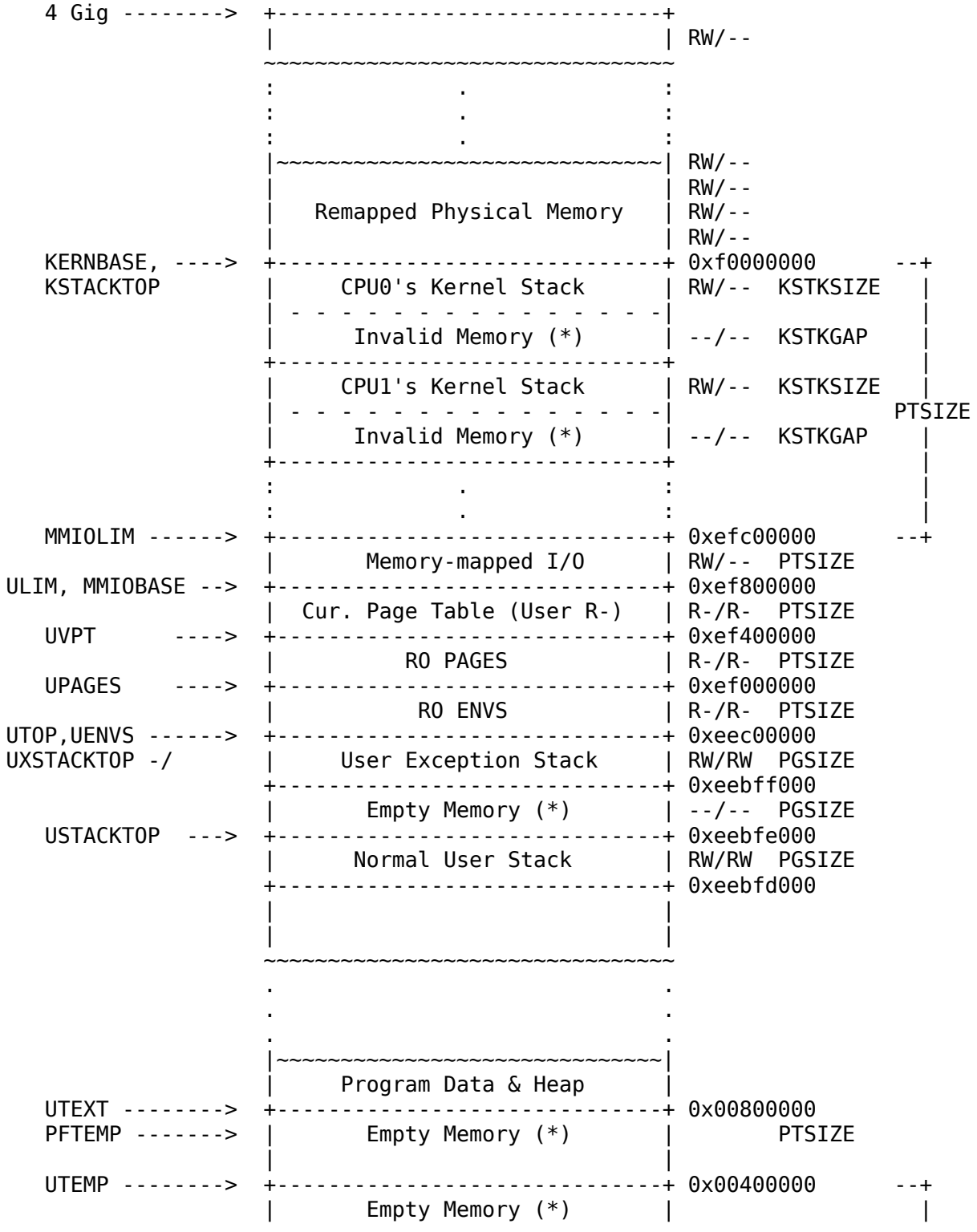
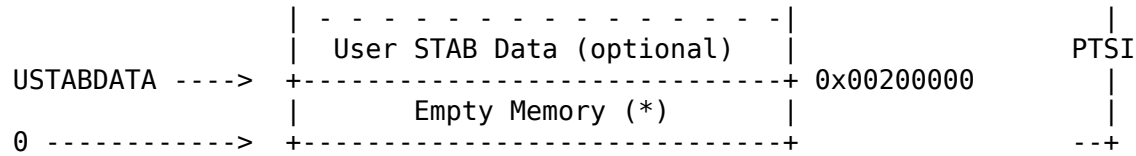
How we will use paging (and segments) in JOS:

- use segments only to switch privilege level into/out of kernel
- use paging to structure process address space
- use paging to limit process memory access to its own address space
- below is the JOS virtual memory map
- why map both kernel and current process? why not 4GB for each? how does this compare with xv6?
- why is the kernel at the top?
- why map all of phys mem at the top? i.e. why multiple mappings?
- (will discuss UVPT in a moment...)
- how do we switch mappings for a different process?





The UVPT

We had a nice conceptual model of the page table as a 2^20-entry array that we could index with a physical page number. The x86 2-level paging scheme broke that, by fragmenting the giant page table into many page tables and one page directory. We'd like to get the giant conceptual page-table back in some way -- processes in JOS are going to look at it to figure out what's going on in their address space. But how?

Luckily, the paging hardware is great for precisely this -- putting together a set of fragmented pages into a contiguous address space. And it turns out we already have a table with pointers to all of our fragmented page tables: it's the page directory!

So, we can use the page *directory* as a page *table* to map our conceptual giant 2^22-byte page table (represented by 1024 pages) at some contiguous 2^22-byte range in the virtual address space. And we can ensure user processes can't modify their page tables by marking the PDE entry as read-only.

Puzzle: do we need to create a separate UVPD mapping too?

A more detailed way of understanding this configuration:

Remember how the X86 translates virtual addresses into physical ones:

CR3 points at the page directory. The PDX part of the address indexes into the page directory to give you a page table. The PTX part indexes into the page table to give you a page, and then you add the low bits in.

But the processor has no concept of page directories, page tables, and pages being anything other than plain memory. So there's nothing that says a particular page in memory can't serve as two or three of these at once. The processor just follows pointers: pd = lcr3(); pt = *(pd+4*PDX); page = *(pt+4*PTX);

Diagramatically, it starts at CR3, follows three arrows, and then stops.

If we put a pointer into the page directory that points back to itself at index V, as in

then when we try to translate a virtual address with PDX and PTX equal to V, following three arrows leaves us at the page directory. So that virtual page translates to the page holding the page directory. In Jos, V is 0x3BD, so the virtual address of the UVPD is (0x3BD<<22)|(0x3BD<<12).

Now, if we try to translate a virtual address with PDX = V but an arbitrary PTX != V, then following three arrows from CR3 ends one level up from usual (instead of two as in the last case), which is to say in the page tables. So the set of virtual pages with PDX=V form a 4MB region whose page contents, as far as the processor is concerned, are the page tables themselves. In Jos, V is 0x3BD so the virtual address of

the UVPT is (0x3BD<<22).

So because of the "no-op" arrow we've cleverly inserted into the page directory, we've mapped the pages being used as the page directory and page table (which are normally virtually invisible) into the virtual address space.