# Pintos Lab1 Overview

TA: Zhong Yinmin

# Pintos or Tacos?

- 10 students submitted design doc for Tacos Lab0
  - 1 student submitted an empty file
  - 2 🤪 submitted their Pintos design docs
  - 7 warriors chose Tacos !!!


群聊：24春Tacos讨论群

该二维码7天内(3月11日前)有效，重新进入将更新

# Today

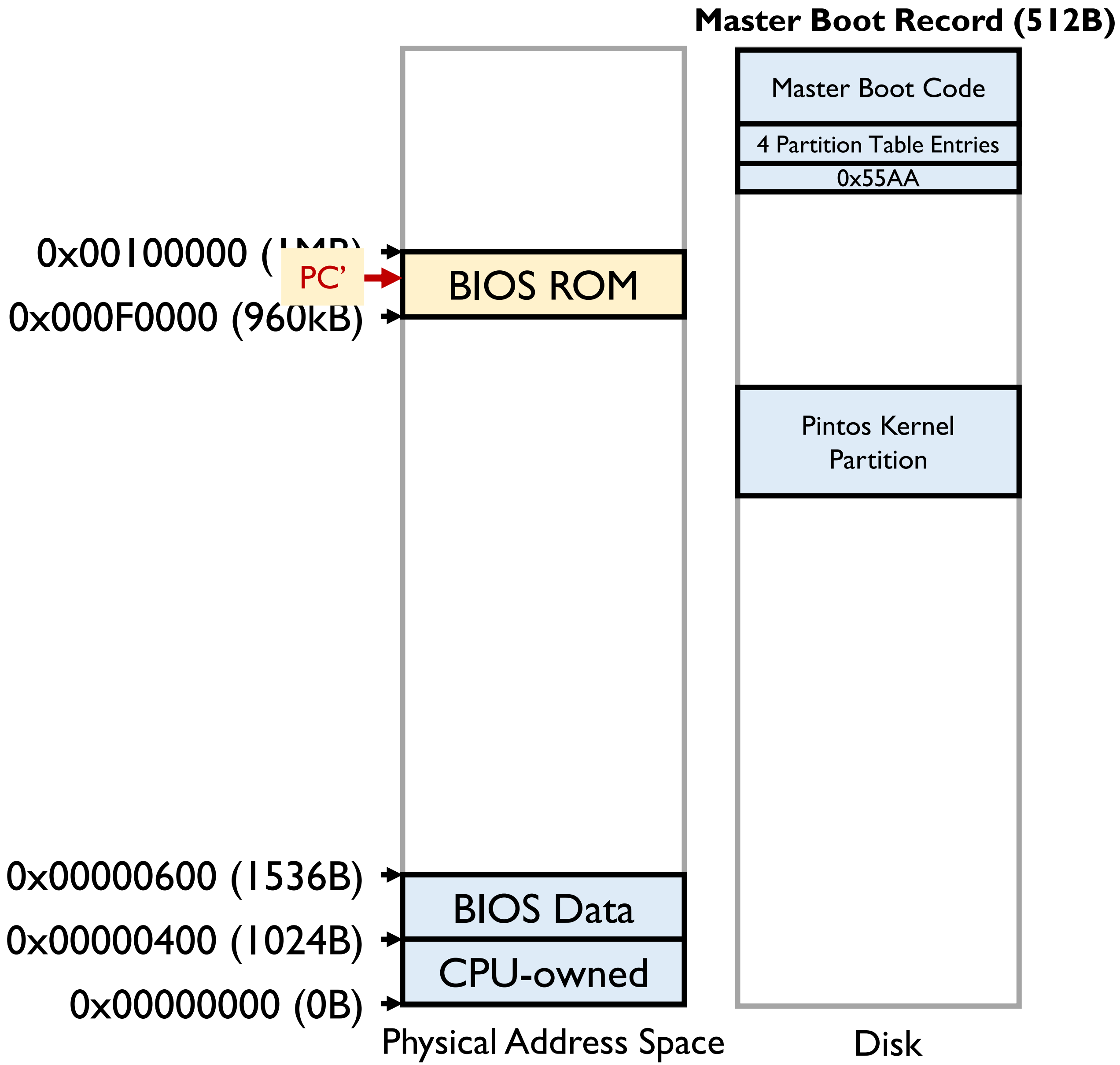
- Lab0 Booting review
- Struct *thread*
- Switching thread
- Create a new thread
- Synchronization
- List
- Lab1 tasks
- Q & A

# Today

- **Lab0 Booting review**
- Struct *thread*
- Switching thread
- Create a new thread
- Synchronization
- List
- Lab1 tasks
- Q & A

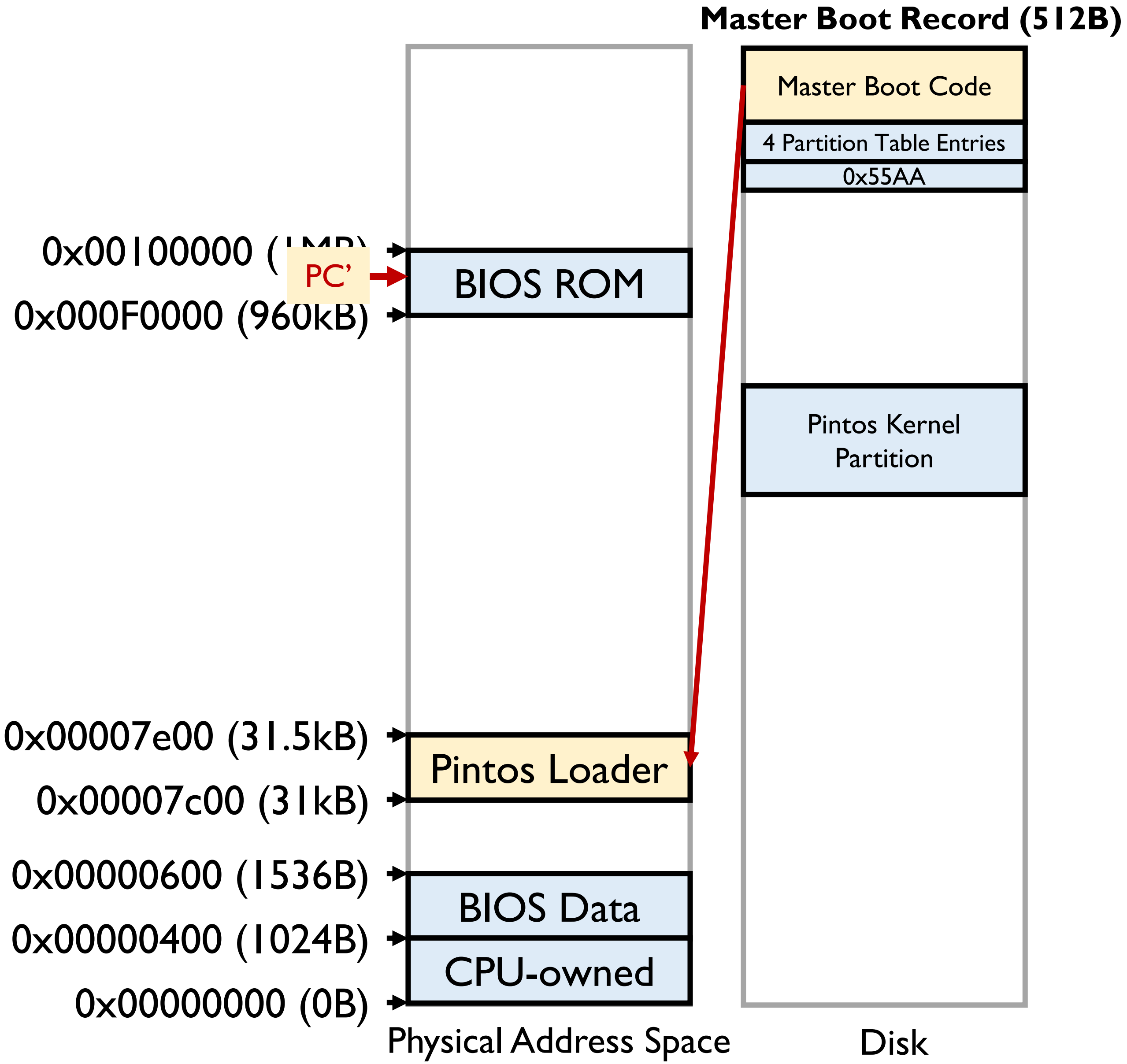
# Lab0 Booting Review

- Power on
- BIOS gets control



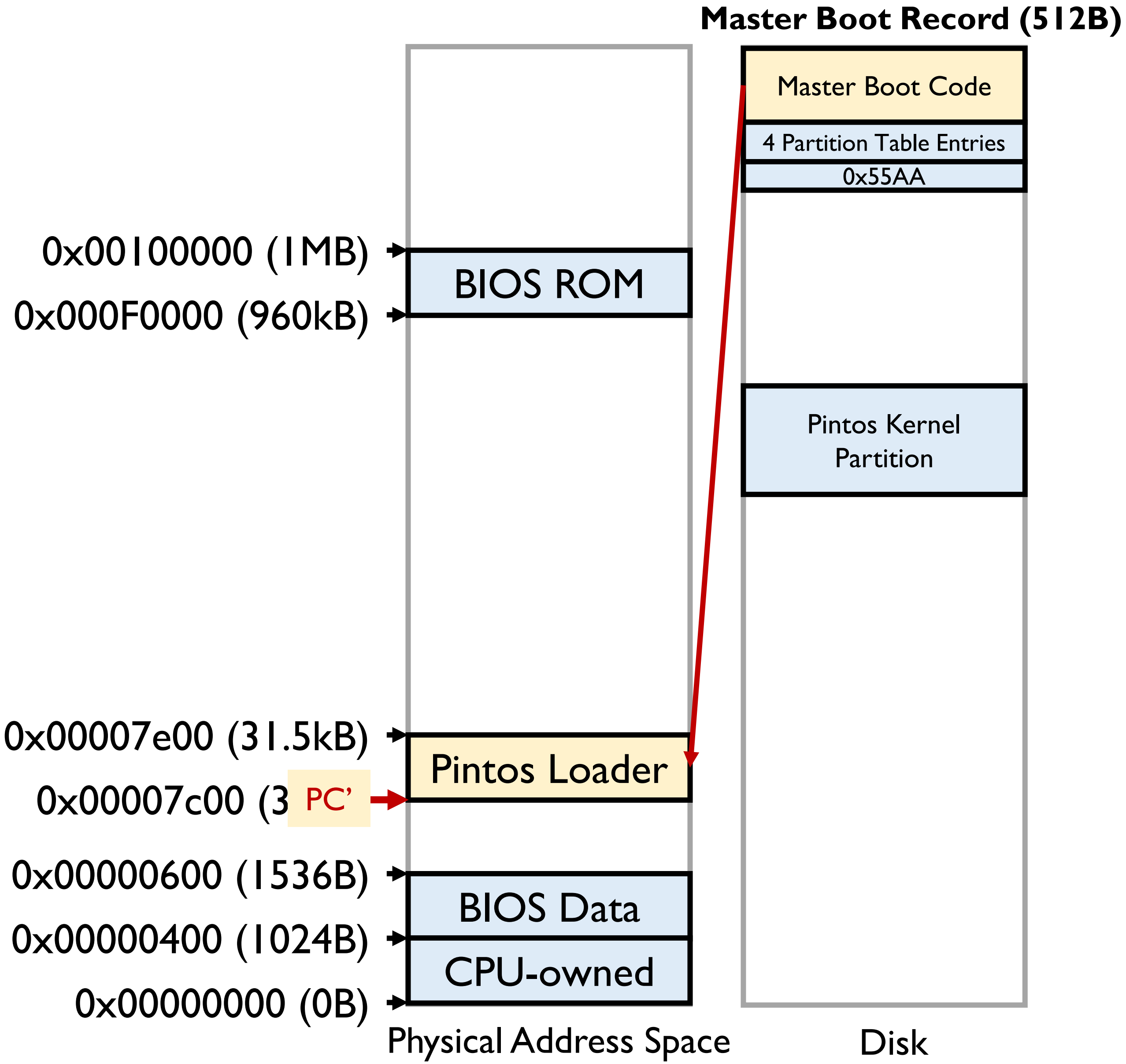
# Lab0 Booting Review

- Power on
- BIOS gets control
- BIOS loads ***pintos loader*** from the first sector of a hard disk to physical address 0x7c00-0x7e00 (512B)



