COS 318 - Operating System

Assignment 4

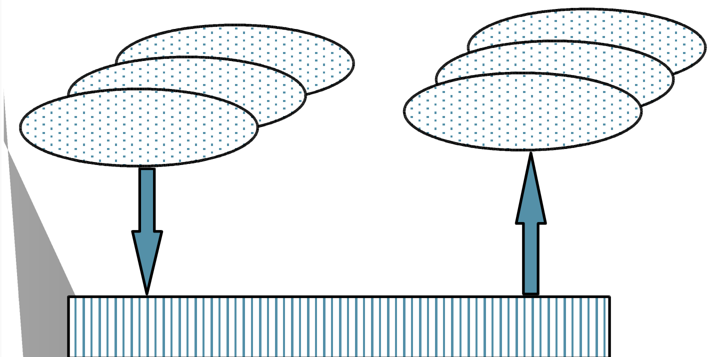
Inter-Process Communication and Process management

Fall 2004

Main tasks

- Inter-Process Communication
  - Implement Mailboxes
  - Keyboard Input
- Minimizing interrupt disabling
- Process Management
  - Be able to load a program from disk
  - Extra credit options

Mailbox - Bounded Buffer



- Buffer
  - Has fixed size
  - Is a FIFO
  - Variable size message
- Multiple producers
  - Put data into the buffer
- Multiple consumers
  - Remove data from the buffer
- Blocking operations
  - Sender blocks if not enough space
  - Receiver blocks if no message

Mailbox - Implementation

- Buffer management
  – Circular buffer: head and tail pointers
- Bounded buffer problem
  – Use locks and condition variables to solve this problem as shown in class
  – 2 condition variables: moreData and moreSpace
  – See mbox.h and mbox.c

Keyboard - Overview

- How the keyboard interacts with OS
  - An hardware interrupt (IRQ1) is generated when a key is pressed or released
  - Interrupt handler talks to the hardware and gets the scan code back.
  - If it is SHIFT/CTRL/ALT, some internal states are changed.
  - Otherwise the handler converts the scan code into an ASCII character depending on the states of SHIFT/CTRL/ALT.

Keyboard - Overview

- How the keyboard interacts with OS
  - An hardware interrupt (IRQ1) is generated when a key is pressed or released
  - init_idt() in kernel.c sets handler to irq1_entry in entry.S
  - irq1_entry calls keyboard_interrupt in keyboard.c

Keyboard - Overview

- keyboard_handler talks to the hardware and gets the scan code back.
  - key = inb(0x60);
  - Call key specific handler

Keyboard - Overview

- If it is SHIFT/CTRL/ALT, some internal states are changed.
- Otherwise normal_handler converts the scan code into an ASCII character.
- normal_handler calls putchar() to add character to keyboard buffer
- You need to implement putchar()
- Also getchar() which is called by the shell

Keyboard - Implementation

- It’s a bounded buffer problem
  - So, use mailbox.
- But, there are some variations
  - Single producer (IRQ1 handler)
  - Multiple consumers (more than one processes could use keyboard)
  - Producer can't block - discard character if buffer is full.

Keyboard - Subtle points

- Producer shouldn't be blocked
  - Solution: check and send message only if mailbox is not full, otherwise discard it.
  - Make use of mbox_stat() function
- Is that all ?
  - What if a process being interrupted by IRQ1 is currently calling getchar()?
  - Address how to fix this issue in design review

Reducing interrupt disabling

- Disable interrupt only when necessary.
- Motivation
  - Otherwise, could lose hardware events
    - For instance, keyboard or timer events
- Where to reduce
  - Very little we can do with scheduler.c
    - Switching stacks, manipulating ready queue
  - Thread.c
    - Locks, condition variables

