Operating Systems
Project 5:
Processes & Multiprogramming

New Files

- New versions of:
  - `kernel.asm`
  - `lib.asm`
  - `bootload.asm`
  - `map.img`
  - `dir.img`
- New files:
  - `proc.h`
  - `testproc.h`

Multiprogramming Requirements

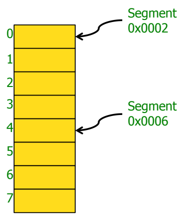
- Memory Management
  - Ability to load multiple programs into memory
- Time Sharing
  - Ability to periodically stop the running process and transfer control to an ISR in the OS.
- Process Management
  - Ability to keep track of and change between executing processes.
    - Context switching
    - Ready queue

Segment-Based Memory Management

- Allow one process to be loaded into each segment
  - Segments: 0x0000, 0x1000, 0x2000, … 0x9000
    - 0x0000 reserved for interrupt vector
    - 0x1000 reserved for kernel
    - 8 segments for user programs
      - 0x2000 – 0x9000
- Maximum program + data + stack?
  - 0x1000 bytes = 65536 bytes = 64kB

Tracking Free Memory

- Memory segment map:
  - Each index corresponds to one memory segment.
    - segment = (index+2)*0x1000
    - index = (segment/0x1000)-2
  - Marked as:
    - FREE
    - USED



Time Sharing:
Programmable Interrupt Timer

- Generates interrupt 0x08
- Will generate approximately 12 interrupts / second
- ISR for interrupt 0x08 will do context switching and scheduling
  - Assembly language code is given
  - Write a C function that gets called on each interrupt
    - Similar to `handleInterrupt21` for interrupt 0x21.

Interrupt 0x08 ISR Details

- `makeTimerInterrupt()`
  - Sets entry 0x08 in interrupt vector to point to timer_ISR assembly routine in kernel.asm.
- `timer_ISR()`
  - Pushes context (GP registers + PC) onto stack of interrupted program.
  - Invokes handleTimerInterrupt with segment & stack pointer of interrupted program
- `handleTimerInterrupt(int segment, int stackPointer)`
  - C function that you add to your *kernel*
  - Does process management and short term scheduling
- `returnFromTimer(int segment, int stackpointer)`
  - Assembly routine in kernel.asm
  - Called at end of handleTimerInterrupt to return to program in segment.
    - Pops context of program from its stack
    - Transfers control to program in segment
  - Call does not return.

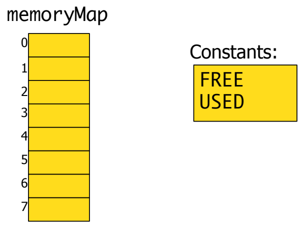
Process Management Responsibilities

- Starting a new process (`executeProgram`)
  - Obtain process control block (PCB) for the process
  - Load program into free segment
  - Put PCB into ready queue
- Short-term scheduling (`handleTimerInterrupt`)
  - Save stack pointer of interrupted process in PCB
  - Pick new process from ready queue
  - Start new process by calling `returnFromTimer`
- Terminating a processes (`terminate`)
  - Release memory segment
  - Release PCB

proc.h

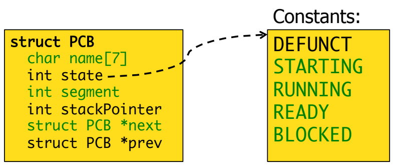
- `proc.h` defines a constants, data structures, global variables. and functions that you will use for memory and process management.
- `proc.h` is given
- You need to write `proc.c` to implement the defined functions.