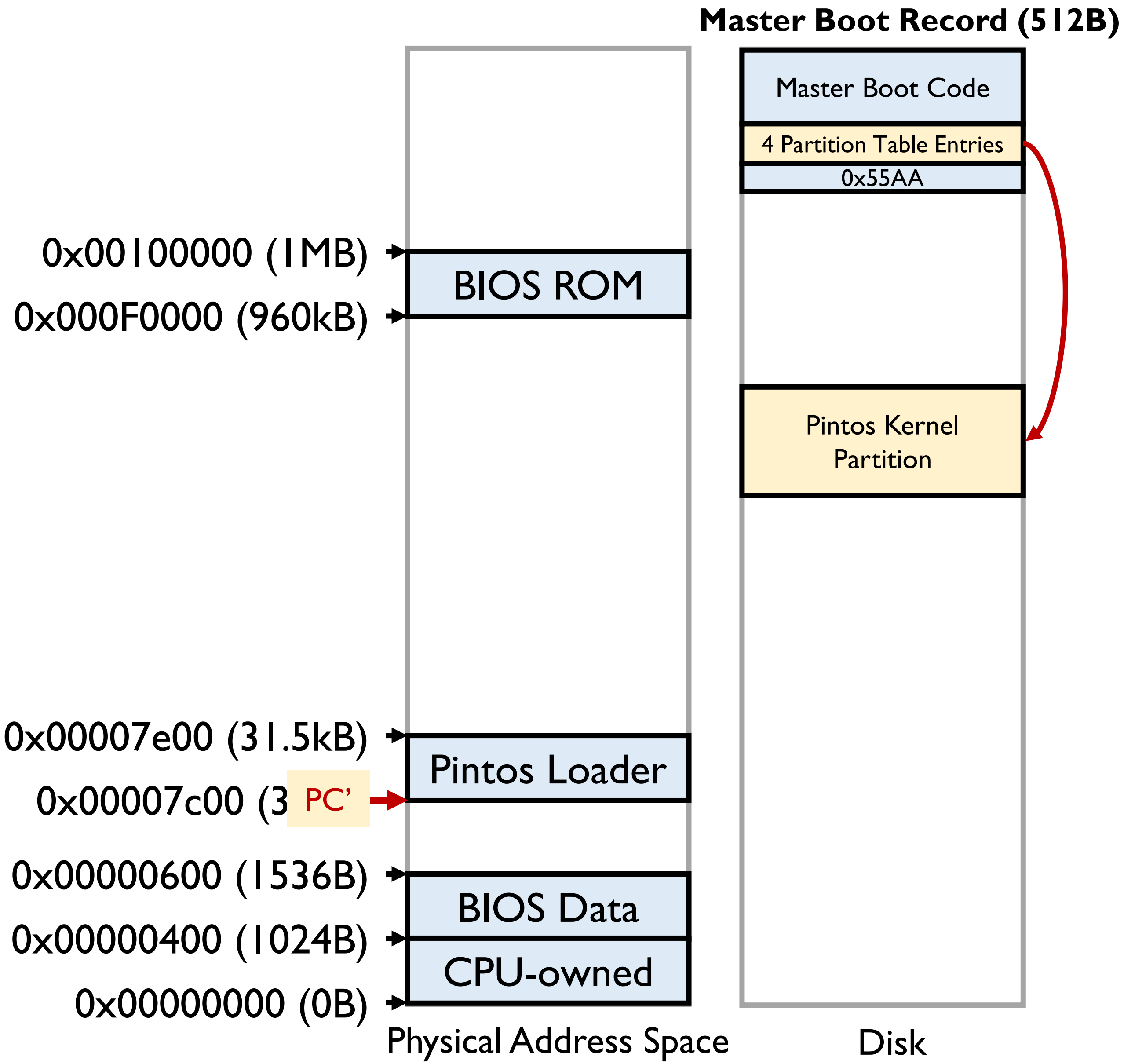
# Lab0 Booting Review

- Power on
- BIOS gets control
- BIOS loads ***pintos loader*** from the first sector of a hard disk to physical address 0x7c00-0x7e00 (512B)
- Jump to the beginning of loader



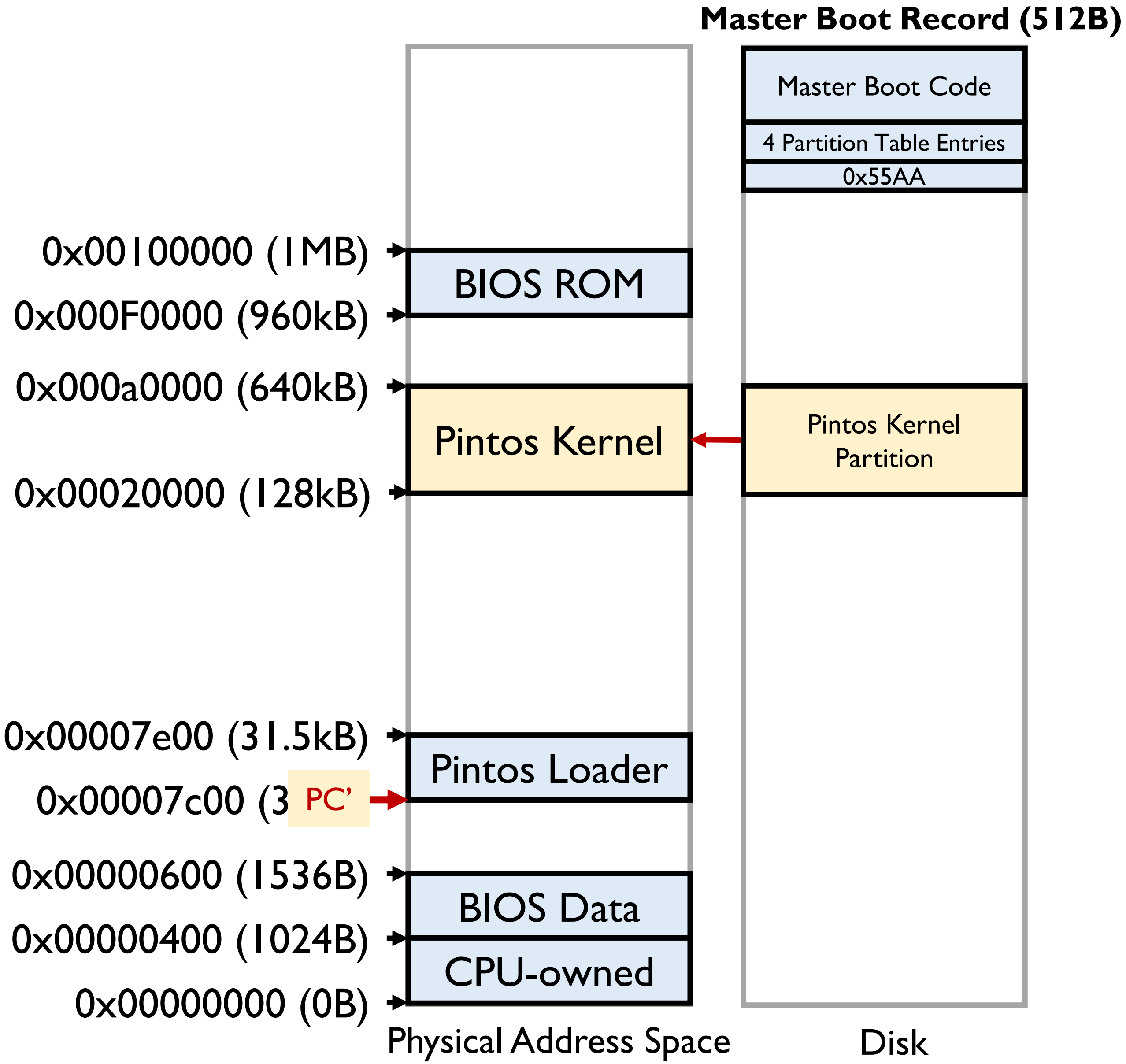
# Lab0 Booting Review

- loader.S
  - Read the **partition table** on each system disk, looking for a **bootable** partition of the type used for a **Pintos Kernel**



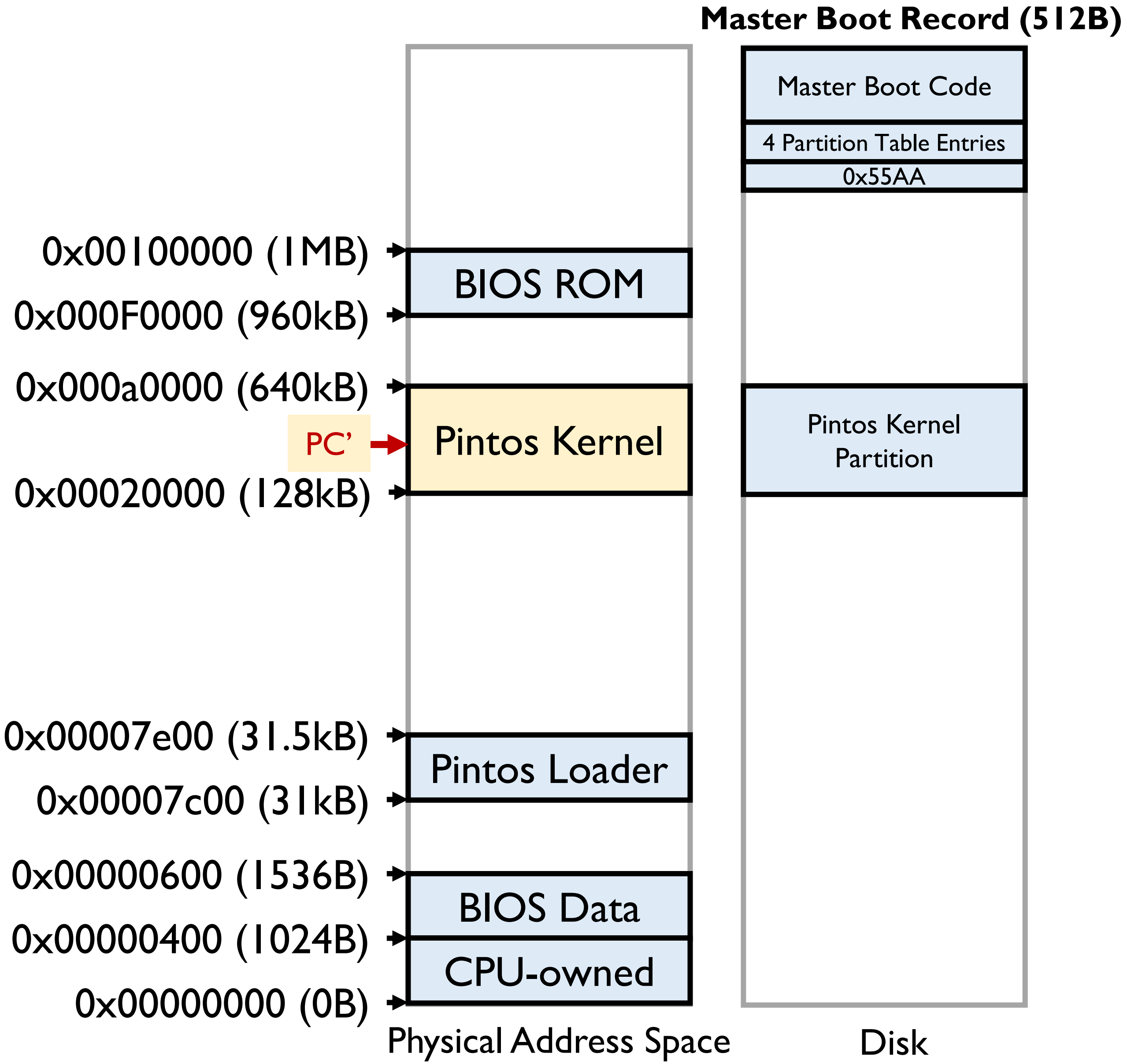
# Lab0 Booting Review

- loader.S
  - Read the **partition table** on each system disk, looking for a **bootable** partition of the type used for a **Pintos Kernel**
  - Read pintos kernel from the disk