Reducing interrupt disabling

- Alternative to interrupt disabling
  - Use spinlock to guarantee atomicity

```
spinlock_acquire( int *l) { while (  !TAS(l)); }
spinlock_release( int *l) { *l = 0; }
```

see thread.c

- One spinlock per lock/condition variable

```
typedef struct {
  int  spinlock;
  struct pcb *waiting;
  int status;
} lock;
```

see thread.h

Using spinlock - An example

- Code from project 3

```
void lock_acquire (lock_t *l){
    CRITICAL_SECTION_BEGIN;
    if (l->status == UNLOCKED) {
        l->status = LOCKED;
    } else {
        block(&l->waiting);
    }
    CRITICAL_SECTION_END;
}
```

- Using spinlock

```
void lock_acquire(lock_t *l) {
    use spinlocks to achieve same thing
    (part of design review)
}
```

- NOTE: block now takes any extra argument - spinlock
  - the spinlock is released in block()

Process Management

- So far, we only handle processes booted along with the OS.
- To support dynamic loading, we must have the followings:
  - Separate address space for each process
  - A simple file system format describing how processes reside on a disk
  - A memory manager
- Read shell.c to find out the commands it supports

Separate Address Space

- Each process has its own CS and DS segment selector, and the program always starts at address 0x1000000 (16MB mark).
- Basic paging only -- no paging to disk yet.
- This is done for you

