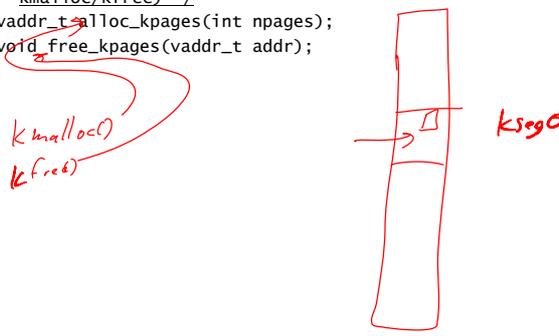


## Assignment 2 Parts 3

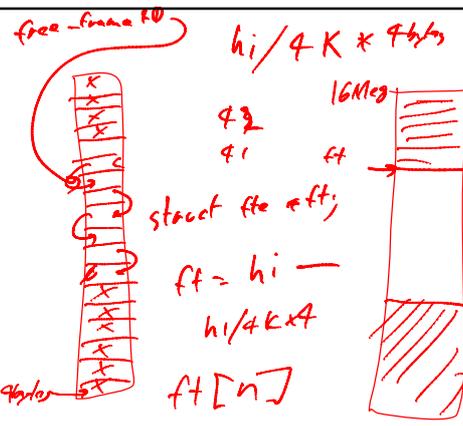
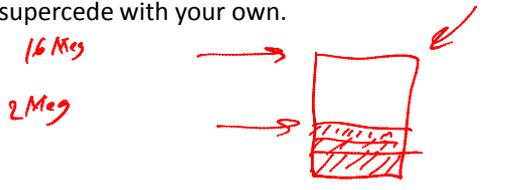
- Memory Management
- Address Space Management and TLB Refill

```
void vm_bootstrap(void);
```

```
/* Allocate/free kernel heap pages (called by
kmalloc/kfree) */
vaddr_t alloc_kpages(int npages);
void free_kpages(vaddr_t addr);
```



- How do I figure out how big memory is?
- ram\_getsize() will return the current top of used memory, and the size of physical memory configured for sys161. See kern/arch/mips/vm/ram.c to understand the basic allocator that you need to mostly supercede with your own.



```
struct fte *ft = 0
```

```
alloc_kpage()
if (ft == 0) {
    ram_steal_mem();
} else {
    // 1x your code //
}
```

### • Where can I put my frame table

- You frame table should be dynamically sizeable based on physical memory in the machine. My suggestion is that you compute somewhere in RAM to put it and just use it in that location.
  - Here is a little code to illustrate what I mean
- ```
struct frame_table_entry *frame_table;
location = top_of_ram - (top_of_ram / 4K * 4);
frame_table = (struct frame_table_entry *) location;
```
- Note that you will have to mark entries the in table as used for both the table itself, and os/161 allocated to this point in time.

- OS/161 allocated more than a page using `alloc_kpages`
- Yes, the sample implementation of `execve` does this - you don't need to support it. To avoid this (and enable you to run more testing programs), set `__ARG_MAX` to 4096 in `kern/include/kern/limits.h`.

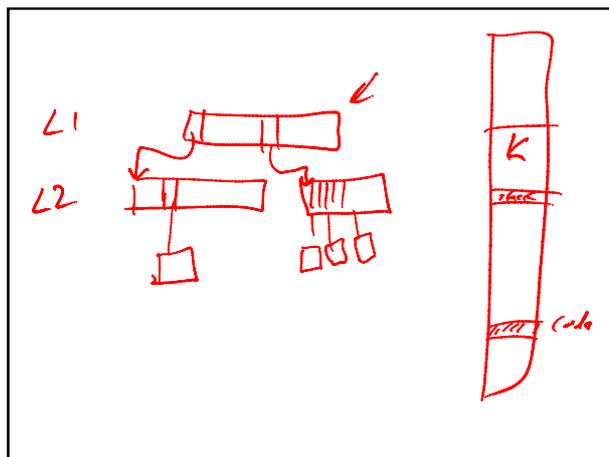
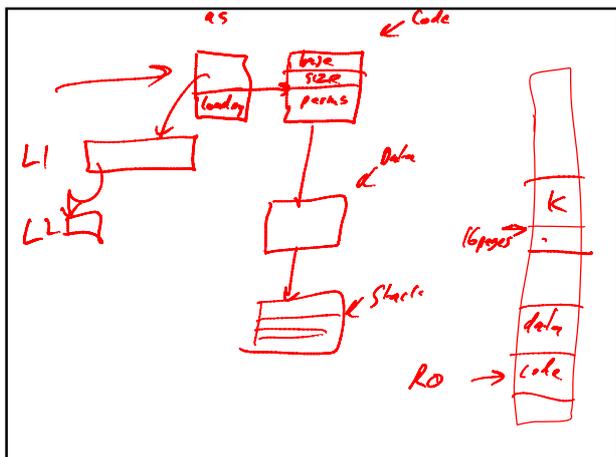
```

struct addrspace *as_create(void);
int as_copy(struct addrspace *src, struct addrspace **ret);
void as_activate(struct addrspace *);
void as_destroy(struct addrspace *);

int as_define_region(struct addrspace *as, vaddr_t vaddr, size_t sz, int readable, int writeable, int executable);

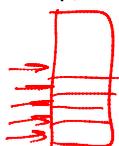
int as_prepare_load(struct addrspace *as);
int as_complete_load(struct addrspace *as);
int as_define_stack(struct addrspace *as, vaddr_t *initstackptr);

/* Fault handling function called by trap code */
int vm_fault(int faulttype, vaddr_t faultaddress);
    
```



*trace 161*

- Don't use `kprintf` style debugging in the TLB refill routine after TLB write
- `kprintf` causes a context switch (it blocks) which flushes the TLB, which potentially ejects the entry you have just loaded - infinite loop, here we come.



- How do I allocate page tables?
- `alloc_kpage()` returns a single page that happens to be the correct size for a two-level page table with 4-byte entries. Note: You need to use 4 bytes entries to avoid undue complexity in allocation, [EntryLo](#) and pointers happen to be 4 bytes in size.

- How can my my allocator work before and after it is initialised?

- Try something along the lines of:

```

struct frame_table_entry *ft = 0;
alloc_kpages()
{
    if (ft == 0) {
        /* use ram_stealmem */
    }
    else {
        /* use my allocator as frame table is now
        initialises */
    }
}

```

```

struct L1e {
    uint32_t *L2pagetable;
};

void func(vaddr_t v)
{
    struct L1e *L1;
    uint32_t *L2;
    uint32_t pte;
    unsigned int v1, v2;

    v1 = v >> 22;
    v2 = v << 10 >> 22;

    L2 = L1[v1].L2pagetable;
    if (L2 == NULL) {
        panic();
    }
    pte = L2[v2];
}

```