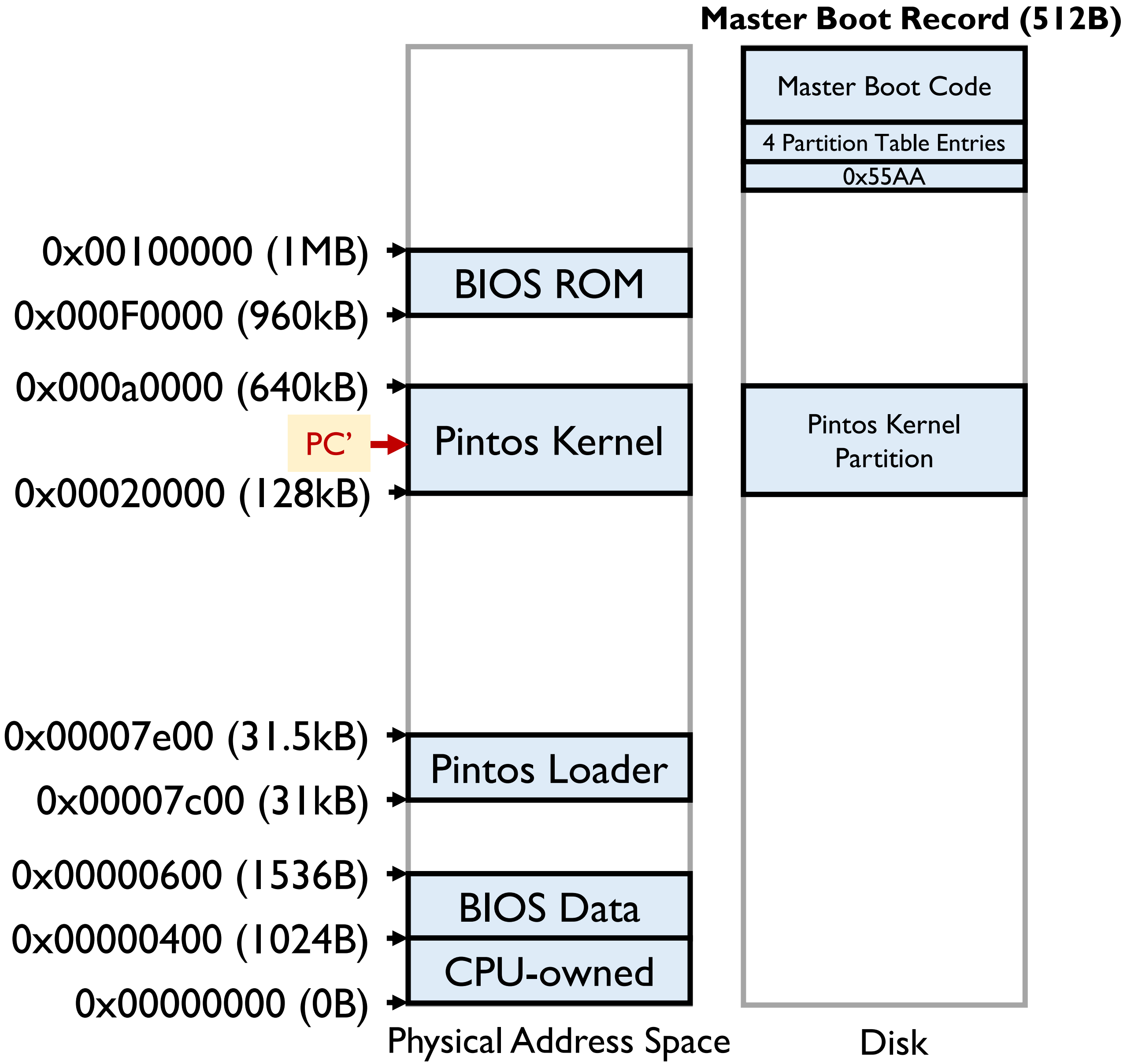
# Lab0 Booting Review

- loader.S
  - Read the **partition table** on each system disk, looking for a **bootable** partition of the type used for a **Pintos Kernel**
  - Read pintos kernel from the disk
  - Jump to its start address
    - Read the start address out of **ELF header**
    - Jump to "**start**"



# Lab0 Booting Review

- start.S
  - Obtain the machine's **memory size**
    - global variable *init_ram_pages*
    - up to 64MB
  - Enable the A20 line



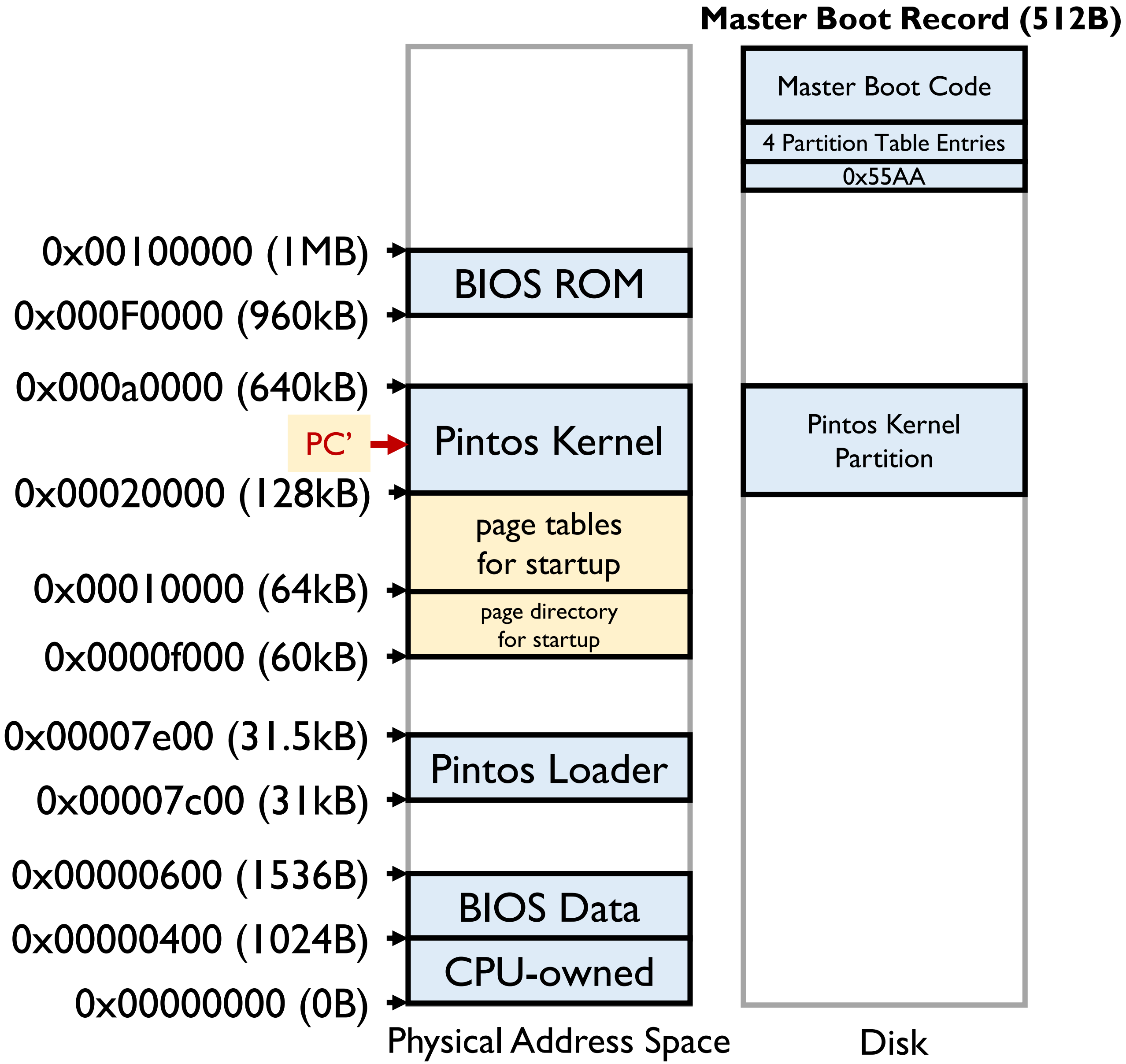
# Lab0 Booting Review

- start.S
  - Obtain the machine's **memory size**
    - global variable *init_ram_pages*
    - up to 64MB
  - Enable the A20 line
  - **Create a basic page table**

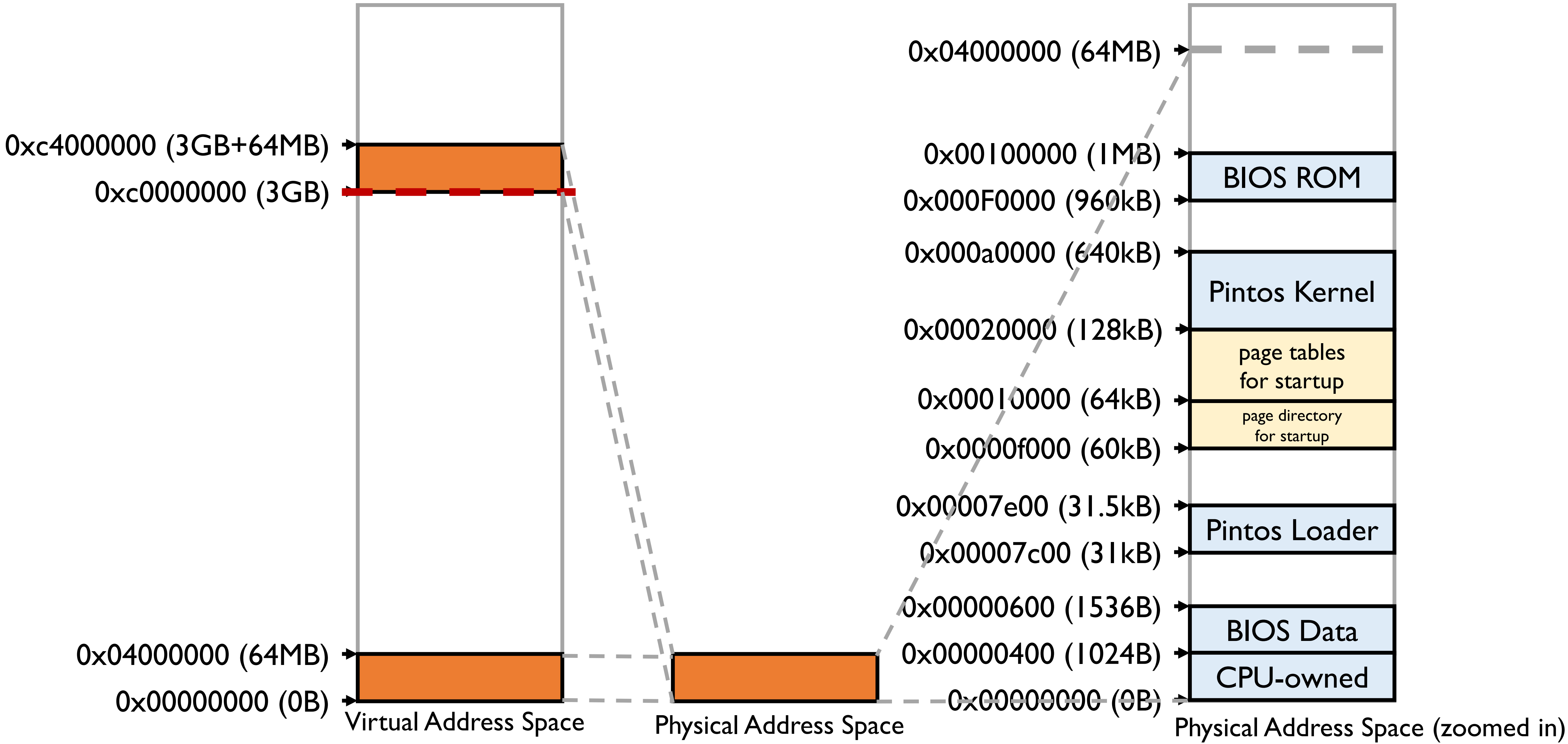
```
# Create page directory at 0xf000 (60 kB) and fill with zeroes.
    mov $0xf00, %ax
    mov %ax, %es
    subl %eax, %eax
    subl %edi, %edi
    movl $0x400, %ecx
    rep stosl
```

```
# Add PDEs to point to page tables for the first 64 MB of RAM.
# Also add identical PDEs starting at LOADER_PHYS_BASE.
# See [IA32-v3a] section 3.7.6 "Page-Directory and Page-Table Entries"
# for a description of the bits in %eax.
```

```
# Set up page tables for one-to-map linear to physical map for the
# first 64 MB of RAM.
# See [IA32-v3a] section 3.7.6 "Page-Directory and Page-Table Entries"
# for a description of the bits in %eax.
```