Paging in x86

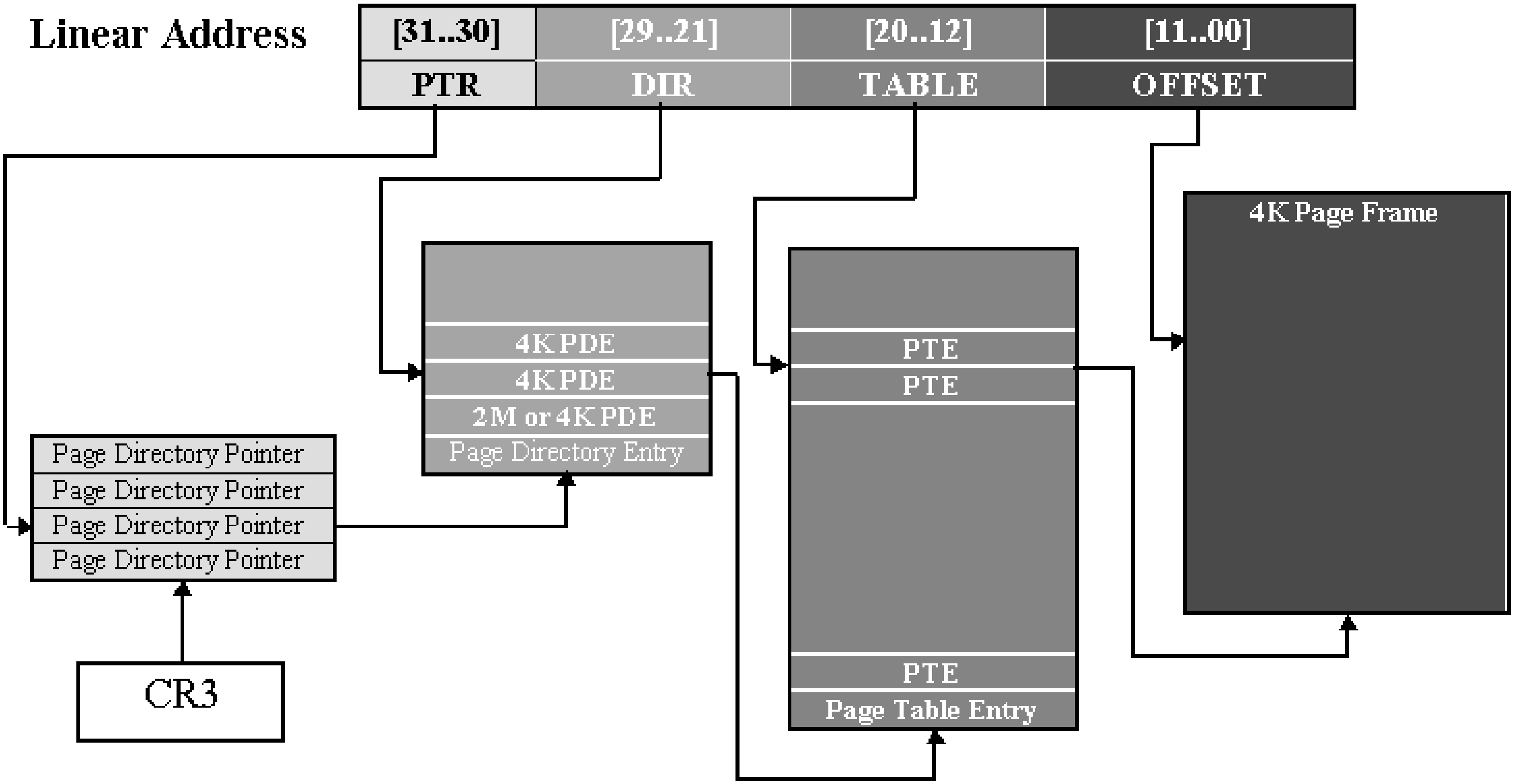