`proc.h` Data Structures



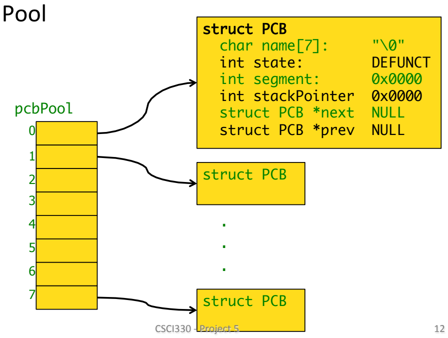
`proc.h` Data Structures

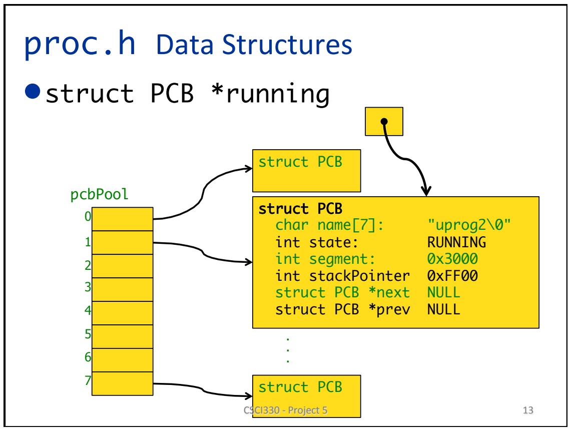
- Process Control Block:



`proc.h` Data Structures

- PCB Pool

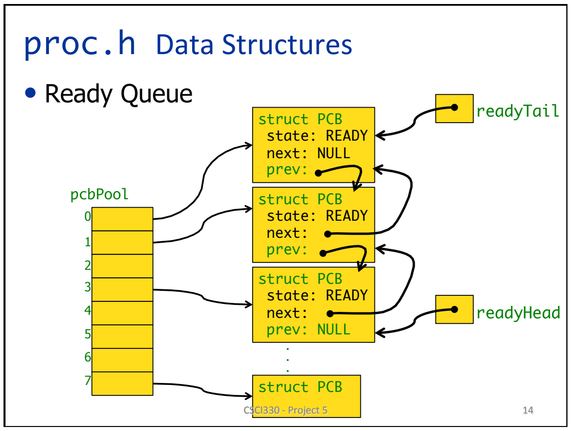




proc.h Data Structures

- struct PCB *running

```
struct PCB
   char name[7]:       "uprog2\0"
   int state:          RUNNING
   int segment:        0x3000
   int stackPointer    0xFF00
   struct PCB *next    NULL
   struct PCB *prev    NULL
```



proc.h Data Structures

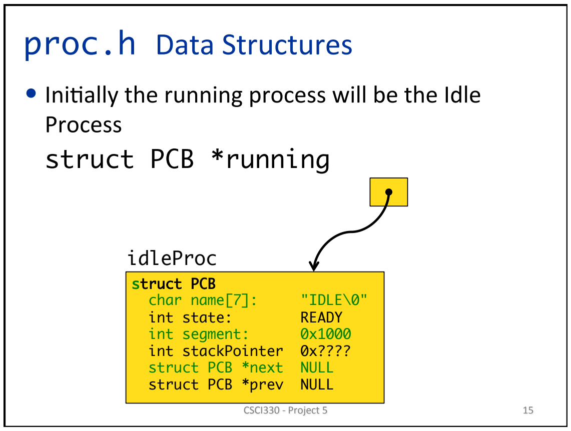
- Ready Queue

```
struct PCB
  state: READY
  next: NULL
  prev:

struct PCB
  state: READY
  next:
  prev:

struct PCB
  state: READY
  next:
  prev: NULL

struct PCB
```

