Memory Management

Combining These . . .

- Segmented paging: segment the page table
  - Each entry in segment table contains base, length of part of page table
- Paged segmentation: page the segment table
  - Segment table contains segment lengths, page table base (virtual) address

Segmented Paging

- Virtual address is (page number, page offset)
- In this address, page number is (segment number, segment offset)
- To get physical address from virtual address:
  1. Get segment number and add STBR
  2. Get segment table entry
  3. Compare segment offset with page table length; if offset greater, it's an illegal reference
  4. Get page table base, add segment offset
  5. Get page table entry
  6. Use the frame number in it and page offset to get physical address

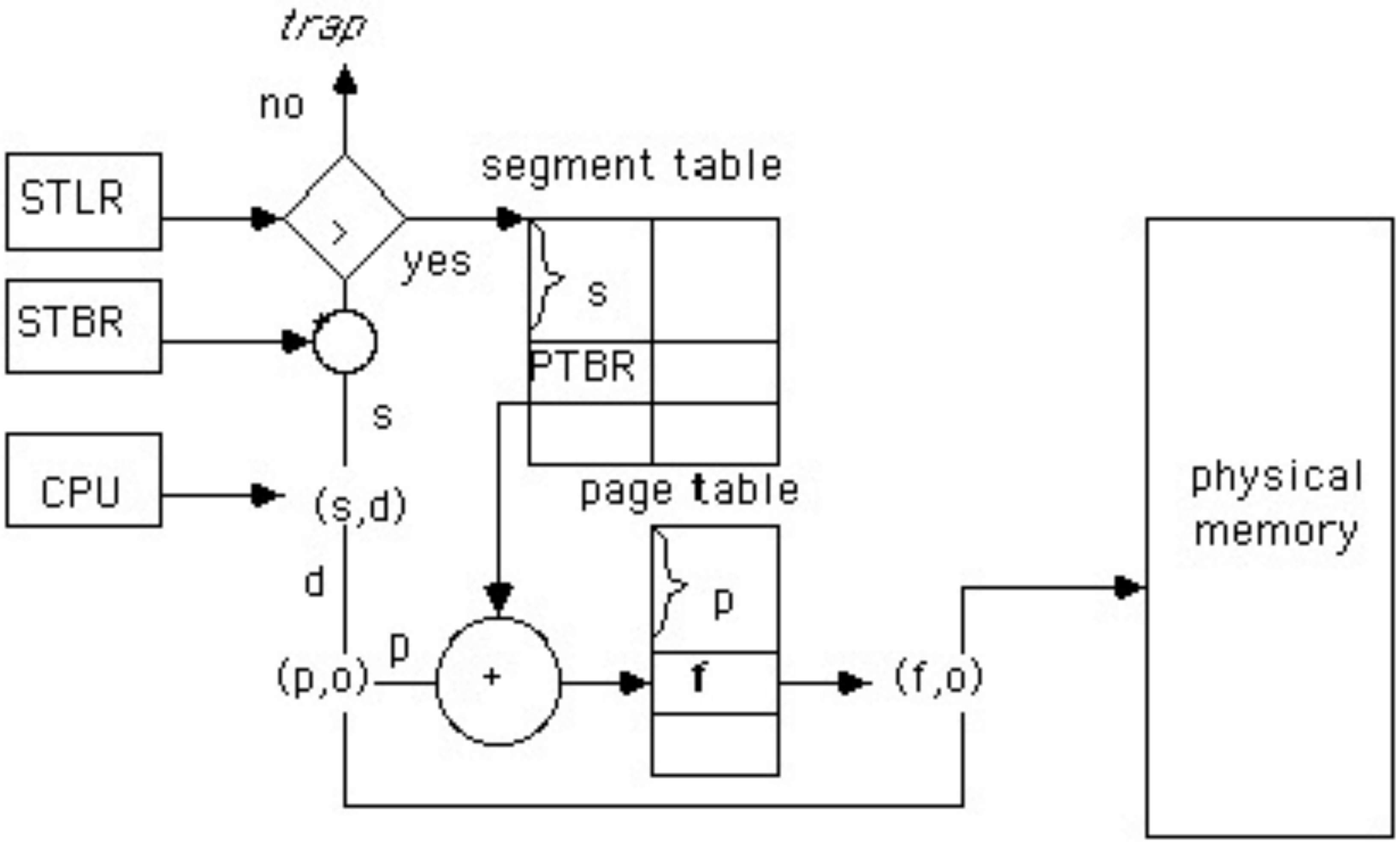
Segmented Paging

- Used when most of page table is empty
- This happens when address space is large and programs use just a small fraction of the memory space

