Week 7 (lec)

Wednesday, March 04, 2009
4:14 PM

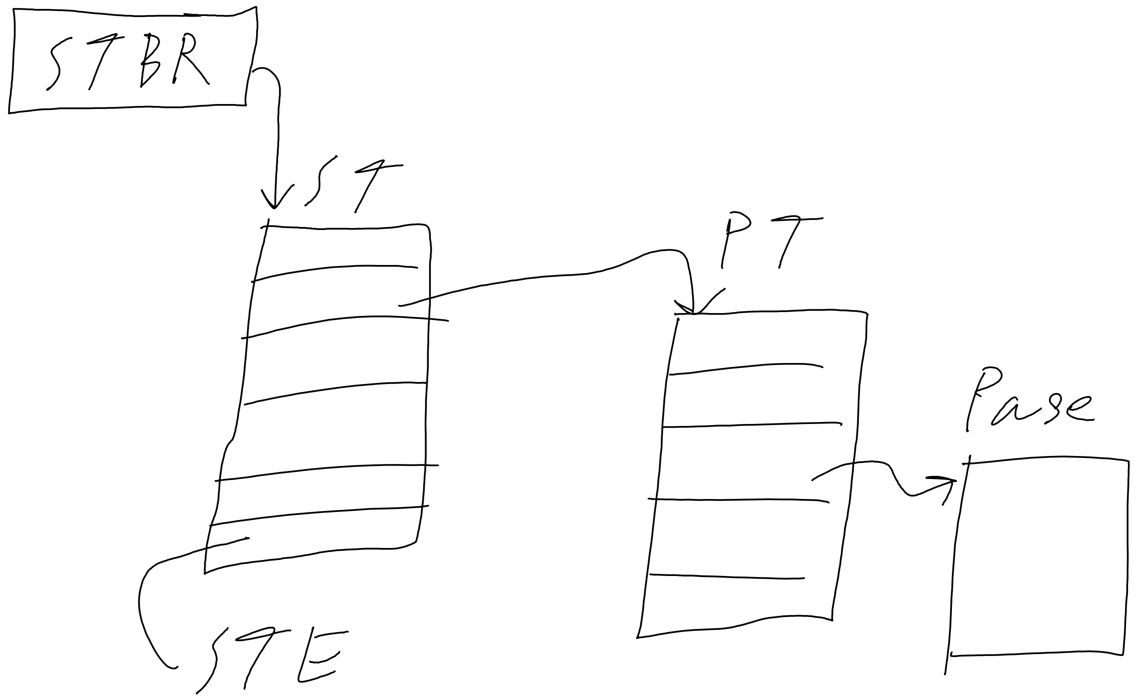
Midterm is tonight: 7pm, 306 Soda.

Page tables are either referred to with real addresses or OS virtual memory. It CANNOT be stored in the user's space, or user can bypass protection.

page table can be stored in OS virtual memory, and then paged out.

"wired down" means cannot be paged out. Example, you can't page out the page fault handler

Pages/segments provide protection because our process doesn't generate real addresses, and page table points only to those sections of memory we're allowed to access. This is a major advantage to using paging.



Combination of segments and pages. We put valid bit, dirty bit in page table. protection bits in segment table, or in page table, but usually in segment table since all pages share the same protection. Each segment has a bounds field to hold its size

Segments still correspond to logical units: code, data, separately linked functions…etc
Note: UNIX uses 3 segments, code, data, and stack

We don't need to do bounds checking when access segments, if we are willing to make the bounds a multiple of the page size. Because if we try to write past the last page we will trap because the valid bit is not on.

virtual addresses are now of the form: | segment number | page number | byte offset |
Note: This limits the amount of memory that you can address since the high order bits are used to specify the segment number. Example, if you have 20bit addressing and you use 4bits for segments then each segment is limited to 64K (2^16).

Main disadvantages:
-internal fragmentation within pages (not a big deal in modern computers)
-overhead to handle both paging and segmentation.

Pages vs. Segments: pages are for memory management, the user (programmer) is not aware of them. Segments are visible to user and are for logical units, they also have variable size unlike pages.

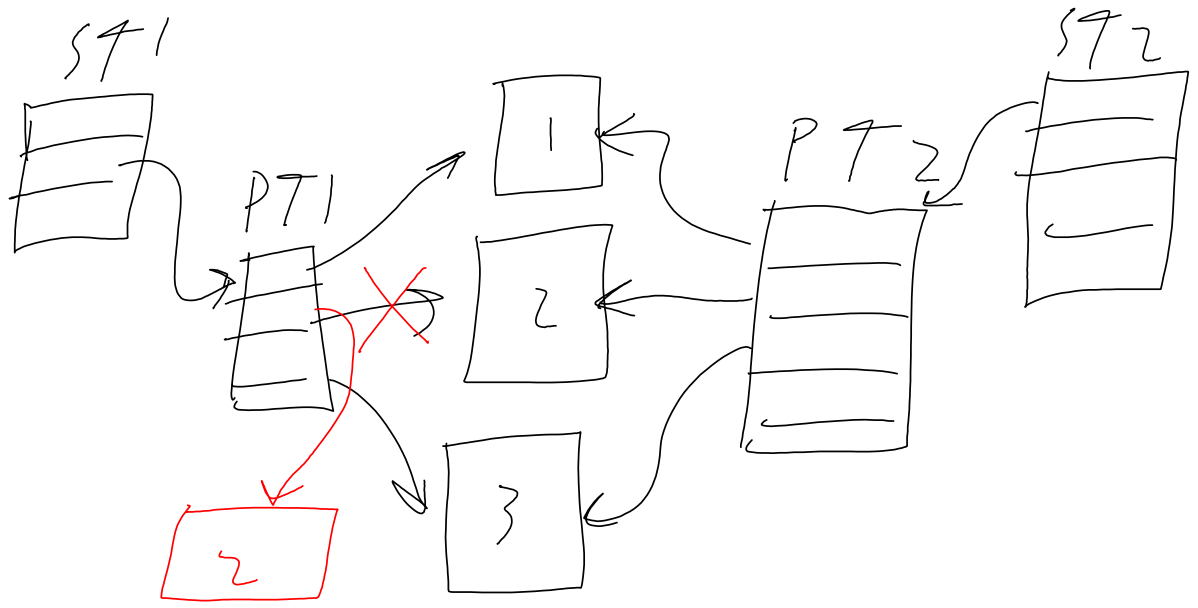
Question, David: so do all segments on a machine have to be the same size?
Answer: no, they are usually different sizes.

