Local versus Global Allocation

- When process A has a page fault, where does the new page frame come from?
- More precisely, is one of A's pages reclaimed, or can a page frame be taken from another process?
- If another process, do we bias the selection in any fashion?
- If page replacement affects only the current process, we have a *local* policy; if we look at all processes, we have a *global* allocation policy

Choosing

- Global policies tend to work better
- If you use a local policy and the working set grows, you can get thrashing
- Similarly, if the working set shrinks, you waste memory
- With a global policy, though, you need to decide how much memory to allocate to each process

Memory Allocation

- Fixed allocation — the same amount of space for all processes ($a = m/p$) — is too simplistic
- Better idea — allocate each process some memory in proportion to its size: $a_i = m \cdot m_i / \sum_{i=1}^{p} m_i$
- Do we want to use *size* or *working set*?
- What about process priority? $a_i = m \cdot (f_{\text{prio}}(m_i) / \sum_{i=1}^{p} m_i$

3 / 36

Memory Requirements Change

- Processes grow and shrink
- Working sets grow and shrink
- Allocations must be changed over time
- Monitor the *page fault frequency* (PFF) for each process
- A process with a high PFF gets a larger allocation; a process with a small PFF gets a smaller allocation

4 / 36

Measuring PFF

- Count the number of page faults per second
- Accumulate this as a moving average, of the type we've seen several times before
- For many algorithms, including LRU, PFF goes down as memory allocation increases

5 / 36

Algorithms versus Allocation

- Algorithms such as LRU and FIFO work with either local or global allocation policies
- Working set and WSclock are local-only
- There's no such thing as a working set for the entire system
- Must rely on allocation policy for global effects

6 / 36

Swapping

- As mentioned, the paging system has to interact with the scheduler
- Thrashing can be detected when the PFF rate of some processes has gone up, but none have gone down
- Must *swap* some processes to disk: write out all (or most) of their pages and reclaim their page frames

7 / 36

Controlling Swapping

- Which processes should get swapped out?
- Do we look at priority? Size? History?
- Once processes are swapped out, when do they come back in?
- Need a two-level scheduler, one for ordinary CPU access and one for swapping out and in
- For this second scheduler, what are we optimizing for? CPU utilization? Throughput?

8 / 36

Page Size

- With large pages, we waste memory: on average, half of the last page isn't used
- With small pages, we use a lot of memory for page tables
- Call the average process size s and the page size p. Assume that each page table entry (and associated data structures) takes e bytes
- The overhead o is $o = (s/p)e + p/2$
- To optimize for memory use, differentiate and set to 0: $do/dp = -se/p^2 + 1/2 = 0$
- Best size: $p = \sqrt{2se}$

Simulating Larger Pages

- Possible to treat several smaller pages as one larger page
- Still need separate hardware page table entries, but can reduce overhead elsewhere
- Big gain: fewer page faults
- Other gains: auxiliary data structures

Page Sharing and Remapping

- Context switch overhead can be reduced by page-sharing
- Example: shared memory in Unix (`shmat()`, `shmget()`, etc.) share memory between processes
- Caution: processes must use appropriate locks
- *Comm pages* allow processes to read (some) kernel data
- Example: `getpid()` can be a simple subroutine

Memory-Mapped I/O

- Instead of doing I/O, processes map a file onto a memory area, i.e., `mmap()`
- Easy random access
- Let the page algorithm handle the I/O
- On Multics, there was *no* disk I/O; all files were simply areas of memory
- Disadvantage: file size was limited by address space (actually, by segment size)

Page-Mapped I/O

- Suppose a user I/O buffer is page-sized and page-aligned
- Make sure that kernel disk buffers are page-sized and page aligned, too
- When the user process does a `read()`, change the page table so that the disk buffer is mapped to user space and the user's buffer becomes part of kernel memory
- No overhead for copying!
- Harder to do for `write()` — does the user process still want access to its data?
- Can sometimes "lend" pages, but mark them read-only

13 / 36

Don't Copy!

- Copying bulk data is very expensive
- Limited by memory bandwidth; could use a lot of cache space
- It's worth considerable effort to avoid handling data extra times

14 / 36

Allocating Swap Space

- Where does swap space come from?
- Some systems allocate swap space as soon as the application is given main memory
- In other words, *all* of the memory of every process has a reserved spot on disk
- Other systems allocate space as needed
- What if they run out?

15 / 36

Storage for Disk Mapping

- Where is the disk block address stored for a page?
- Some systems reuse the page table entry if the “valid” bit is off
- Works poorly if there’s a lot of swap space
- Doesn’t work if if you keep the disk images of pages in case they’re not dirty when reclaimed

16 / 36

Logical Segmentation

- Earlier, we talked about segments for VM
- There's another type: user-controlled segments
- Segments introduce non-linearity into the address space
- There's no carry into the segment bits when doing address arithmetic

Why Use Segments?