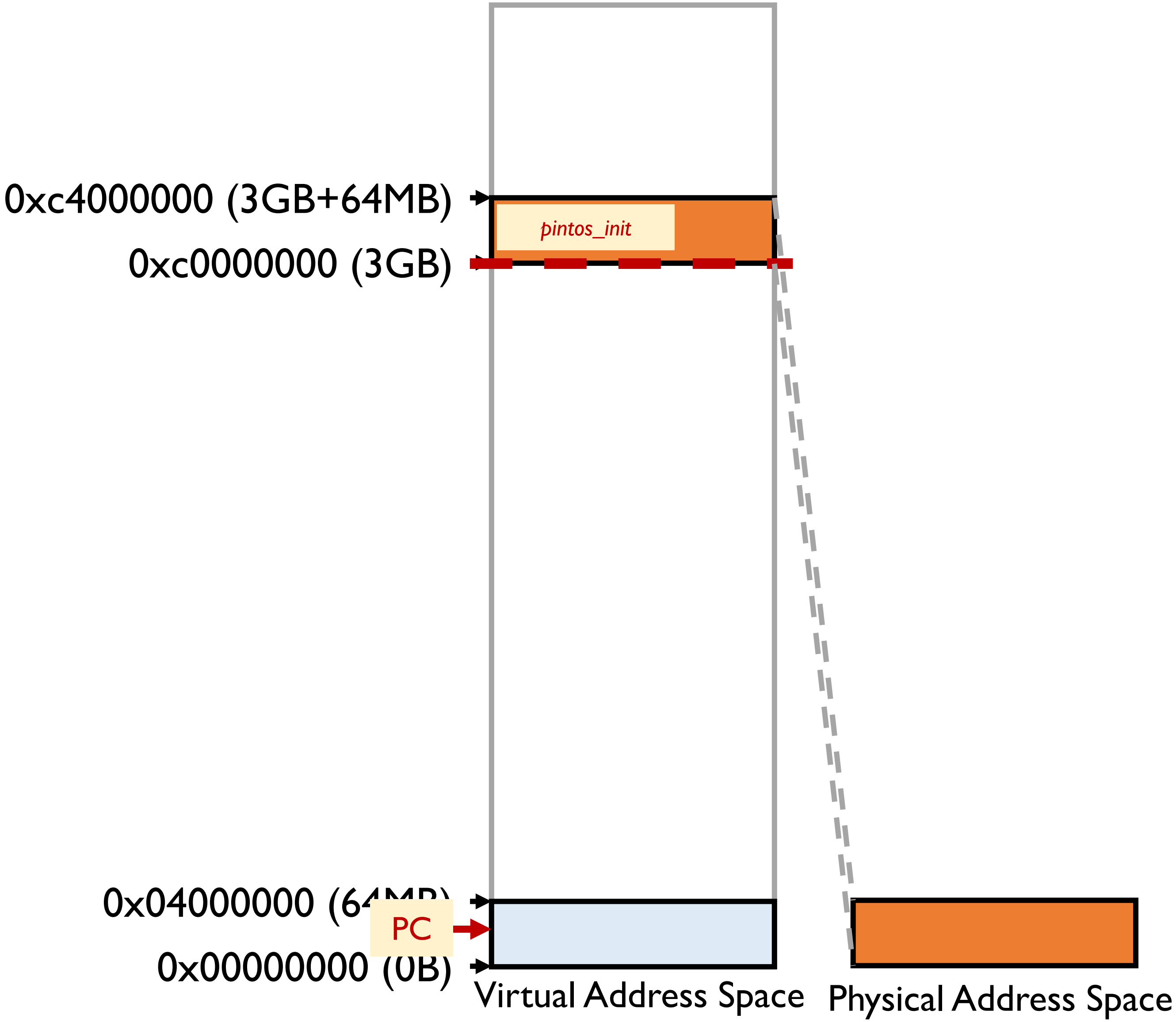
# Basic Page Table



# Why to map the first 64MB of virtual memory

If not map the first 64MB of virtual memory…

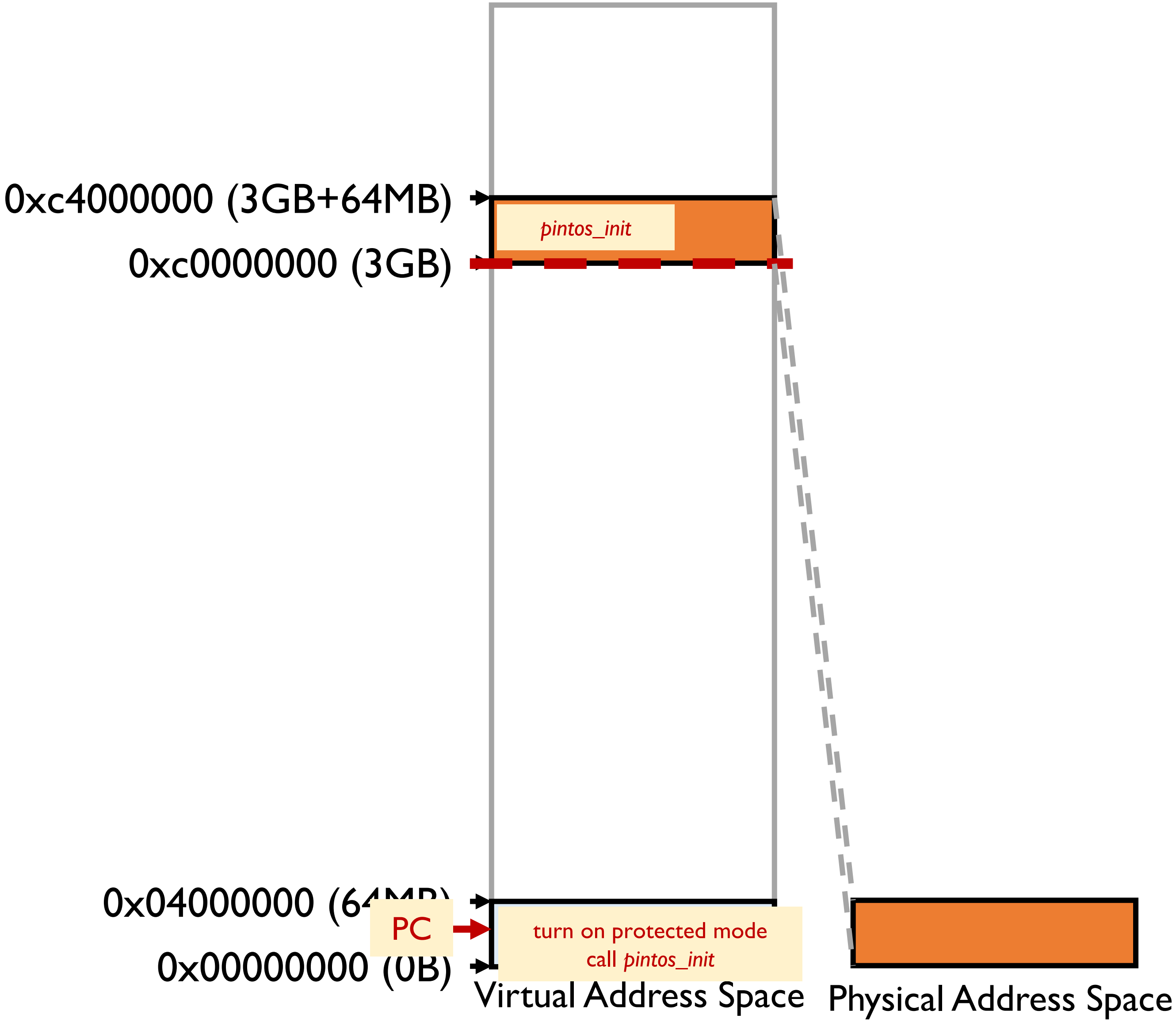
- In raw mode, PC is roughly 0x20000
- The address of *pintos_init* is about 0xc0020000



# Why to map the first 64MB of virtual memory

If not map the first 64MB of virtual memory…

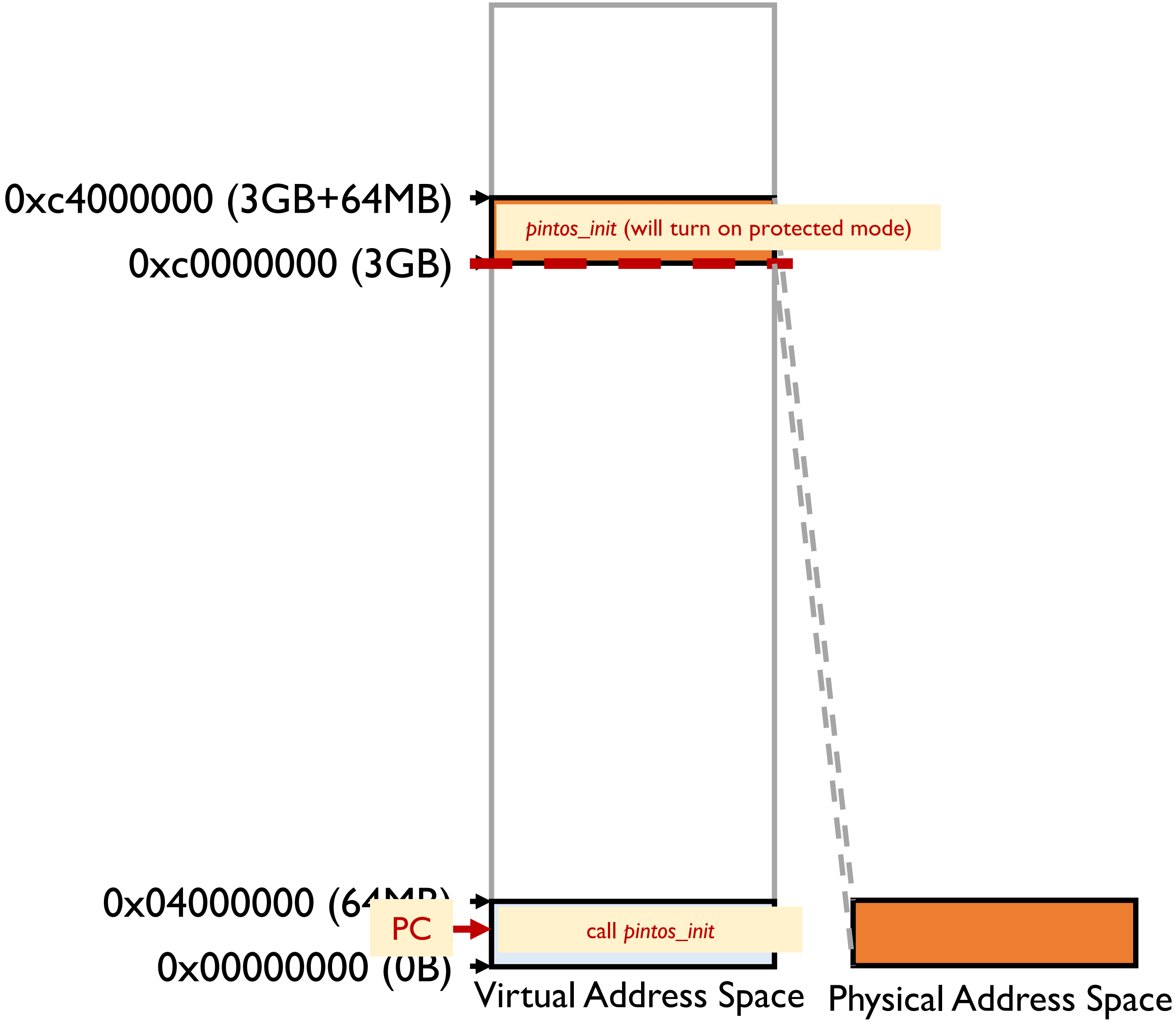
- In raw mode, PC is roughly 0x20000
- The address of *pintos_init* is about 0xc0020000
- **If turn on protected mode before jumping to *pintos_init***
  - cannot call *pintos_init* normally for the erroneous address translation



