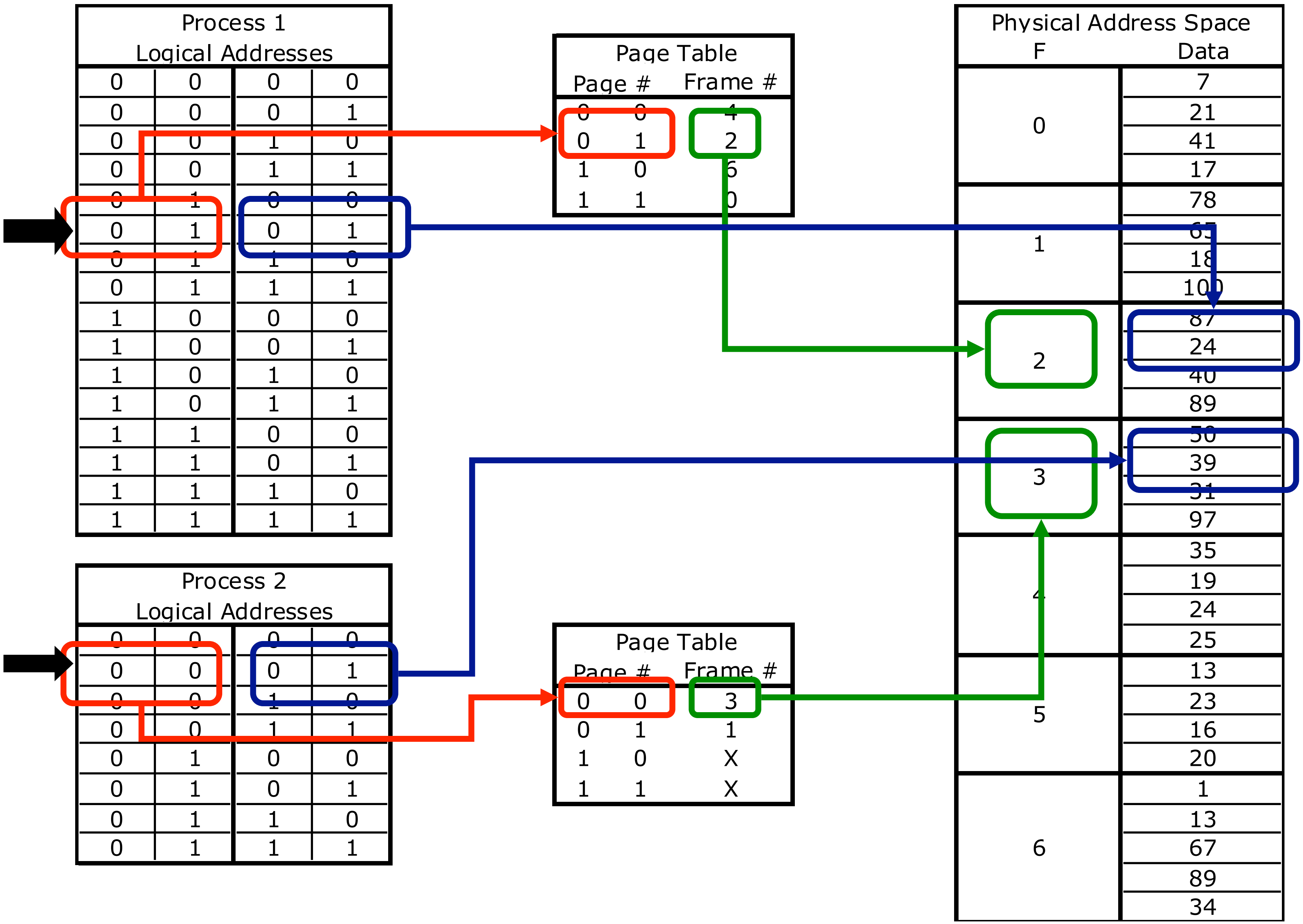
Memory Management
Virtual Memory

Chapter 8-9



Process 1
Logical Addresses
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
Process 2
Logical Addresses
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
Page Table
Page # Frame #
0 0 4
0 1 2
1 0 6
1 1 0
Page Table
Page # Frame #
0 0 3
0 1 1
1 0 X
1 1 X
Physical Address Space
F Data
0 7 21 41 17
1 78 65 18 100
2 87 24 40 89
3 50 39 31 97
4 35 19 24 25
5 13 23 16 20
6 1 13 67 89 34

Paging in the real world

- 1 page table per process.
  - Page table may too big for **contiguous allocation**
  - What did we do the first time to solve contiguous allocation?

Page the Page Table too..