Sharing can be done at two levels: either share a whole segment or share a page.
However since programmer isn't aware of pages we usually don't share them, instead process one and two share a segment, that is one of their STE points to the same page table.
This is done to share things between separate processes, not different threads. Example, the shared segment might be a message buffer.

Note: the virtual address to the shared memory might not be the same for each process, since they could have different segment numbers. Therefore we can't have absolute (virtual) addresses in the shared segment, otherwise it would point to different things for the processes. We can have relative addresses, if they remain in the shared page. Or they can be offsets to registers.

Question, Adam: is this a problem when the PT is not shared?
Answer: no, we can have normal virtual addresses.

What if the data we're sharing is huge?
We use a trick called "copy on write". what we do is to save our own copy only when we make a modification to the shared data.



If process one makes a modification to page 2, then we make a copy and change where his page table points, so that it points to the copy of page 2.

To make this work set the shared pages to not be read-only, then we will trap when we try to write. and in this handler it will make us a new copy.

SVC stands for SuperVisor Call

problem: user makes an SVC call, passing a virtual address in a register that points to a buffer. user wants OS to write something to this buffer. The OS has to translate this since it can't use the virtual address to do a look up in its own page table.

solution 1: translate to a physical address using the user's page table. however the buffer might not be contiguous in physical memory, even if it was in virtual address space.

solution 2: specify that we should use a different page table. this requires that the machine provides a User PTBR and a System PTBR.

solution 3: map the user's pages into the OS page table. now that the pages are essentially shared the OS can use its address to do the look up. This is probably the best solution.

to speed up virtual to real address translation, we use TLB (should have been covered in 61C).

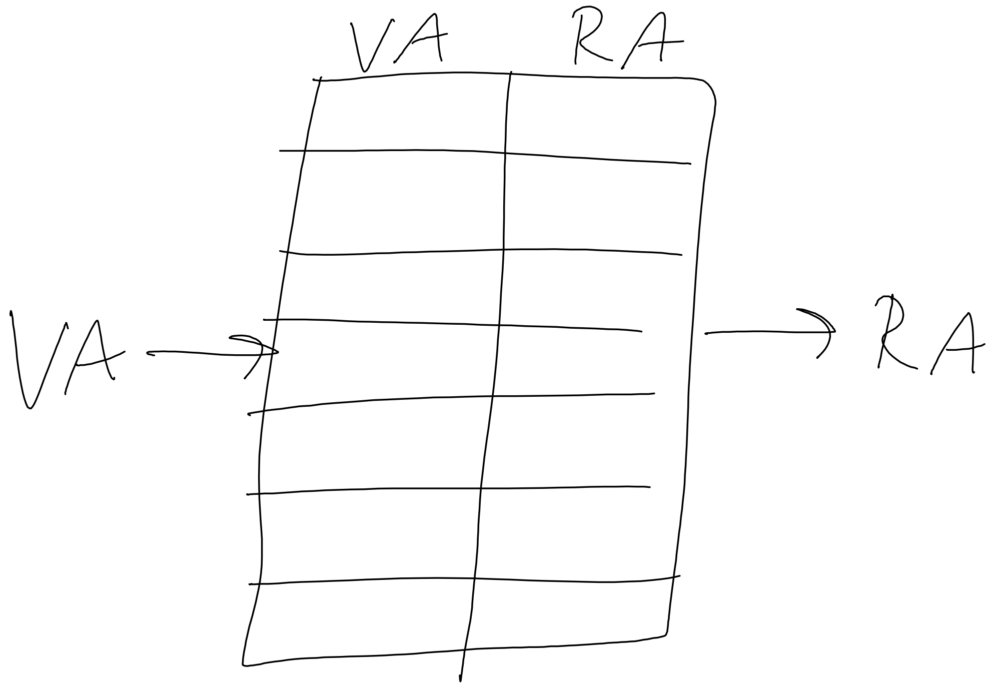
TLB (translation lookaside buffer) is the same thing as "address translation cache" (ATC), directory lookaside table (DLAT), translation buffer (TB). Address translation cache is probably the best name, since the TLB is just a cache of recently translated addresses.

It would seem that the TLB does not work well because there might be lots and lots of pages in a large program. However because of the Principle of Locality it works very well.
Principle of Locality says that things used recently are likely to be used again, and data that is near the last address accessed is likely to be used soon.
reason: programs are sequential or have loops.

In practice TLBs work very well, often > 99% effective.

Question, Matt: Do multiple processes share the TLB?
Answer: Yes, there is usually just one TLB.

Architecture of TLBs



TLBs are usually full-associative, meaning the look can be done in constant time (every entry compared to the key in parallel). However associative memory is expensive, and cannot built very big (TLBs limited to ~64 entries).

To solve this make TLB into 4 sets. then each virtual address has 2 bits reserved to specify which set.