# Why to map the first 64MB of virtual memory

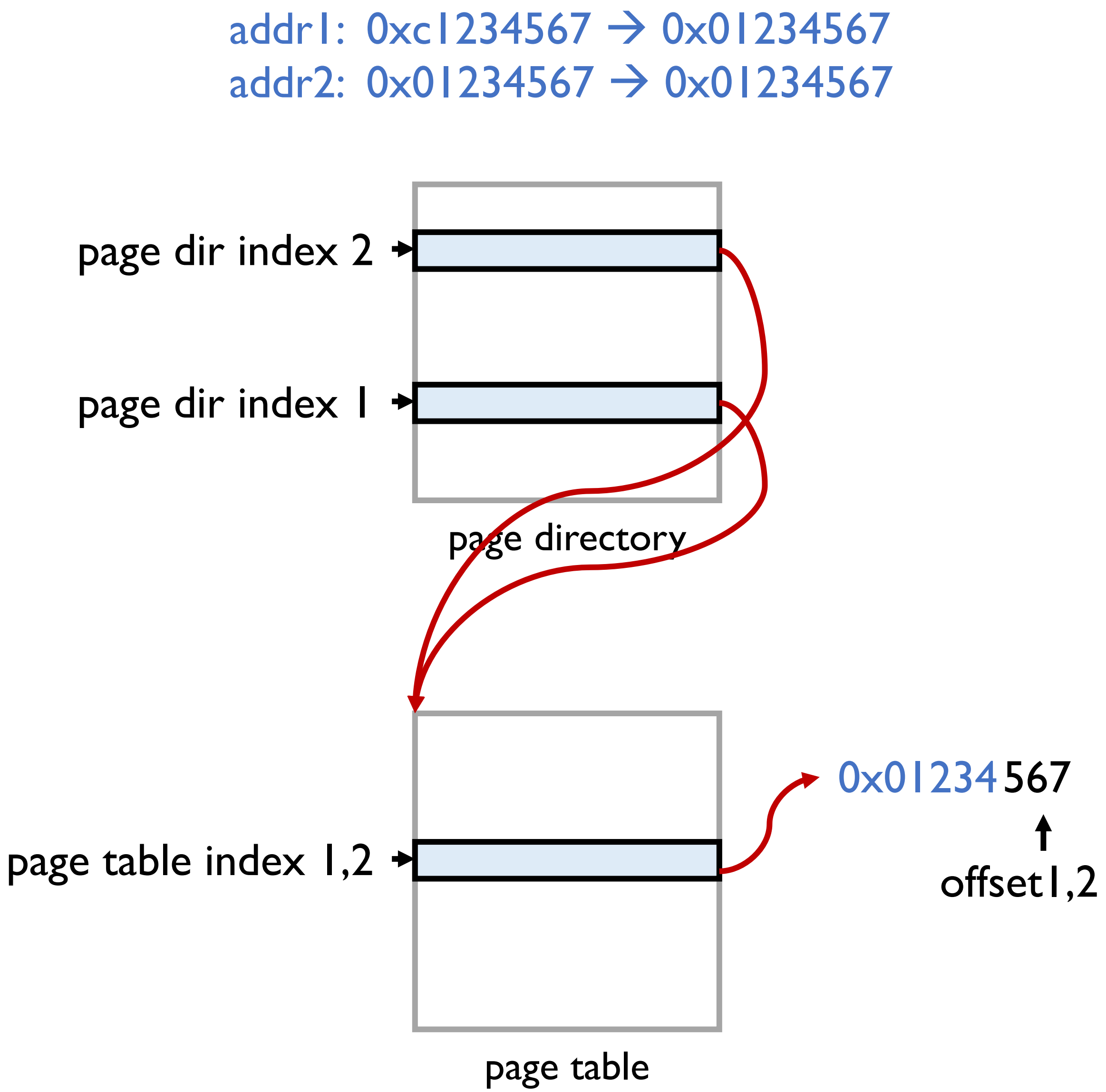
If not map the first 64MB of virtual memory…

- In raw mode, PC is roughly 0x20000
- The address of *pintos_init* is about 0xc0020000
- **If turn on protected mode before jumping to *pintos_init***
  - cannot call *pintos_init* normally for the erroneous address translation
- **If jump to *pintos_init* before turning on protected mode**
  - jump over 0xc0000000 without address translation on



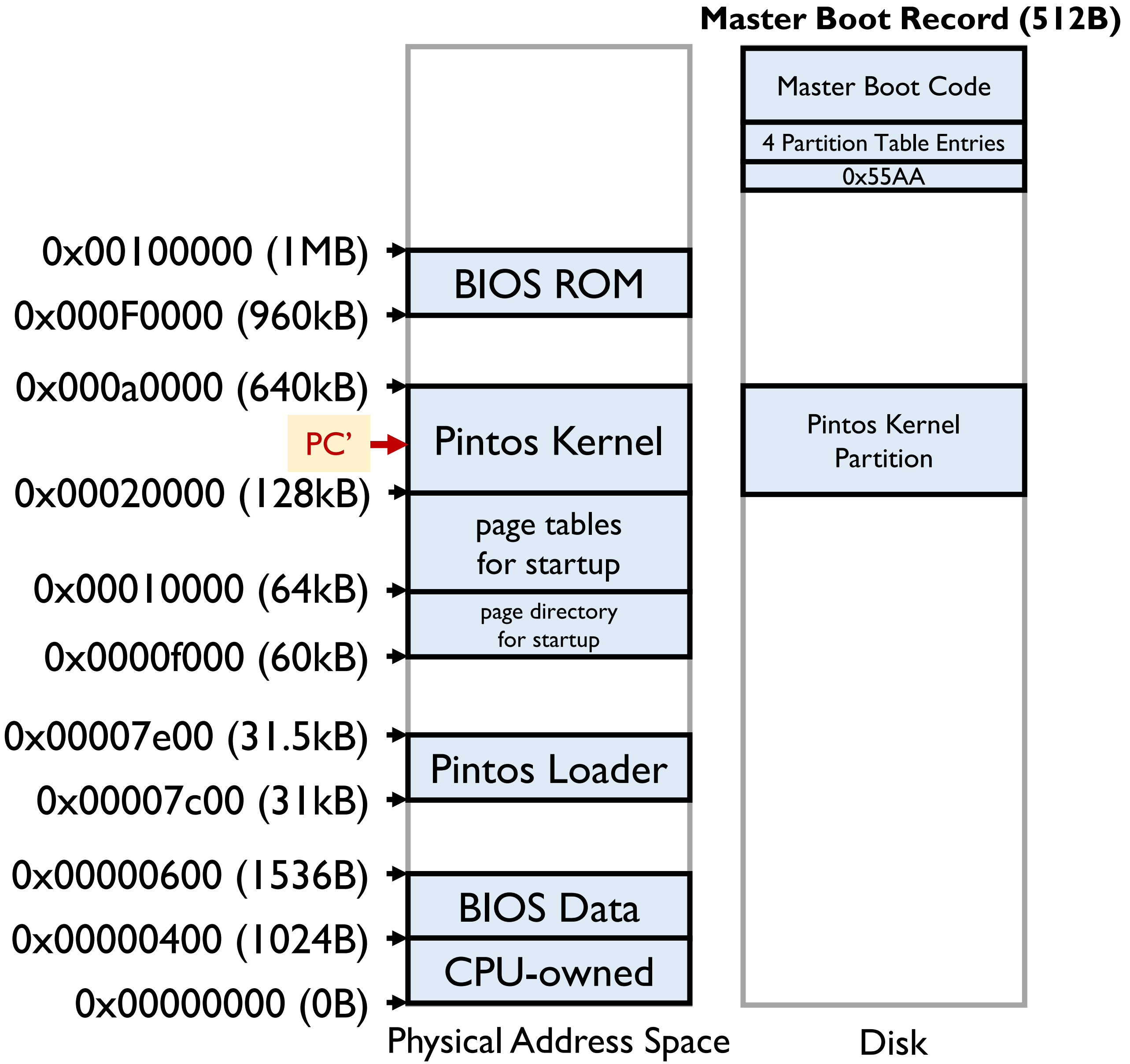
# Basic Page Table

- Only map 64MB and **multiplex** page tables
  - Virtual addresses *0xc0000000 + y* and *y* should be translated to physical address *y*
  - They have different indices of page directory, same index of page table and offset
- **1 page directory**:
  - 1 * 4KB = 4KB
- **16 page tables**:
  - 64MB/4MB * 4KB = 64KB
- the rest mapping: not used so far



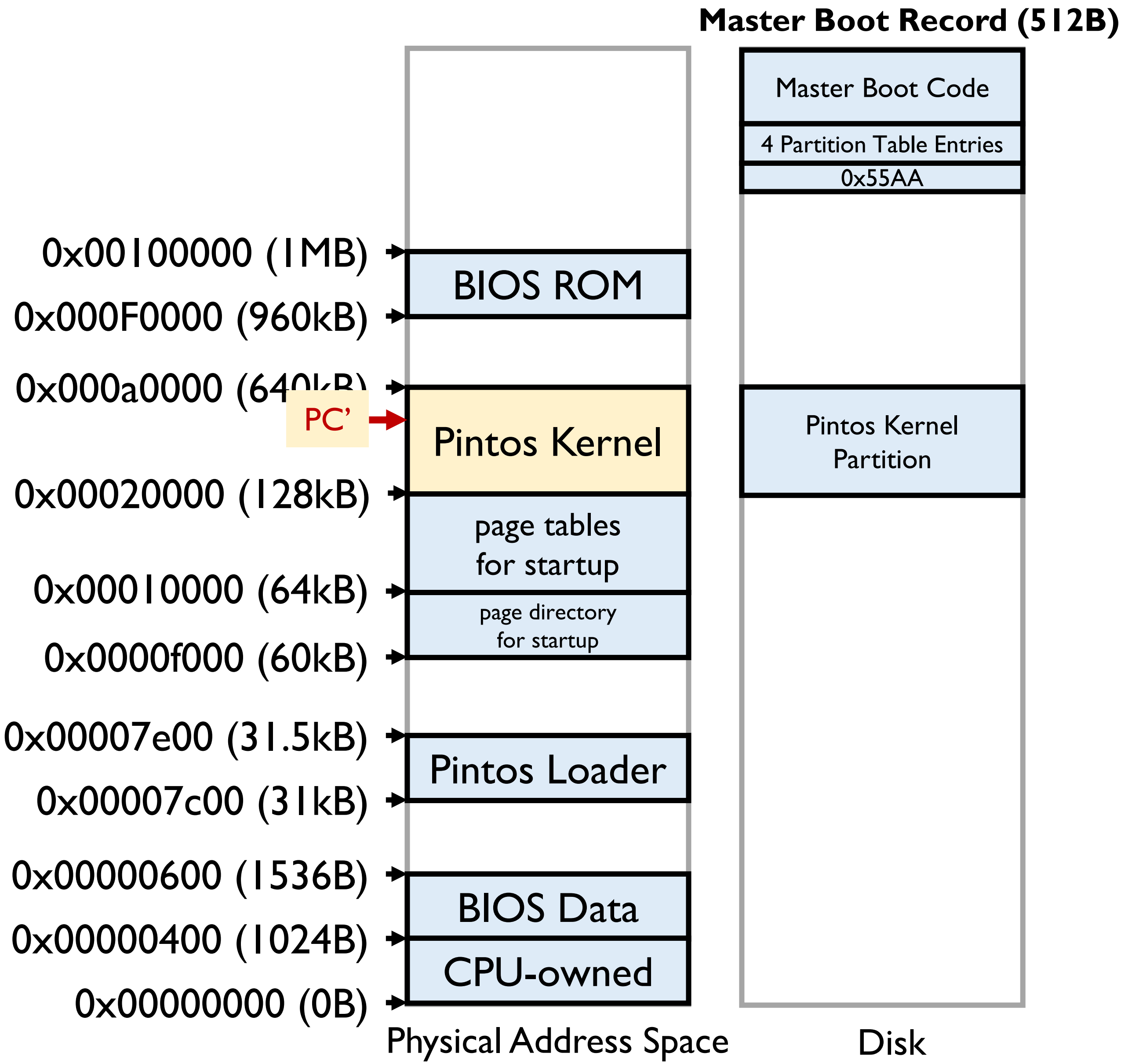
# Lab0 Booting Review

- start.S
  - Obtain the machine's **memory size**
    - global variable *init_ram_pages*
    - up to 64MB
  - Enable the A20 line
  - **Create a basic page table**
  - Turn on protected mode and paging
  - Set up the segment registers