Paged Segmentation

- Virtual address is (segment number, segment offset)
- In this address, segment offset is (page number, page offset)
- Entries in segment table are (page table base, page table length)
- To get physical address from virtual address:
  1. Get segment number and compare it to segment table length; if number greater, it's an illegal reference
  2. Add STBR to segment number
  3. Get segment table entry
  4. Add page number to page table base address
  5. Get page table entry
  6. Use the frame number in it and page offset to get physical address

Paged Segmentation



Paged Segmentation

- Used when segment sizes are large and external fragmentation is a problem
- Also when fining free space takes a long time
- As with paging, last page of a segment may not be full
    - On average, half a page of internal fragmentation
- But no external fragmentation!

Some Philosophy: What Is Virtual Memory

- Virtual memory allows the execution of processes not completely in memory; useful because . . .
  - Programs often have code to handle unusual error conditions; in many cases, this code may almost never be used;
  - Arrays and tables are often allocated more memory than needed; and
  - Some options and features are seldom used and even if all are used, they are seldom used all at once.

Benefits

- Some added benefits of not requiring the whole program to be in memory
  - Programs are not constrained by the amount of available physical memory;
  - More users can run at the same time, increasing CPU utilization and throughput, without increasing response or turnaround times;
  - It takes less I/O to load or swap a process into memory, so each user process seems to run faster

How To Do It

- Overlays
  - One process replaces another in memory
  - In Linux, *execve*(2) does this
- Demand paging
  - Bring in pages only when needed

Overlays

- Keep only the instructions and data needed at a given time in memory
  - When needed, new instructions and data are loaded into space occupied by instructions and data no longer needed
- A multiple pass compiler or assembler is an example
  - Pass 1 routines are loaded into memory and executed
  - When done, control jumps to the overlay driver
  - Overlay driver brings Pass 2 routines into memory, overwriting Pass 1 routines
  - Pass 2 routines are executed

Example: 2-Pass Assembler

- Memory is 32K words
- Full assembler totals 37K words, so doesn't fit in memory
  - Pass 1 takes 8K words
  - Pass 2 takes 10K words
  - Symbol table takes 14K words
  - Routines common to both take 5K words
- Use overlays: overlay driver takes 2K words
  - Overlay 1: pass 1, symbol table, common routines, overlay driver: 29K words
  - Overlay 2: pass 2, symbol table, common routines, overlay driver: 31K words
- So it fits!

Dynamic Loading

- Program loads routines from secondary storage into memory as needed
  - Routines kept in relocatable format.
- When program is loaded and executed, and it calls a routine, the system:
  1. Checks to see if the called routine is in memory
  2. If not, that routine is loaded and the relevant tables are updated
  3. The called routine is executed

Dynamic Loading

- Advantage: only needed routines are loaded into memory
- But operating system may not provide support
  - So user must design and program an overlay structure or loading
  - As program is large, this may get confusing
  - *Much* preferable to have automatic mechanisms to do this
- So . . .

Demand Paging

- Pages reside on secondary storage (area is called “swap space”)
- Page brought in only when needed
- Advantages
  - Decreases swap time, amount of physical memory needed
  - Increases the degree of multiprogramming
- Set invalid bit for page table entries referring to pages not in memory
- When process references such a page, the process *page faults*
  - This signals a page needs to be brought in

Hardware Support

- A page table
  - It can have entries marked invalid via a valid/invalid bit or some special value of protection bits; and
- Backing store for pages not in memory

Pure Demand Paging

- Processes start with no pages in memory
- Execution of first instruction causes page fault
  - And the first page is loaded into memory . . .

Handling Page Faults

1. Page fault causes trap to the operating system.
2. User registers and program state are saved.
3. Operating system determines that trap was page fault trap.
4. Operating system checks that page reference was legal
   - If so, it determines location of page on backing store
5. The operating system initiates a read of the page from the backing store to a free frame:
   a. Request waits in appropriate queue for device;
   b. It waits for the device seek and rotational latencies;
   c. Page transfer begins.