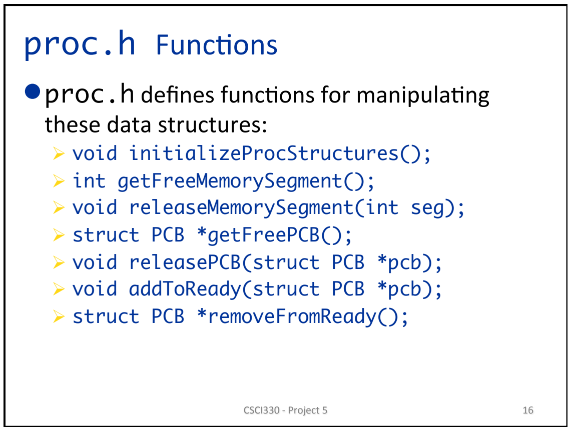


proc.h Data Structures

- Initially the running process will be the Idle Process

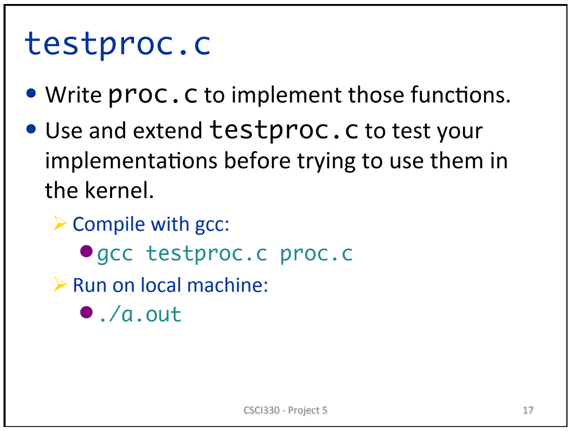
  struct PCB *running

```
struct PCB
   char name[7]:      "IDLE\0"
   int state:         READY
   int segment:       0x1000
   int stackPointer   0x????
   struct PCB *next   NULL
   struct PCB *prev   NULL
```



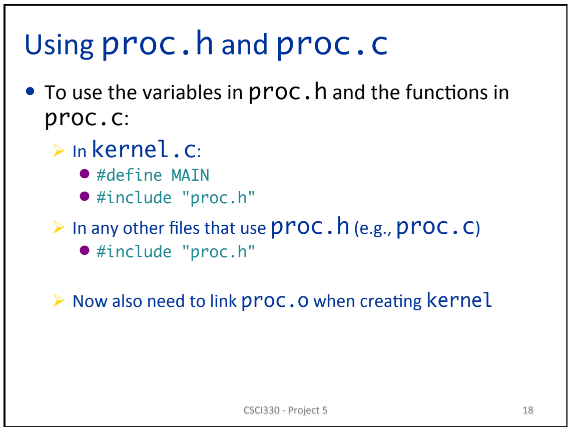
proc.h Functions

- proc.h defines functions for manipulating these data structures:
  - void initializeProcStructures();
  - int getFreeMemorySegment();
  - void releaseMemorySegment(int seg);
  - struct PCB *getFreePCB();
  - void releasePCB(struct PCB *pcb);
  - void addToReady(struct PCB *pcb);
  - struct PCB *removeFromReady();



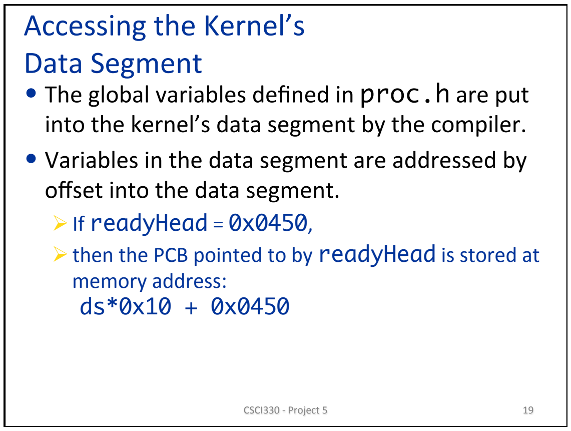
testproc.c

- Write proc.c to implement those functions.
- Use and extend testproc.c to test your implementations before trying to use them in the kernel.
  - Compile with gcc:
    - gcc testproc.c proc.c
  - Run on local machine:
    - ./a.out