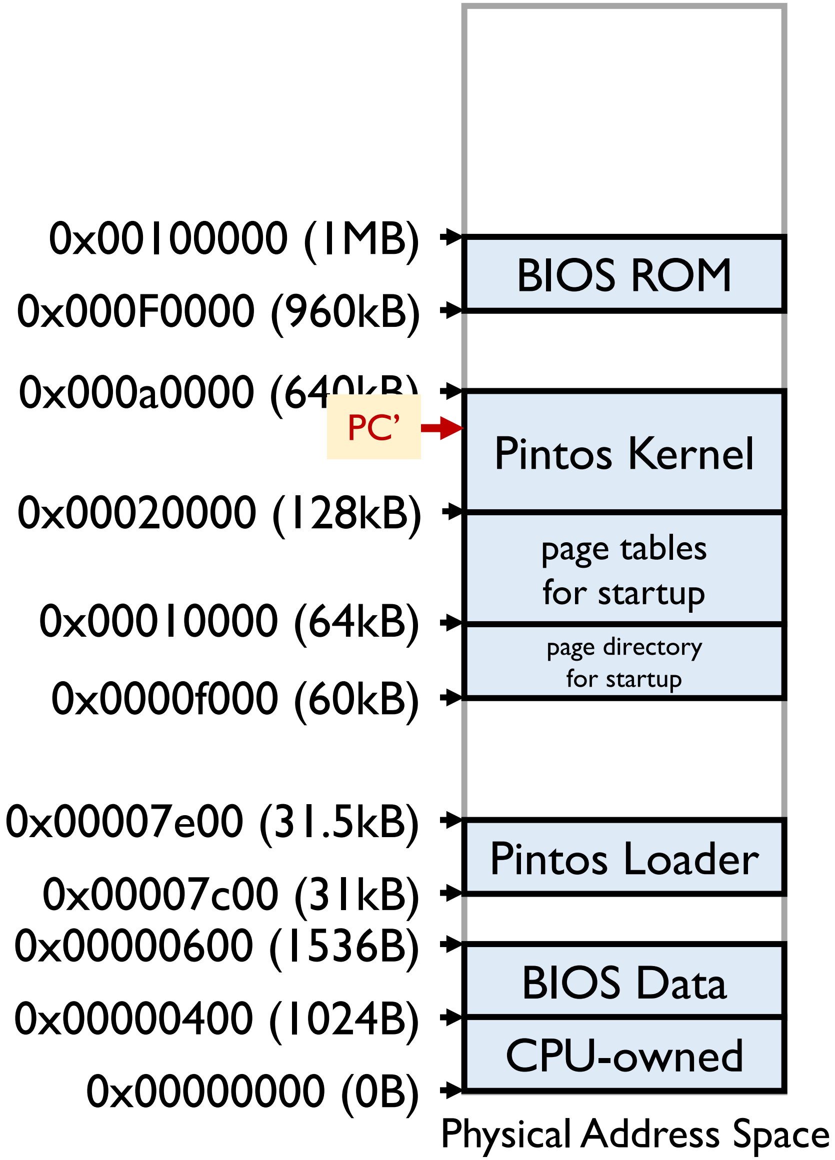
# Lab0 Booting Review

- start.S
  - Obtain the machine's **memory size**
    - global variable *init_ram_pages*
    - up to 64MB
  - Enable the A20 line
  - **Create a basic page table**
  - Turn on protected mode and paging
  - Set up the segment registers
  - Call *pintos_init()*



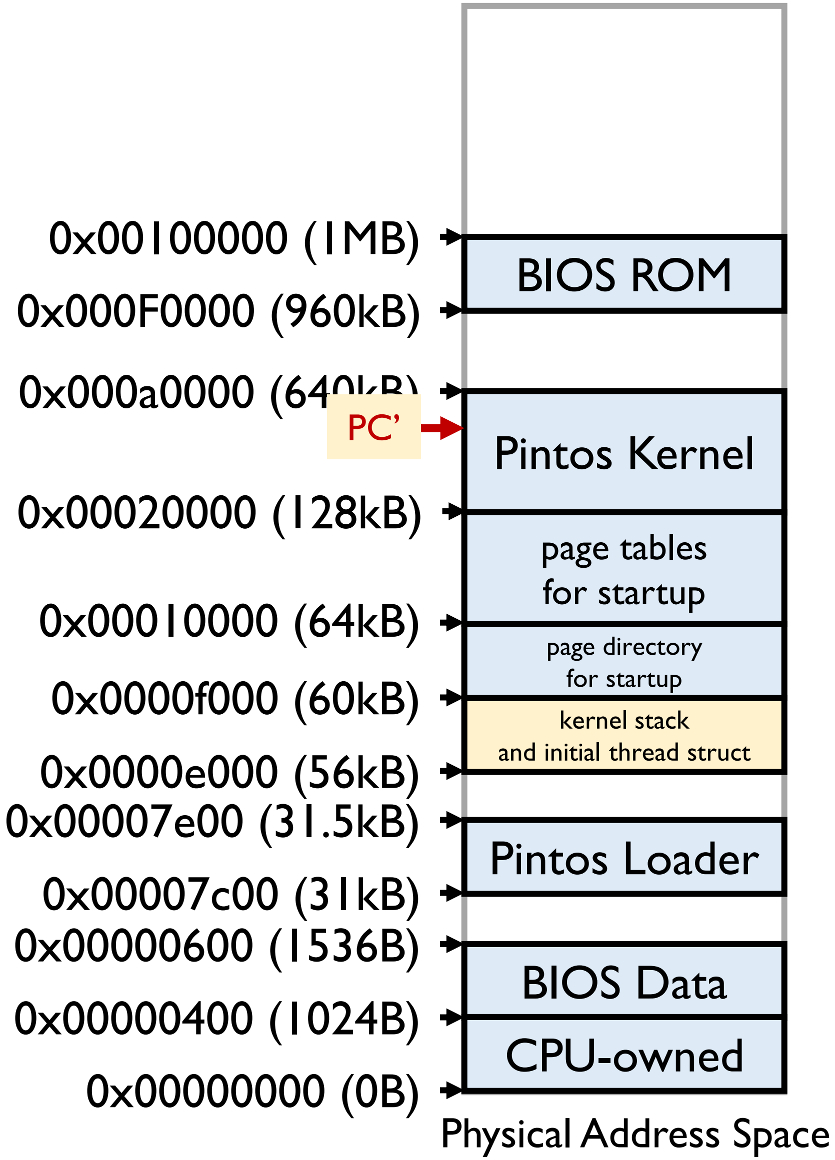
# Lab0 Booting Review

- about *pintos_init()*



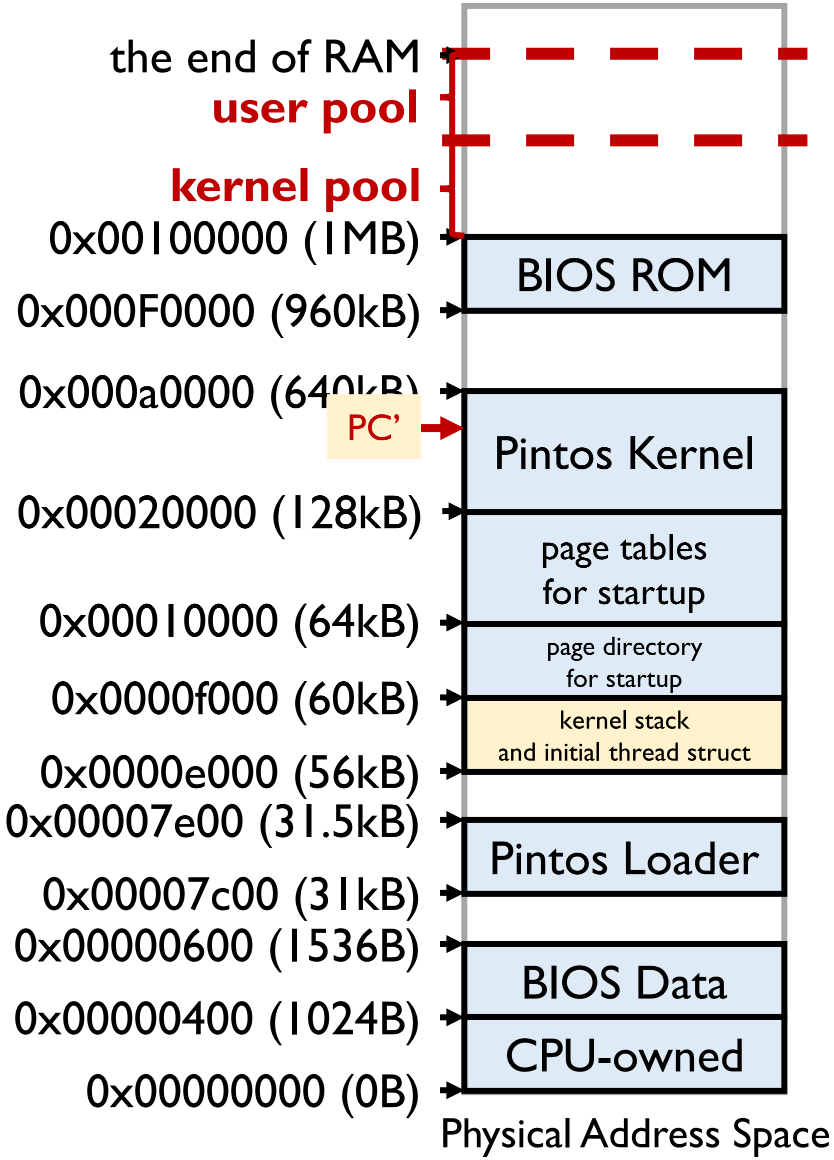
# Lab0 Booting Review

- about *pintos_init()*
  - **Initialize the first thread: *thread_init***
    - Transform the currently running code into a thread
    - Struct thread later



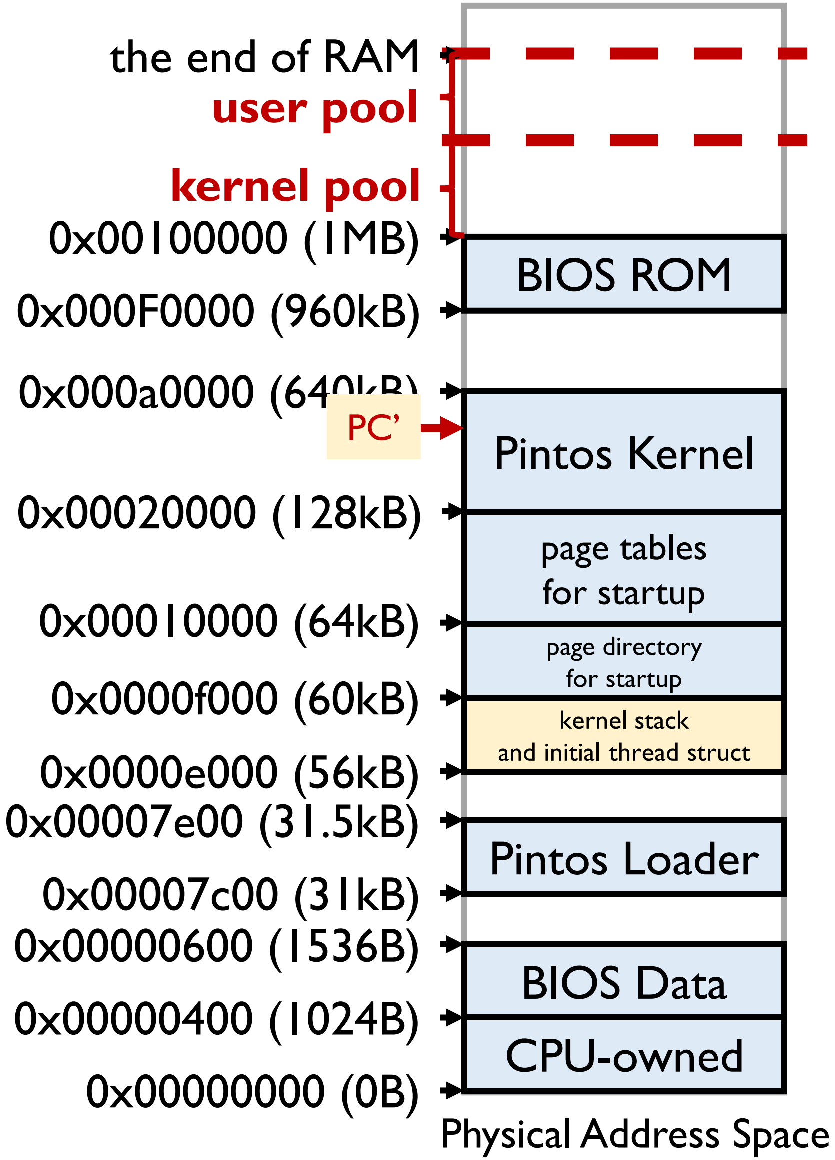
# Lab0 Booting Review

- about *pintos_init()*
  - **Memory management: *palloc_init, malloc_init, paging_init***
    - ***Page allocator (palloc)***
      - Hand out memory in **page-size or page-multiple chunks**
      - Free memory starts at **1MB** and runs to the end of RAM
      - **System memory is divided into two pools called *kernel pool* and *user pool* respectively.** The user pool should be used for allocating memory for user processes and the kernel pool for all other allocations.
      - By default, half of system RAM is given to kernel pool and half to the user pool
      - Until lab3, all allocations should be made from ***the kernel pool***