Handling Page Faults

6. While waiting for I/O to complete, operating system reallocates CPU to another process
7. When I/O completes, interrupt occurs
8. System saves registers and program state of current process
9. Operating system determines interrupt was from backing store
10. It updates page table (and other tables) to show page is now in memory
11. Operating system now reallocates CPU
12. Appropriate process is (re)started

Performance

- *a* is memory access time
- *f* is time to service a page fault
- *p* is probability of a page fault
- Then effective memory access time *emat* is:

$$emat = pf + a(1-p)$$

Example

- Assume average page fault service time is 10ms
- Assume memory access time is 1μs

  emat = (1 – *p*) 1μs + *p*10ms = (1 - *p*) + 10000 μs = (1 + 9999*p*) μs

- So effective memory access time is proportional to the page fault probability
- If $p = \frac{1}{1000}$ then $emat = \frac{9999}{1000} \times 1\mu s + \frac{1}{1000} \times 10ms = \frac{10999}{1000}\ \mu s \approx 11\ \mu s$
- So with demand paging, effective memory access time is 11 times slower than without it

Example

- To get less than 10% degradation, choose p such that $1.1 \geq 1 + 9999p$
  - Comes from setting *emat* = 1.1 and solving earlier equation for p
- So $\frac{1}{10} \geq 9999p$ or $p \leq \frac{1}{99990} \approx \frac{1}{100000}$
- So a page fault should occur no more than once in about 100,000 memory references

Page Replacement

- Once a page is in memory, it stays there
- Problem: memory assigned to process gets full
  - Page fault occurs
  - Trap to operating system, sees it is a page fault
  - Operating system locates needed page on backing store
  - But there is no free frame!
- What does the operating system do?

What Operating System Can Do

- Terminate the program
  - The whole point of paging is to hide paging from the user
- Swap out process temporarily
- Replace some pages in memory
  - If no free frames, find some frame not in use
  - Write its contents to backing store and update relevant tables
  - Load new page into that frame

Page Fault Service Routine

1. Locate desired page on backing store
2. Look for a free frame
   - If one found, use it
   - If not, select victim frame, write it to backing store, update tables to reflect this
3. Read in the new page and update tables to reflect this
4. Resume the user process

Optimization: Dirty Bit

- If no free frames, this requires two transfers (one out and one in)
- Optimization: associate with each frame a *dirty bit*, set when the page is written to
- If victim page has dirty bit set, it *must* be written to backing store
- If victim page does not have dirty bit set, don't bother writing the page to backing store

Stepping Back . . .

- Size of virtual memory is no longer constrained by size of computer's physical memory
- Pages can be moved in and out as needed
- This completes separation of virtual memory from physical memory

- To do this, system needs two algorithms
  - A page replacement algorithm
  - A frame allocation algorithm

Page Replacement Algorithms

- These select victim page (that is, the page to be removed)
- Reference string is the set of page numbers that a process references
  - Example: system page size is 100 words, and the following memory references occur: 100 432 101 612 102 103 104 101 611 102 103
  - Pages referenced: 1 4 1 6 1 1 1 1 6 1 1
  - Corresponding reference string: 1 4 1 6 1 6 1

First In, First Out (FIFO)

- Oldest page selected for removal
- Example: reference string is 1 2 3 4 5 4 2 3 4 5 1 2 3 4 5
- 3 frames available: total of 14 page faults

	1	2	3	4	5	4	2	3	4	5	1	2	3	4	5
Frame 1	1	1	1	4	4	4	4	3	3	3	1	1	1	4	4
Frame 2		2	2	2	5	5	5	5	4	4	4	2	2	2	5
Frame 3			3	3	3	3	2	2	2	5	5	5	3	3	3
pf	•	•	•	•	•		•	•	•	•	•	•	•	•	•

First In, First Out (FIFO)

- Example: reference string is 1 2 3 4 5 4 2 3 4 5 1 2 3 4 5
- 4 frames available: total of 10 page faults