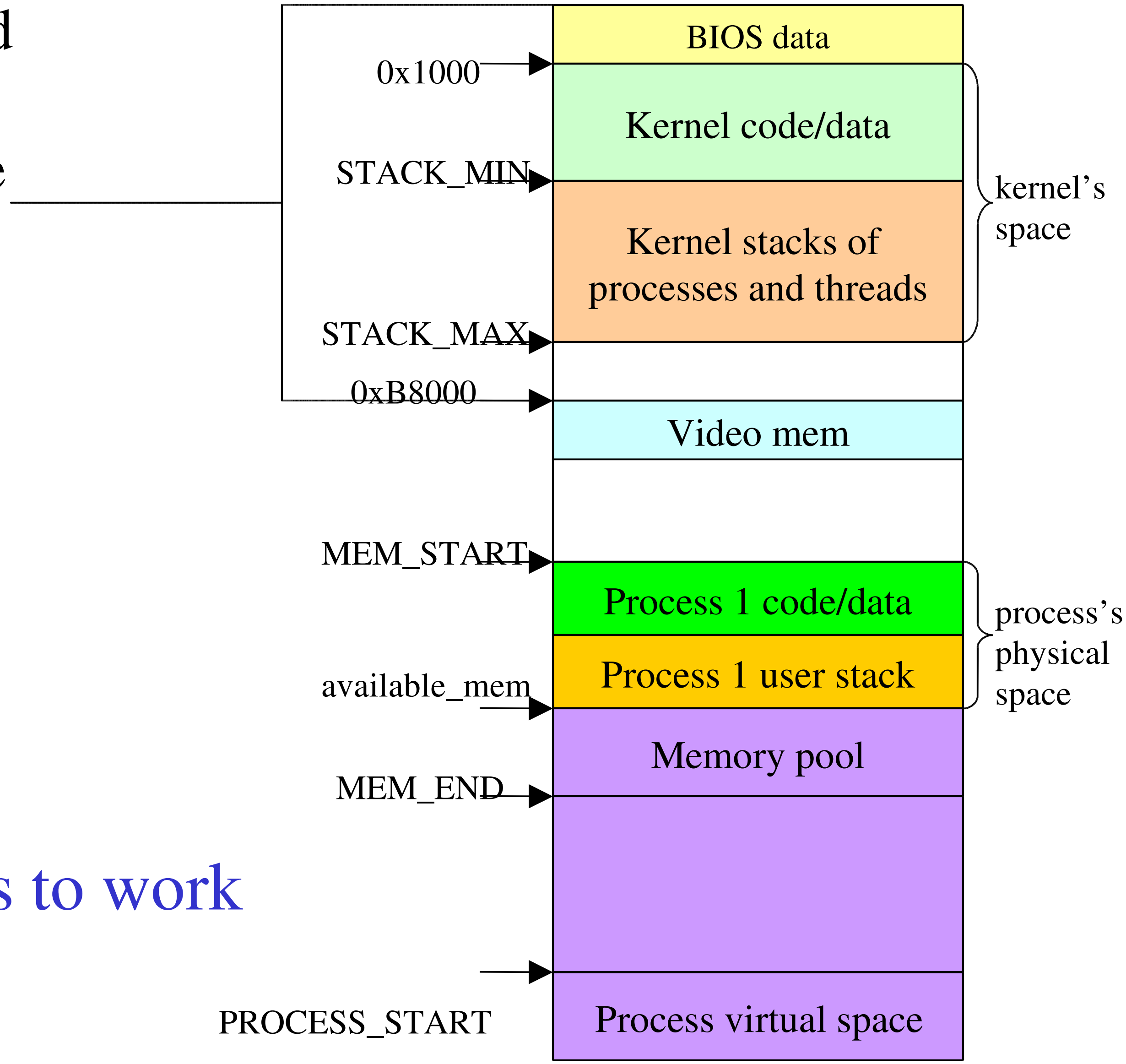