More paging issues

- Even with multi-level paging strategies, page table(s) still quite large.

- One page table for each *logical* address space (1 per process).

Instead, one page table for *physical* addresses (frames).
Inverted Page Table

Virtual Memory

- Paging does not solve all our problems…

Remove the restriction of having an entire process in memory…

Program memory usage tends to be sparse…

This creates an “unlimited” address space.

Page Fault

- Step 1: Translate Logical Address to determine page # & consult page table
- Step 2: If page is invalid, trap to OS
- Step 3: OS locates page on backing store
- Step 4: Find free frame in memory and copy page from backing store
- Step 5: Update page table
- Step 6: Restart program that initiated the request

Demand paging

- We must decide *which* pages to bring into memory when program begins:
  - Pure Demand Paging: Only ***page*** pages when requested (when process begins, it has only one page
  - Additional pages are only brought in when used
  - Lazy Swapping / Lazy Pager

Page Replacement

- To implement demand paging, we need two things:
  - Frame Allocation Algorithm
  - Page Replacement Algorithm

Overall Goal: Reduce Page Faults

Frame Allocation

- We'll get back to this… but
  - We'll be selecting a # of frames to allocate a specific process.
  - No matter what, more frame generally means less page faults (on average)
    - There are some anomalous circumstances where this isn't the case, for **every** situation

Page Replacement

- On a page fault, we must find a free frame in memory - what if its full?

- Select a ***victim*** frame to remove from memory to make room (put on backing store)
  - – Page fault time doubles… (16,000,000ns)
  - – Always copy back to backing store?

Page Replacement Analysis

- To simulate a process running, we list the sequence of memory addresses:
  - We can keep track of which memory locations result in a page fault
  - This is called a ***reference string***

- We can simplify this to only list the page numbers to be accessed.

Reference Strings

0100, 0432,

0101, 0612, 0102, 0103, 0104,

0101, 0611, 0102, 0103, 0104,

0101, 0610, 0102, 0103, 0104,

0101, 0609, 0102, 0105

If Page Size is 100 Bytes:

1, 4,1, 6, 1, 6, 1, 6, 1, 6, 1

What is this code probably doing?

How many page faults if we have 1 frame?

Page Replacement Algorithms

- **7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1**
  - **3 Frames**

- FIFO: Replace Oldest page
- Optimal: Replace page that won't be used for the longest time
- LRU: Replace the least recently used
- Second-Chance FIFO

LRU Implementation

- Typically we don't keep track of exact timing…
  - Reference bits -> 00000
  - On each page access, set leftmost bit to 1
    - 10000
  - At given interval, shift **all page's** string right
    - 01000
- To find least recently used, find smallest number (likely to be many ties --- ok.
- To some extent, second chance FIFO is just LRU with one bit.

Thrashing

- Very high rates of paging is **very** bad
  - Typically called "thrashing"
- Causes:
  - CPU tries to allocate as many processes as possible to maintain CPU utilization
  - Page Faults reduce CPU Utilization
  - CPU responds by allocating even more processes…

What must this do to the page fault rate?

Working Set

- Programs exhibit locality -
  - The use the same group of memory addresses repeatedly for “relatively” long time periods
- Working Set: Pages in which a high percentage (90-99%) of memory addresses reside on
  - Working Set holds for some time period: Δ

Avoiding Thrashing

- Approximate Working Set

- OS must allocate enough frames for a program to fit its current working set
  - If it cannot, must suspend the process

- Working Set leads to periodic page fault surges…

Program Structure

```
int data[128][128];

for ( i = 0; i < 128; i++)
  for ( j = 0; j < 128; j++ )
     data[i][j] = 0;
```

Page size = 128 integers (1024 bytes)

Page Size?

- Large page size means…
  - Small page table
  - More internal fragmentation
  - Less IO time (more efficient)
  - Extends TLB reach
- Small page size means…
  - Larger page table
  - Less internal fragmentation
  - More IO time (less efficient)
  - Lower TLB reach

In general, grow page sizes based on how much memory you have…

Exam 3 – Next Week

- Covers Memory Management basics
  - Logical vs. Physical Addresses
  - Compile / Load / Runtime Binding
  - Standard paging
    - Computing page size, # of pages based on address scheme
    - Resolving pages to frames
- Advanced Paging
  - Multi-Level Paging
  - TLB
  - Inverted Page Table

Exam 3 – Next Week

- Virtual Memory
  - Understand each step towards resolving a page fault
  - Be able to compute reference string
  - Explain role of backing store
  - Calculate # of page faults based on replacement strategy
- Frame allocation
  - Explain Working Set