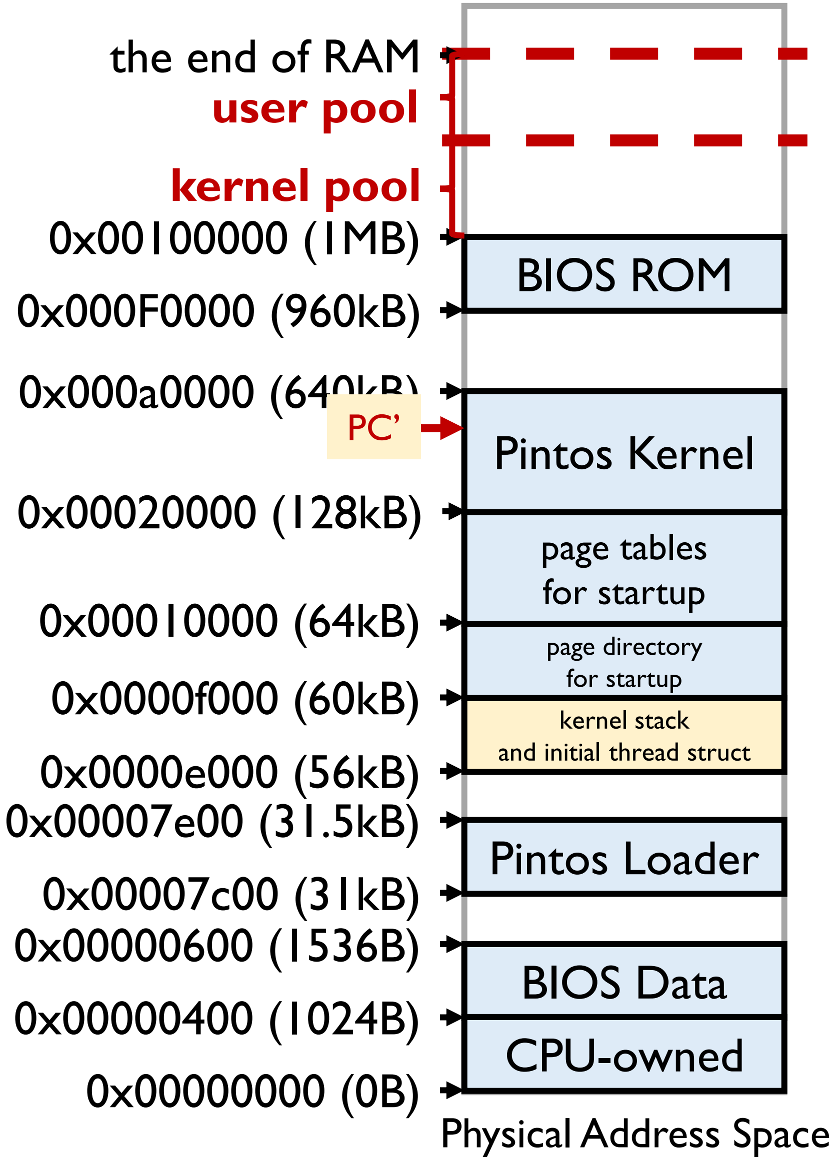
# Lab0 Booting Review

- about *pintos_init()*
  - **Memory management: *palloc_init, malloc_init, paging_init***
    - ***Block allocator (malloc)***
      - On top of page allocator, it can allocate **blocks of any size**
      - Block returned by the block allocator are obtained from ***the kernel pool***
      - Block allocator uses **2 different strategies** for allocating memory (see section Block allocator)



# Lab0 Booting Review

- about *pintos_init()*
  - **Memory management: *palloc_init, malloc_init, paging_init***
    - ***paging_init***
      - Obtain memory from pools and **create a new page directory and page tables** with the kernel virtual mapping (only 0xc0000000-0xc0000000+64MB)
      - Set up the CPU to use the new page directory
      - See source code

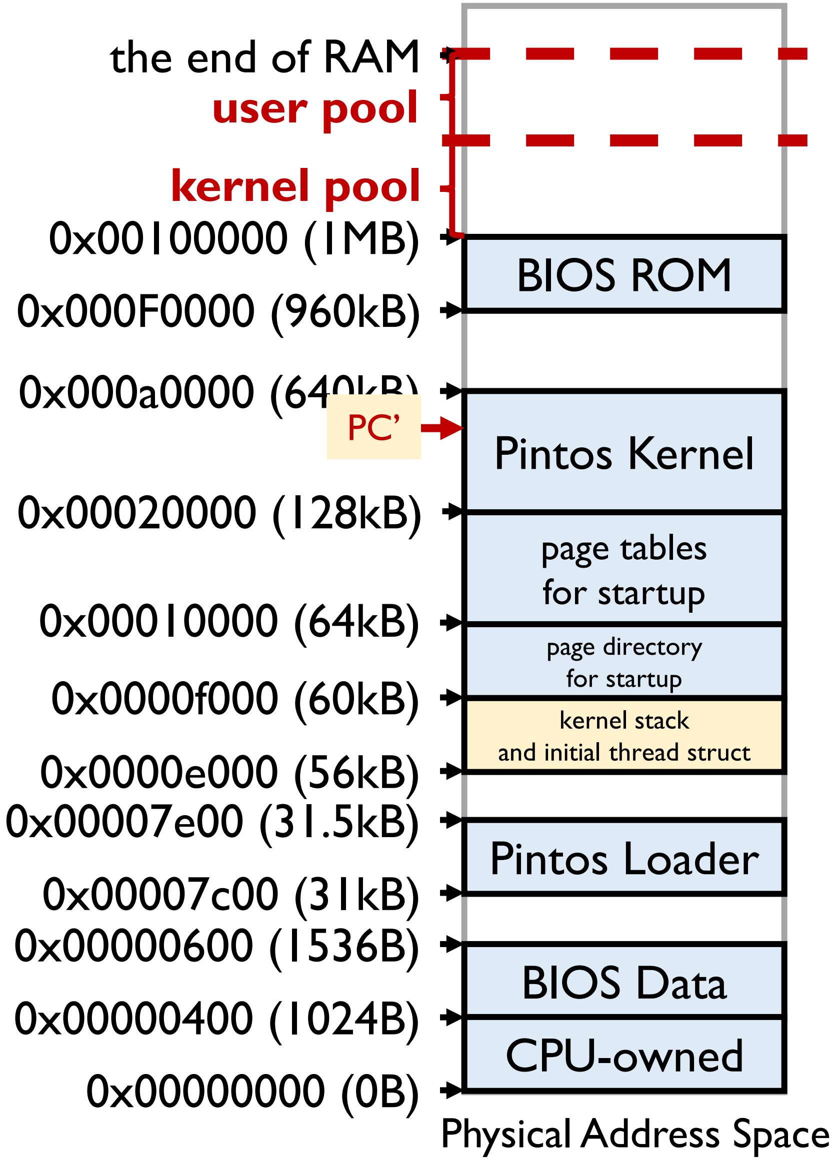


# Lab0 Booting Review

- about *pintos_init()*
  - **Initialize the interrupt system**
    - See section Interrupt Handling / more till lab2
    - **timer:** timer interrupts handled in devices/timer.c *timer_interrupt*

```
/** Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
}
```

  - ***run_actions***
    - Parses and executes actions specified on the kernel command line



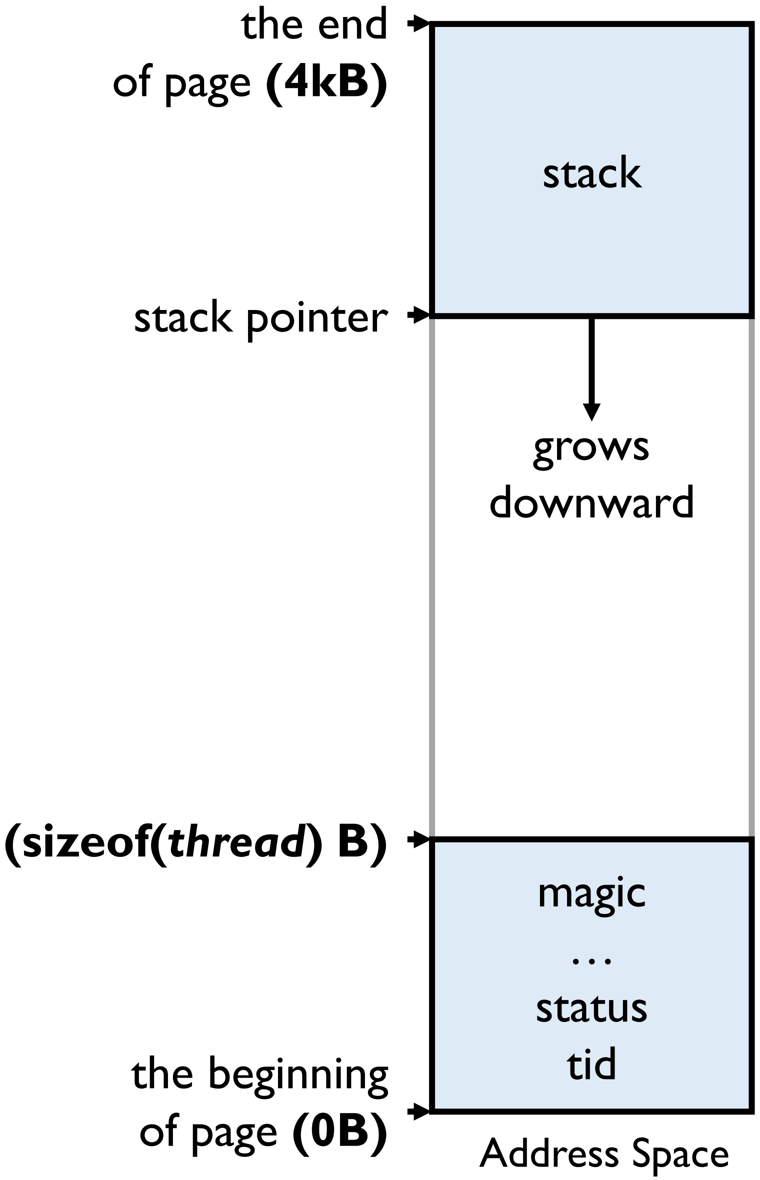
# Today

- Lab0 Booting review
- **Struct *thread***
- Switching thread
- Create a new thread
- Synchronization
- List
- Lab1 tasks
- Q & A