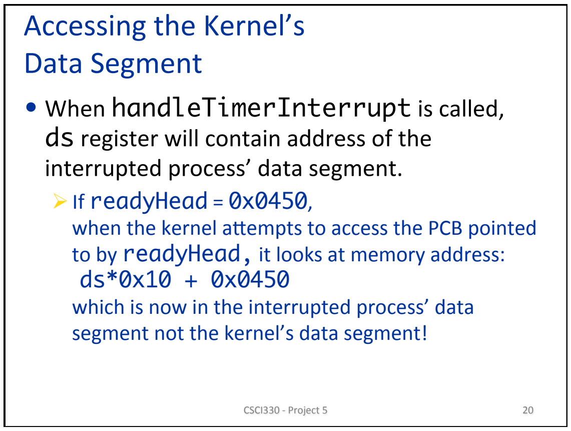
Using proc.h and proc.c

- To use the variables in proc.h and the functions in proc.c:
  - In kernel.c:
    - #define MAIN
    - #include "proc.h"
  - In any other files that use proc.h (e.g., proc.c)
    - #include "proc.h"
  - Now also need to link proc.o when creating kernel



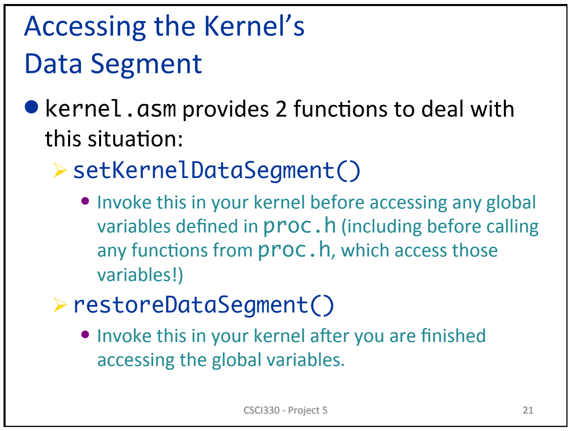
Accessing the Kernel’s Data Segment

- The global variables defined in `proc.h` are put into the kernel’s data segment by the compiler.
- Variables in the data segment are addressed by offset into the data segment.
  - ➢ If `readyHead = 0x0450`,
  - ➢ then the PCB pointed to by `readyHead` is stored at memory address: `ds*0x10 + 0x0450`



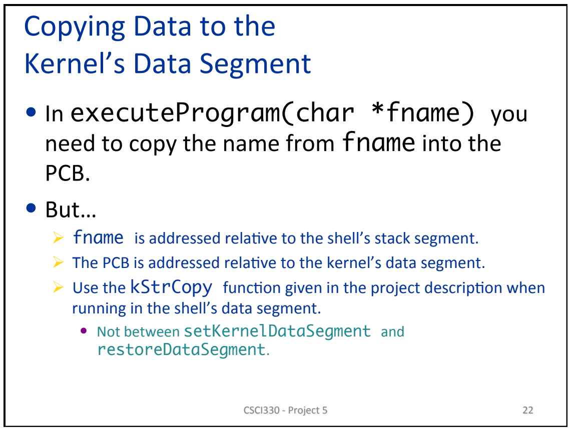
Accessing the Kernel’s Data Segment

- When `handleTimerInterrupt` is called, `ds` register will contain address of the interrupted process’ data segment.
  - ➢ If `readyHead = 0x0450`, when the kernel attempts to access the PCB pointed to by `readyHead`, it looks at memory address: `ds*0x10 + 0x0450` which is now in the interrupted process’ data segment not the kernel’s data segment!



Accessing the Kernel’s Data Segment

- `kernel.asm` provides 2 functions to deal with this situation:
  - ➢ `setKernelDataSegment()`
    - Invoke this in your kernel before accessing any global variables defined in `proc.h` (including before calling any functions from `proc.h`, which access those variables!)
  - ➢ `restoreDataSegment()`
    - Invoke this in your kernel after you are finished accessing the global variables.



Copying Data to the Kernel’s Data Segment

- In `executeProgram(char *fname)` you need to copy the name from `fname` into the PCB.
- But…
  - ➢ `fname` is addressed relative to the shell’s stack segment.
  - ➢ The PCB is addressed relative to the kernel’s data segment.
  - ➢ Use the `kStrCopy` function given in the project description when running in the shell’s data segment.
    - Not between `setKernelDataSegment` and `restoreDataSegment`.