Image courtesy x86.org. (Intel manual vol. 3 for more information)

Paging – shared mapping

identity-mapped memory with kernel privilege access

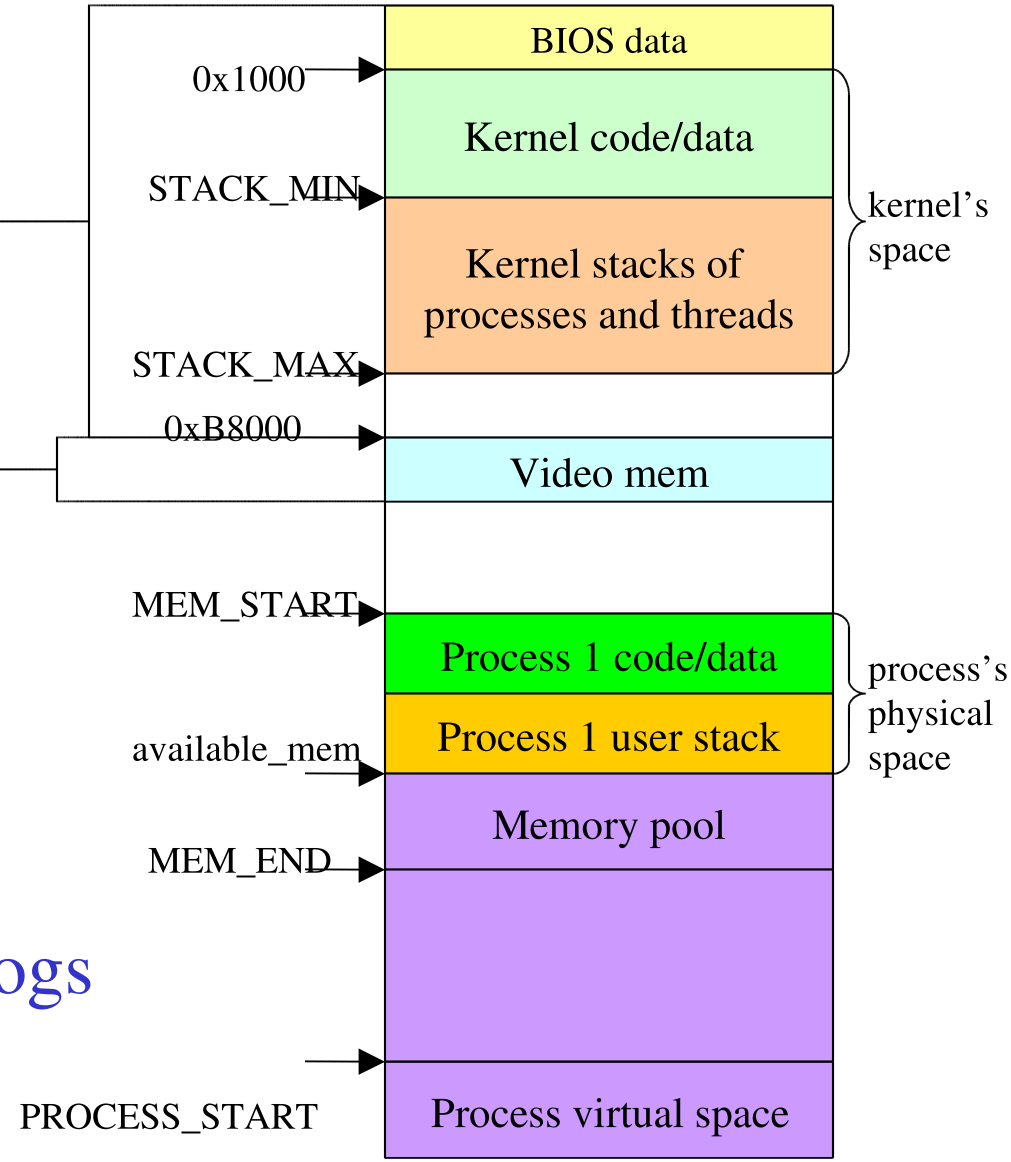