	1	2	3	4	5	4	2	3	4	5	1	2	3	4	5
Frame 1	1	1	1	1	5	5	5	5	5	5	5	5	5	4	4
Frame 2		2	2	2	2	2	2	2	2	2	1	1	1	1	5
Frame 3			3	3	3	3	3	3	3	3	3	2	2	2	2
Frame 4				4	4	4	4	4	4	4	4	4	3	3	3
pf	•	•	•	•	•						•	•	•	•	•

Belady’s Anomaly

	1	2	3	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	4	4	4	5	5	5	5	5	5
Frame 2		2	2	2	1	1	1	1	1	3	3	3
Frame 3			3	3	3	2	2	2	2	2	4	4
pf	•	•	•	•	•	•	•			•	•	

3 frames: 9 page faults

Belady's Anomaly

	1	2	3	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	4	4	4	5	5	5	5	5	5
Frame 2		2	2	2	1	1	1	1	1	3	3	3
Frame 3			3	3	3	2	2	2	2	2	4	4
pf	•	•	•	•	•	•	•			•	•	

3 frames: 9 page faults; 4 frames: 10 page faults!

	1	2	3	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	1	1	1	5	5	5	5	4	4
Frame 2		2	2	2	2	2	2	1	1	1	1	5
Frame 3			3	3	3	3	3	3	2	2	2	2
Frame 4				4	4	4	4	4	4	3	3	3
pf	•	•	•	•			•	•	•	•	•	•

Optimal (OPT)

- Example: reference string is 1 2 3 4 5 4 2 3 4 5 1 2 3 4 5
- 4 frames available: total of 7 page faults

	1	2	3	4	5	4	2	3	4	5	1	2	3	4	5
Frame 1	1	1	1	1	5	5	5	5	5	5	1	1	1	1	5
Frame 2		2	2	2	2	2	2	2	2	2	2	2	2	2	2
Frame 3			3	3	3	3	3	3	3	3	3	3	3	3	3
Frame 4				4	4	4	4	4	4	4	4	4	4	4	4
pf	•	•	•	•	•						•				•

Optimal (OPT)

- Select page that will not be used for longest period of time
- Example: reference string is 1 2 3 4 5 4 2 3 4 5 1 2 3 4 5
- 3 frames available: total of 10 page faults

	1	2	3	4	5	4	2	3	4	5	1	2	3	4	5
Frame 1	1	1	1	4	4	4	4	4	4	4	4	2	2	4	4
Frame 2		2	2	2	2	2	2	3	3	3	3	3	3	3	5
Frame 3			3	3	5	5	5	5	5	5	1	1	1	1	1
pf	•	•	•	•	•			•			•	•		•	•

Least Recently Used (LRU)

- Track for each page time of *last* use; replace page not used for longest period of time
- Example: reference string is 1 2 3 4 5 4 2 3 4 5 1 2 3 4 5
- 3 frames available: total of 13 page faults

	1	2	3	4	5	4	2	3	4	5	1	2	3	4	5
Frame 1	1	1	1	4	4	4	4	4	4	4	4	2	2	2	5
Frame 2		2	2	2	5	5	5	3	3	3	1	1	1	4	4
Frame 3			3	3	3	3	2	2	2	5	5	5	3	3	3
pf	•	•	•	•	•		•	•		•	•	•	•	•	•

Least Recently Used (LRU)

- Example: reference string is 1 2 3 4 5 4 2 3 4 5 1 2 3 4 5
- 4 frames available: total of 10 page faults

	1	2	3	4	5	4	2	3	4	5	1	2	3	4	5
Frame 1	1	1	1	4	5	5	5	5	5	5	5	5	5	4	4
Frame 2		2	2	2	2	2	2	2	2	2	1	1	1	1	5
Frame 3			3	3	3	3	3	3	3	3	3	2	2	2	2
Frame 4				4	4	4	4	4	4	4	4	4	3	3	3
pf	•	•	•	•	•						•	•	•	•	•

Least Recently Used (LRU)

- Too expensive without hardware assistance
  - Stack or counters updated for each reference
- But that causes interrupts every page reference, increasing effective memory access time

Stack Algorithms

- One for which the set of pages in memory for n frames $M(n)$ is a subset of the set of pages in memory for n+1 frames $M(n+1)$
  - That is, $M(n) \subseteq M(n+1)$
  - Examples: OPT, LRU