- Code, data, and the stack are each separate segments
- Shared libraries can each occupy a separate segment
- That way, only the segment pointer needs to be separate; the same page table can be used for each process using the library
- In Multics, each file was mapped to a particular segment

Protection and Segments

- Segments can have memory protection bits associated with them
- For the uses just described, this is more convenient and more natural than protecting each page independently

The Problems with Segments

- Maximum contiguous address space is limited by segment size
- For example, on the Intel 286, segments were limited to 64K; that meant that no array could be larger than 64K bytes
- Explicit segments are not often used today

Segments on the Pentium

- Six segment registers: code, data, four others
- 8K system and 8K user segments permitted
- A segment descriptor contains a base/limit pair and a pointer to memory
- That memory address may be virtual, in which case two levels of page table are used
- Three extra memory look-ups per memory reference!
- Good thing we have a TLB...

21 / 36

Memory Allocation

22 / 36

Types of Memory

- Kernel code — wired down (but some systems have used disk-resident system calls that are swapped in as needed)
- User code — paged in and out
- Page tables — must be dynamically allocated
- Stacks — also dynamically allocated
- Disk I/O buffers
- Network I/O buffers

22 / 36

Network I/O Buffers

- Allocation can be fixed
- If a user process writes too much, block
- If a remote process writes too much, use *flow control* to make it shut up
- If it doesn't listen, drop packets

23 / 36

Disk I/O Buffers

- How much memory should be allocated for disk I/O buffers?
- Simplest solution: some fixed percentage of memory
- Better solution: dynamically use memory for disk or for applications, as needed
- When system goes I/O-bound, leave fewer pages for applications
- When system is memory-bound, use fewer pages for disk I/O
- Sounds good, but getting the balance right is tricky

24 / 36

Kernel Memory Allocation

- Many kernel routines need to allocate memory dynamically
- Similar to `malloc()` for application programs
- These routines generally grab pages
- If there are no page frames free, the request can fail
- Often, there is a process context, and the process can block while waiting for a page
- Sometimes, the memory reclamation daemon is told to speed up

25 / 36

Managing Kernel Memory

- Kernel memory allocation is very similar to non-VM memory region allocation and `malloc()` allocation
- As before, see Knuth vol. 1 for details
- But — some systems will change the kernel's memory map to permit creation of large contiguous memory regions

26 / 36

A Theory of Paging?

- Can we figure out a theory of paging?
- Can we predict performance?
- Can we explain — or prevent — things like Belady's Anomaly?

Reference Strings

- A process can be characterized by an ordered list of the pages it accesses
- This is called the *reference string*
- A paging system can be described by three things: the reference string of the process, the page replacement algorithm, and the number of page frames available

An Abstract Model

- The process we're modeling has n pages
- There are m page frames
- Assume an n-element array M that keeps track of memory
- $M[n-m:n-1]$ represents in-memory pages
- $M[0:n-m-1]$ contains all other pages

29 / 36

Simulating the Process

- Take an entry from the reference string
- If the top half of M has room, put the page in it
- Otherwise, there's a page fault
- Apply the selected algorithm to move a page from the top of M to the bottom
- Move the new page to the top half
- Rearrange the top and bottom halves according to the algorithm

30 / 36

Simulating LRU

Reference string is 0 2 1 3 5 4 6 3 7 4 7 3 3 5 5.
Four page frames available.

0	2	1	3	5	4	6	3	7	4	7	3	3	5	5
0	2	1	3	5	4	6	3	7	4	7	3	3	5	5
	0	2	1	3	5	4	6	3	7	4	7	7	3	3
		0	2	1	3	5	4	6	3	3	4	4	7	7
			0	2	1	3	5	4	6	6	6	6	4	4
				0	2	1	1	5	5	5	5	5	6	6
					0	2	2	1	1	1	1	1	1	1
						0	0	2	2	2	2	2	2	2
								0	0	0	0	0	0	0
P	P	P	P	P	P	P		P					P	
∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1

Stack Algorithms

- If m varies over the possible page frames and r is an index into the reference strings, we may have
$$M(m, r) \subseteq M(m+1, r)$$
- That is, for a given initial sequence of a reference string, those pages that are at the top of M will still be in the top of M if there is one more page frame
- Algorithms that satisfy this property are called *stack algorithms*
- Belady's Anomaly cannot occur with stack algorithms

Is LRU a Stack Algorithm?

- Whenever a page is pushed below the line in LRU, it goes to the top of the bottom section
- If we move the boundary down by one page frame, we therefore include the previously-displaced page
- That means that the stack property holds — LRU is indeed a stack algorithm
- FIFO is not

Distance Strings

- Assume a stack algorithm
- A *distance string* d is a set of page references where the value is "distance from the top of the stack"
- An unreferenced page isn't on the stack and has distance ∞
- Distance strings are algorithm-dependent
- Small values are good; they indicate locality of reference
- You want most elements of d to be less than the number of page frames
- If d is mostly large numbers, you're out of luck

Predicting Page Fault Rates

- Scan the distance string and see how many times each value occurs
- Let C_i be the number of times i is found; C_∞ exists, too
- For our example, $\langle C_1, C_2, \ldots, C_7, C_\infty \rangle = \langle 4, 2, 1, 4, 2, 2, 1, 8 \rangle$
- If m is the number of page frames,

$$F_m = \sum_{k=m+1}^{n} C_k + C_\infty$$

- F_m is the number of page faults for that distance string and number of page frames

35 / 36

Origin of our Strings

- Where do reference and distance strings come from?
- As always, we can simulate them, but we're much better off getting real traces from real programs
- Note the implication: paging algorithm behavior can change if our workload changes

36 / 36