Needed for interrupts to work

Paging – shared mapping

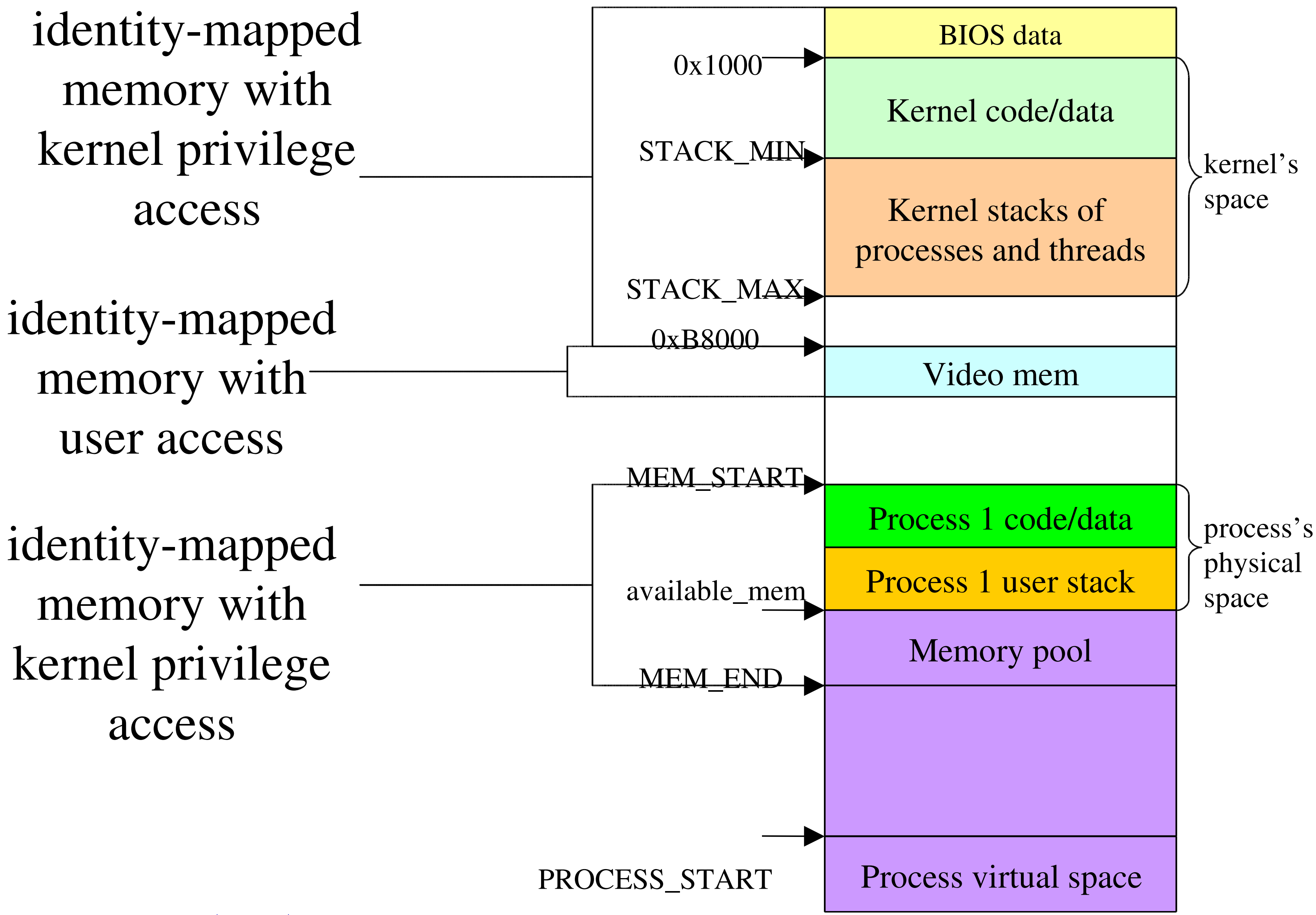
identity-mapped memory with kernel privilege access

