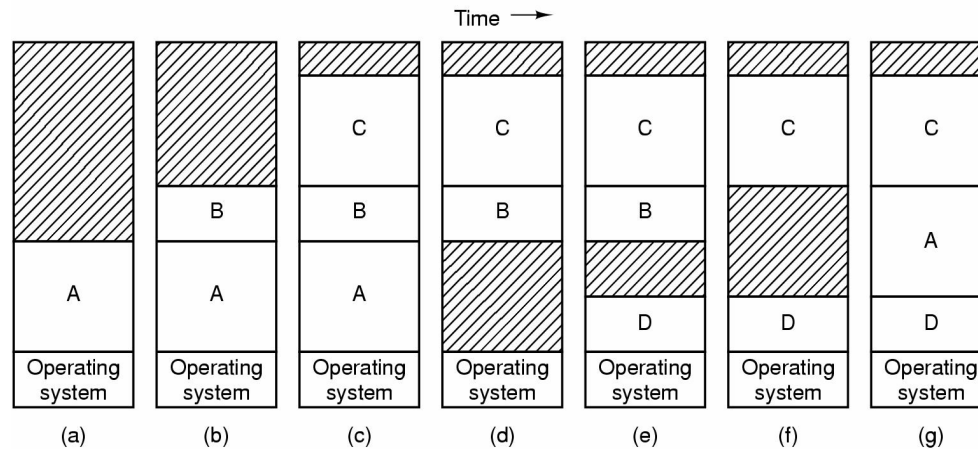
Total free memory is much greater than that of any contiguous chunk

We can *compact* our allocated memory

Shift all allocations to one end of memory, and all holes to the other end

Temporarily eliminates of external fragmentation

Compaction (example)



Lucky that A fit in there! To be sure that there is enough space, we may want to compact at (d), (e), or (f)

Unfortunately, compaction is problematic

It is very costly. How much, exactly?

How else can we eliminate external fragmentation?

Review: Paging

Divide memory into *pages* of equal size

We don't need to assign contiguous chunks

Internal fragmentation can only occur on the last page assigned to a process

External fragmentation cannot occur at all

Need to map contiguous logical memory addresses to disjoint pages

Review: Virtual Memory

RAM is expensive (but fast), disk is cheap (but slow)

Need to find a way to use the cheaper memory

Store memory that isn't frequently used on disk

Swap pages between disk and memory as needed

Treat main memory as a cache for pages on disk

Page Replacement

We may not have enough space in physical memory for all pages of every process at the same time.

But which pages shall we keep?

Use the history of page accesses to decide

Also useful to know the *dirty* pages

Page Replacement Strategies

It takes two disk operations to replace a dirty page, so:

Keep track of dirty bits, attempt to replace clean pages first

Write dirty pages to disk during idle disk time

We try to approximate the optimal strategy but can seldom achieve it, because we don't know what order a process will use its pages.

Best we can do is run a program multiple times, and track which pages it accesses

Page Replacement Algorithms

Optimal: last page to be used in the future is removed first

FIFO: First in First Out

Based on time the page has spent in main memory

LRU: Least Recently Used

Locality of reference principle again

MRU: most recently used = removed first

When would this be useful?

LFU: Least Frequently Used

Replace the page that is used least often

Example

Physical memory size: 4 pages

Pages are loaded on demand

Access history: 0 1 2 3 4 0 1 2 3 4 ...

Which algorithm does best here?

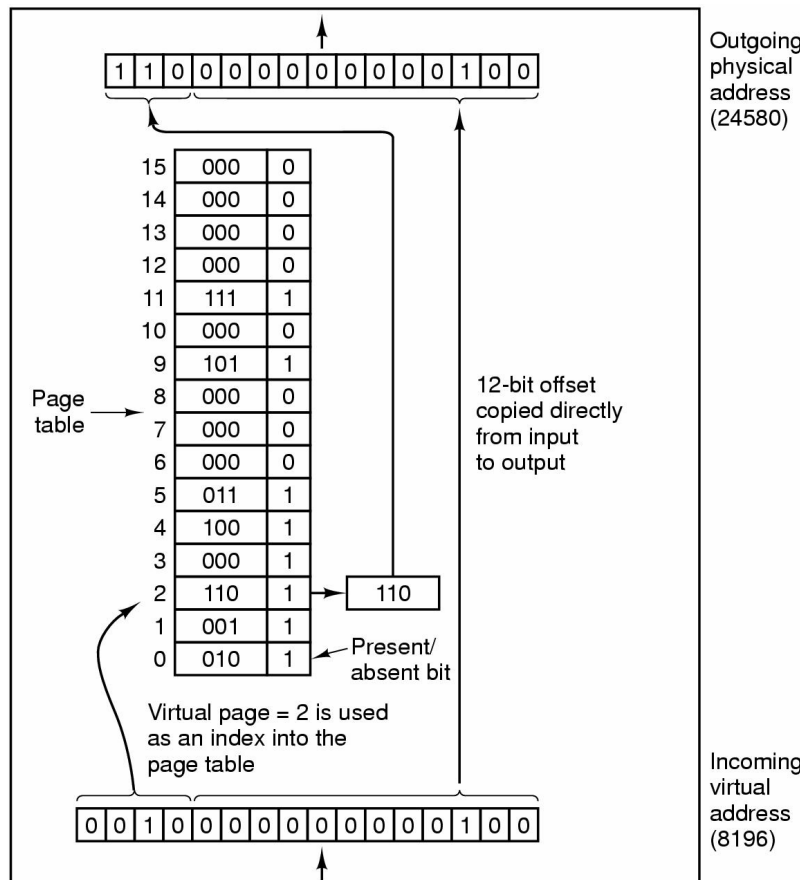
Access history: 0 1 2 3 4 4 3 2 1 0 ...