identity-mapped memory with user access



Direct write to screen for progs

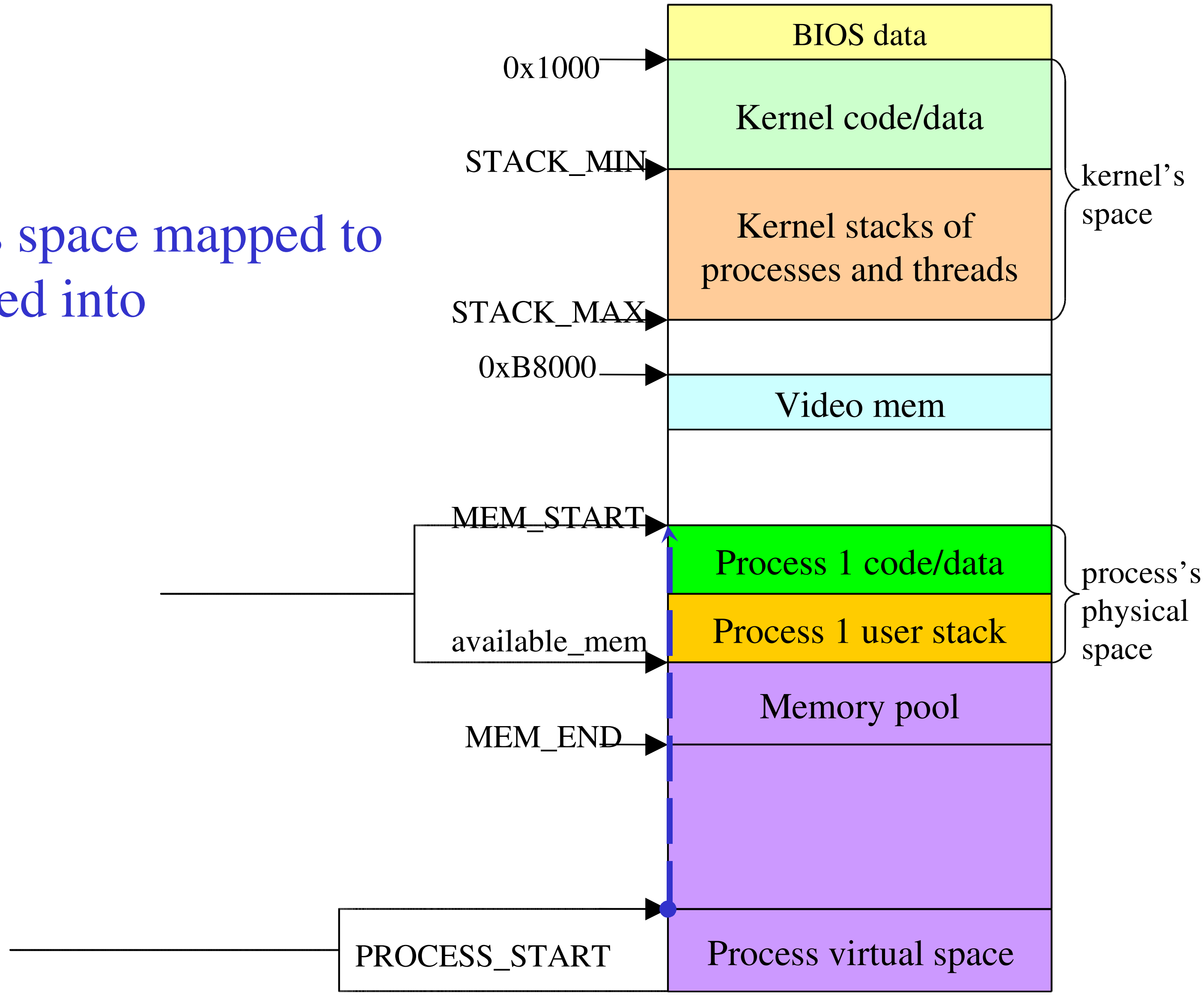
Paging – shared mapping



For mem mgmt. etc

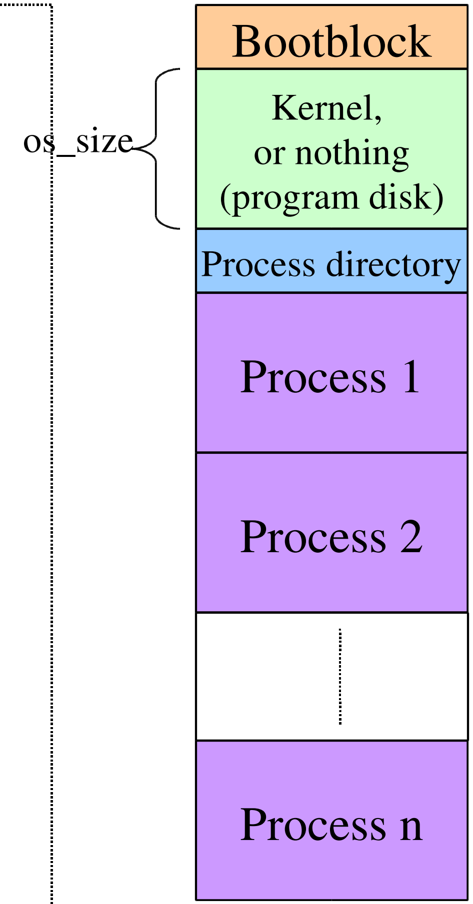
Paging – per process mapping

Process 1 address space mapped to location it is loaded into



Simple File System

- A bootblock followed by 0 or 1 kernel image.
- A process directory, the i^{th} entry records the offset and length of process i.
- Bootblock only loads kernel. Kernel loads the shell only initially.
- Some calculation involved in locating process directory