And here?

Page Tables and Address Translation

Matching virtual addresses to
physical memory locations

Review: Page Tables



64kB logical address space

8 pages * 4kB == 32kB RAM

16-bit virtual address consists of:

Page number (4 bits)

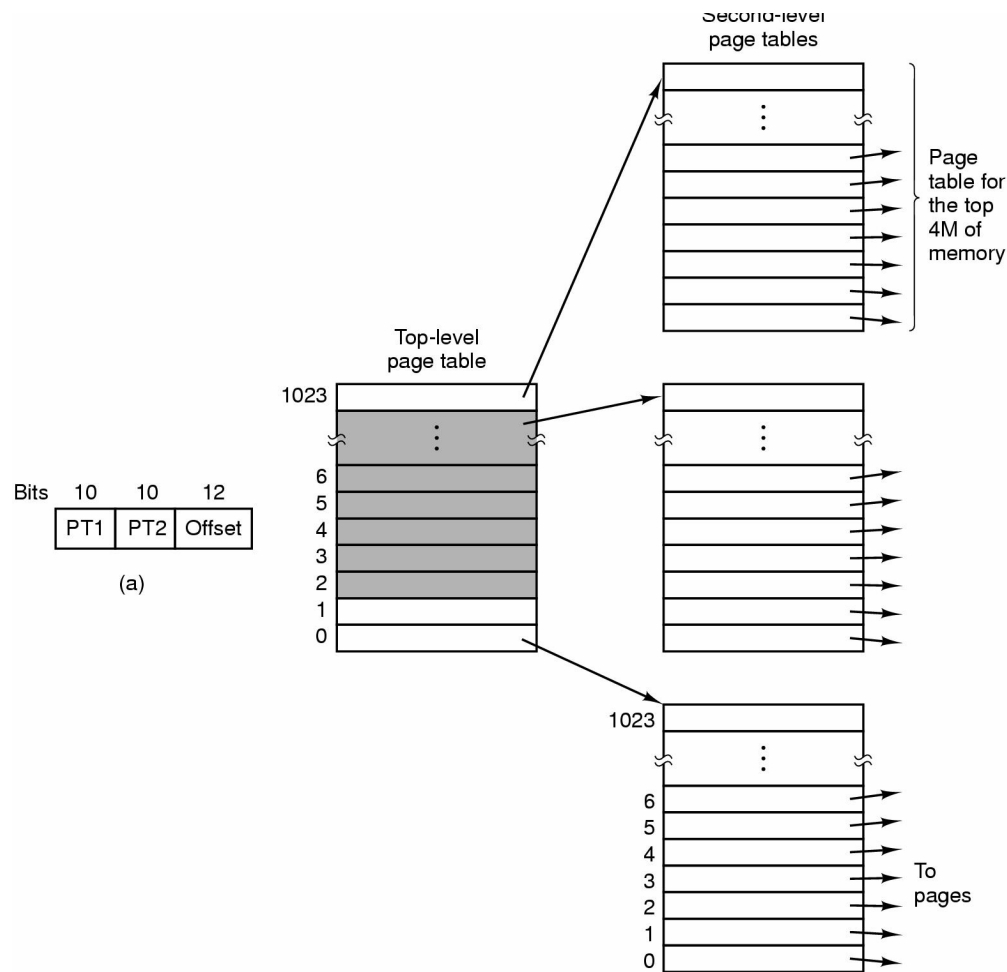
Page offset (12 bits)

Virtual page number – table index

Physical frame number – value

Present bit – is page in memory?

Multi-level Page Tables



Instead of one large table, keep a tree of tables

Top-level table stores pointers to lower level page tables

First n bits of the page number == index of the top-level page table

Second n bits == index of the 2nd-level page table

Etc.

Example: Two-level Page Table

32-bit address space (2GB)

12-bit page offset (4kB pages)

20-bit page address

First 10 bits index the top-level page table

Second 10 bits index the 2nd-level page table

10 bits == 1024 entries * 4 bytes == 4kB == 1 page

Need three memory accesses to read a memory location

Why use multi-level page tables?

Split one large page table into many page-sized chunks

Typically 4 or 8 MB for a 32-bit address space

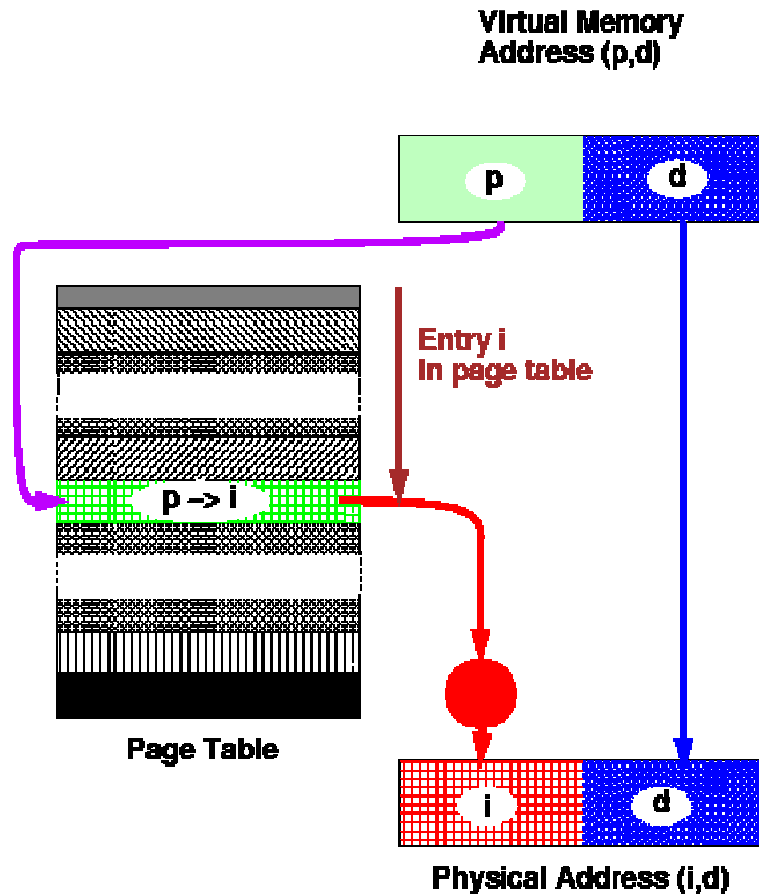
Advantage: less memory must be reserved for the page tables