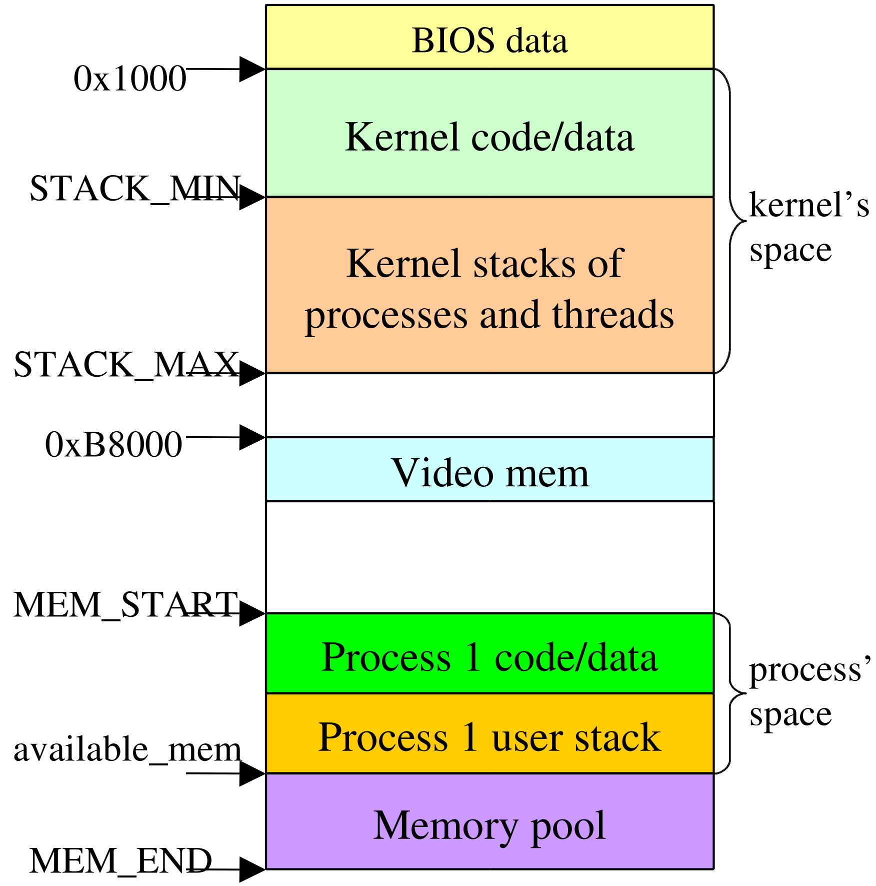


Memory Manager (memory.c)

- alloc_memory() allocates a requested size of memory from the available memory pool.
- free_memory() frees a memory block, (it does nothing right now.)
- Extra credit:
  - Do a better job here, actually free a block, and implement some sophisticated algorithm such as Best Fit.

Runtime memory layout

- The user stack is allocated in process's own address space
- Kernel stack is allocated in kernel's address space



Loading a program

- load <process#> shell command loads a process.
- process# is number reported by “ls” command of shell.
- Process# simply assigned incrementaly starting from 0 – this is inside shell and not something the fs supports.
- Uses readdir and process# to determine location of process
- Use loadproc to load process

Syscall readdir

- Locate process directory location.
- Read the sector containing the process directory.

Syscall loadproc

- Allocate a memory big enough to hold the code/data plus 4K stack
- Read process image from disk
- Allocate and initialize a PCB including:
  - Allocate new CS/DS selectors
  - Allocate user/kernel stack
  - Insert it to the ready queue
  - create_process does this part for you.

Floppy interface

- File: floppy.c
- You will only need to use 3 functions
  - floppy_start_motor: get lock and start motor
  - floppy_read: read a block into memory
  - floppy_stop_motor: stop motor and release lock
- floppy_write: next assignment.
- Watch for update to go back to usb.

Extra credit

- Memory deallocation after process termination
- Better memory management
- ps command in shell
- kill command in shell
- Note: shell is a process, so don’t call anything in the kernel directly.

Notes

- Process 0 is the shell itself. Do not reload it.
- You have to write about 350 lines of code total
- Read the provided code to see how it has changed and what new things have been added – it's a good way to learn.
- Process 3 and 4 to test mbox