Can swap out unused or not recently used tables

Disadvantage: increased access time on TLB miss

$n+1$ memory accesses for n -level page tables

Inverted Page Table



“Normal” page table

Virtual page number == index
Physical page number == value

Inverted page table

Virtual page number == value
Physical page number == index

Need to scan the table for the right value to find the index

More efficient way: use a hash table

Example

Page Table

Index Present Virtual Addr

0	0	
1	1	
2	0	
3	1	
4	1	1010
5	0	
6	1	

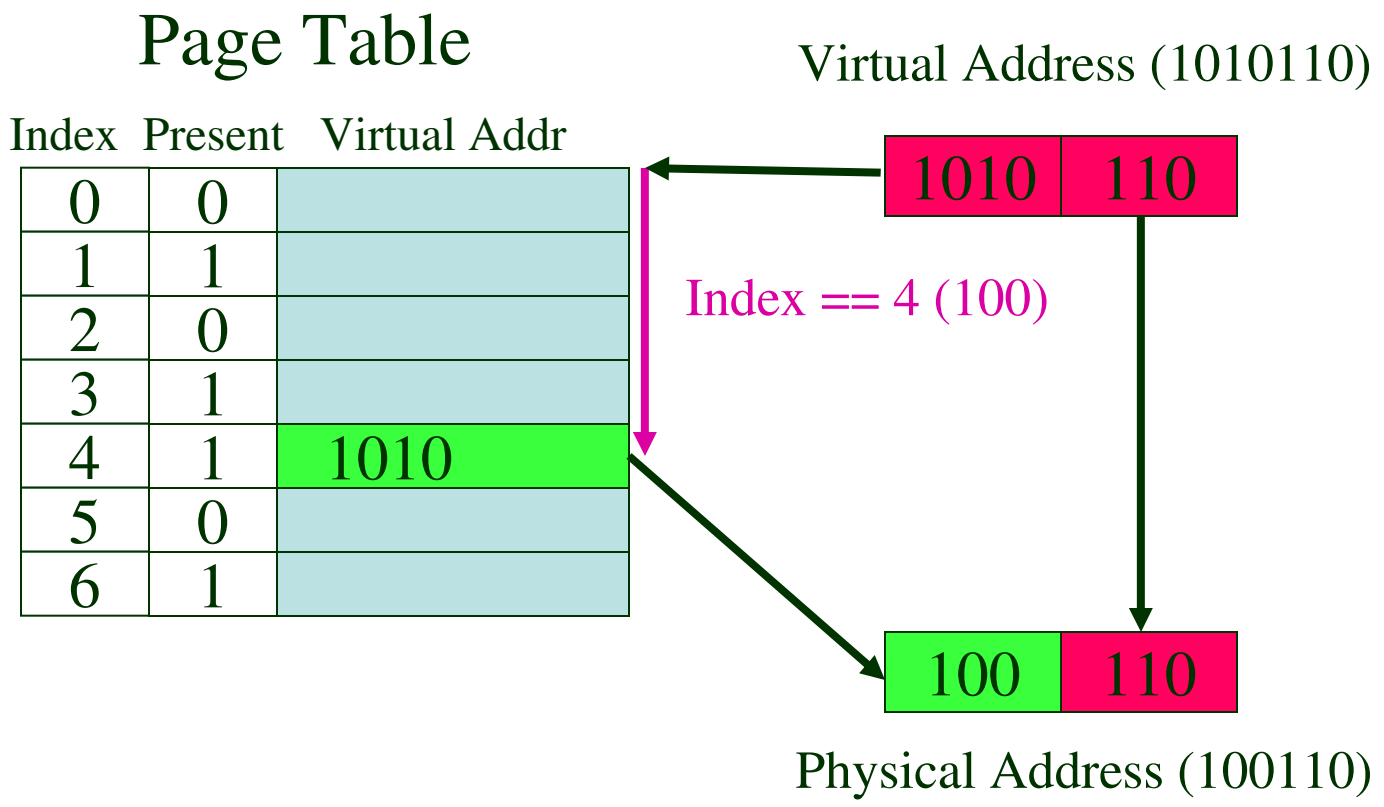
Virtual Address (1010110)

1010 110

Index == 4 (100)

100 110

Physical Address (100110)



Implementation

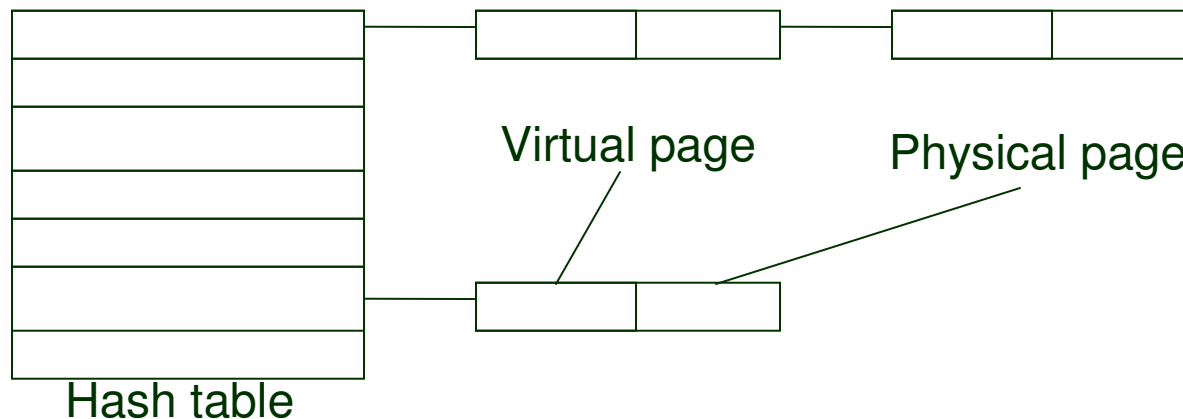
TLB is same as before

TLB miss is handled by software

In-memory page table is managed using a hash table

Number of entries \geq number of physical frames

Not found: page fault



Why use inverted page tables?

One entry for each page of physical memory
vs. one per page of logical address space

Advantage: less memory needed to store the page table

If address space \gg physical memory

Disadvantage: increased access time on TLB miss

Use a hash table to limit the search to one – or at most a few extra memory accesses

Recap

Debugging memory allocation

Contiguous allocation + compaction: slow

Paged virtual memory

- Can address a much larger space than we have physical memory

- Need to replace pages loaded in memory using an appropriate replacement algorithm

Address translation

- Multi-level and inverted page tables