# Struct *thread*

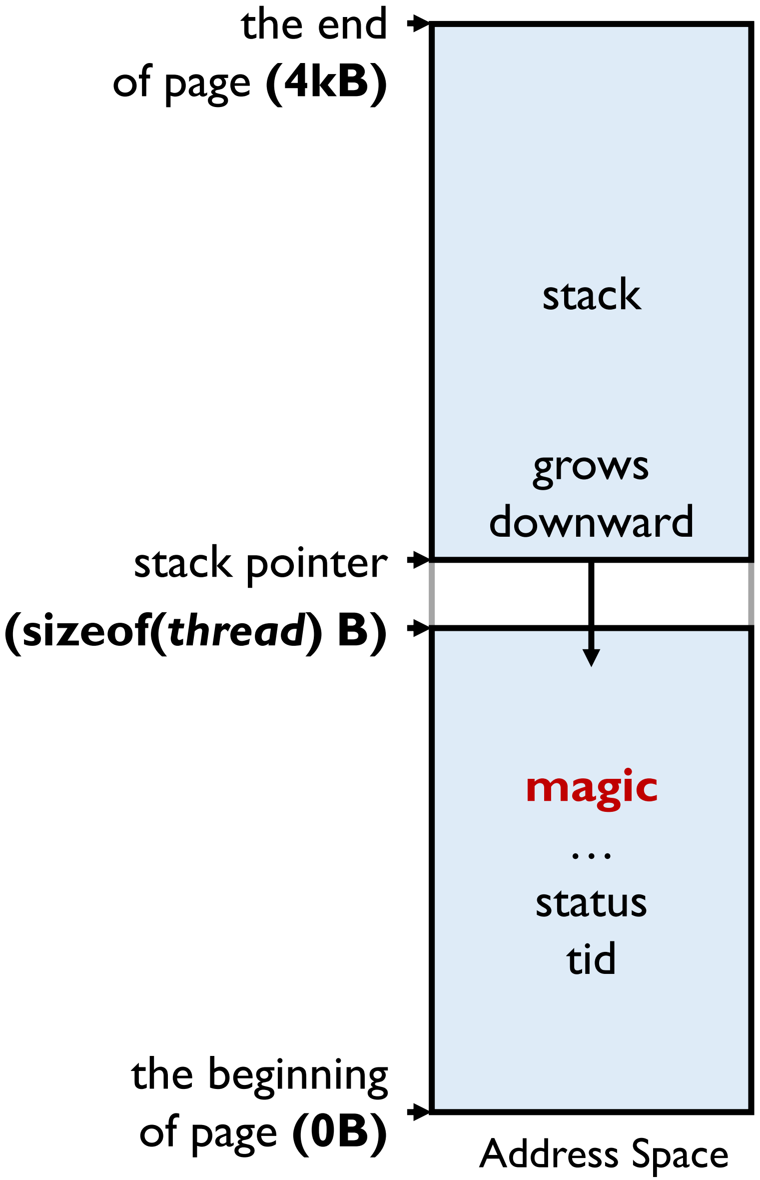
- Memory layout of a running *thread*
  - Struct *thread* represents a thread or a user process
  - Every struct *thread* occupies **the beginning** of its own page of memory
  - The rest of the page is used for the thread's **stack**

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /**< Thread identifier. */
    enum thread_status status;          /**< Thread state. */
    char name[16];                      /**< Name (for debugging purposes). */
    uint8_t *stack;                     /**< Saved stack pointer. */
    int priority;                       /**< Priority. */
    struct list_elem allelem;           /**< List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /**< List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /**< Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /**< Detects stack overflow. */
  };
```



# Struct *thread*

- ***Caveats***
  - Struct *thread* must not be allowed to grow too big
  - Kernel stack must not be allowed to grow too large
    - the size of non-static local variables should not be too large
  - Add new struct members before *magic*



# Initialization of the first thread

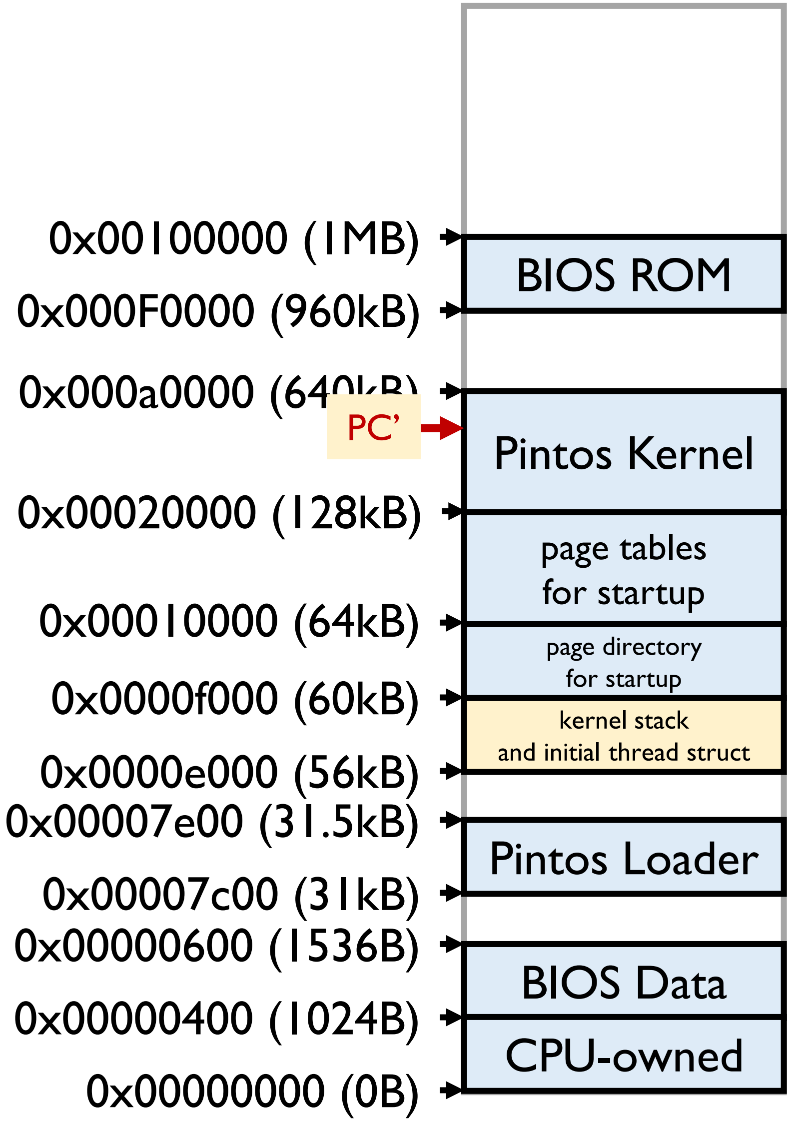
- The first thread is initialized in ***thread_init***

```
/* Set up a thread structure for the running thread. */
initial_thread = running_thread ();
init_thread (initial_thread, "main", PRI_DEFAULT);
```

- The struct address of the first thread depends on %esp, which is initialized in *loader.S*

```
/** Returns the running thread. */
struct thread *
running_thread (void)
{
  uint32_t *esp;

  /* Copy the CPU's stack pointer into `esp', and then round that
     down to the start of a page.  Because `struct thread' is
     always at the beginning of a page and the stack pointer is
     somewhere in the middle, this locates the curent thread. */
  asm ("mov %%esp, %0" : "=g" (esp));
  return pg_round_down (esp);
}
```

```
# Set up segment registers.
# Set stack to grow downward from 60 kB (after boot, the kernel
# continues to use this stack for its initial thread).

    sub %ax, %ax
    mov %ax, %ds
    mov %ax, %ss
    mov $0xf000, %esp
```



# Struct *thread*

- **some handy interfaces**
  - *threads/thread.h*

```c
extern bool thread_mlfqs;

void thread_init (void);
void thread_start (void);

void thread_tick (void);
void thread_print_stats (void);

typedef void thread_func (void *aux);
tid_t thread_create (const char *name, int priority, thread_func *, void *);

void thread_block (void);
void thread_unblock (struct thread *);

struct thread *thread_current (void);
tid_t thread_tid (void);
const char *thread_name (void);

void thread_exit (void) NO_RETURN;
void thread_yield (void);
```

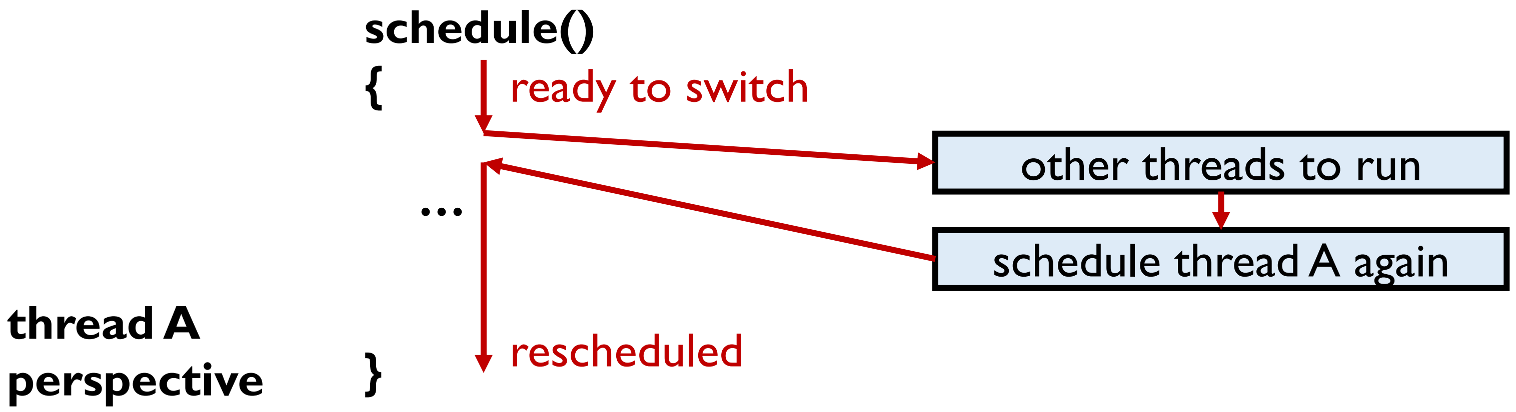
```c
/** Performs some operation on thread t, given auxiliary data AUX. */
typedef void thread_action_func (struct thread *t, void *aux);
void thread_foreach (thread_action_func *, void *);

int thread_get_priority (void);
void thread_set_priority (int);

int thread_get_nice (void);
void thread_set_nice (int);
int thread_get_recent_cpu (void);
int thread_get_load_avg (void);
```

# Today

- Lab0 Booting review
- Struct *thread*
- **Switching thread**
- Create a new thread
- Synchronization
- List
- Lab1 tasks
- Q & A

# Switching thread in Pintos

- Function ***schedule()*** is responsible for switching threads
- **When**
  - ***schedule()*** is called only by three public thread functions:
    - ***thread_block(), thread_exit(), thread_yield()***
- **How (from a thread's perspective)**
  - When a thread **calls *schedule()***, try to find another thread to run and switch to it
  - When a thread **returns from *schedule()***, it is scheduled again



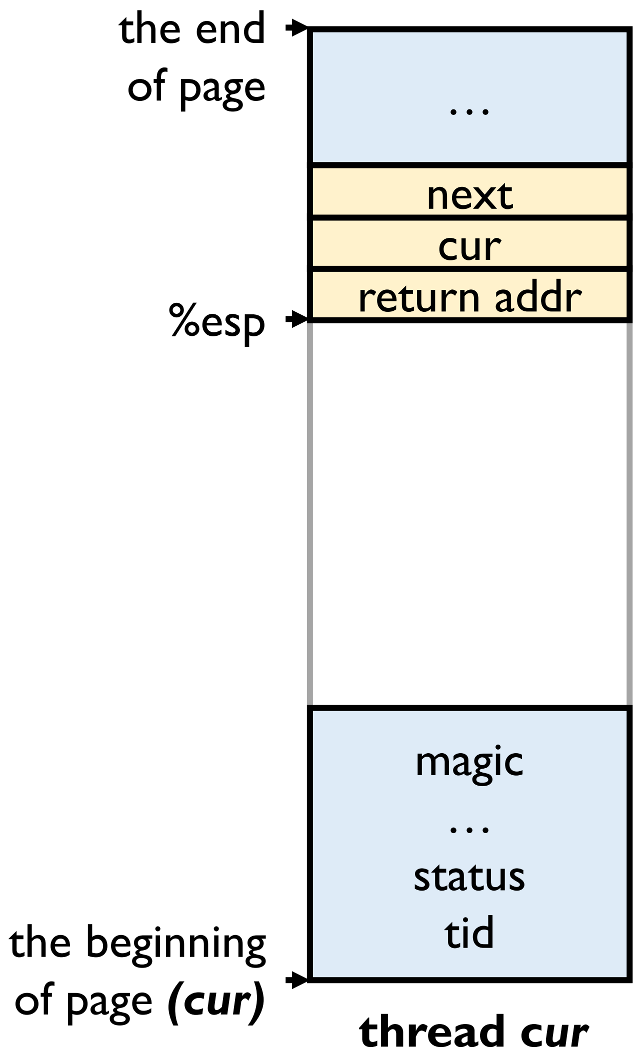
# Source code

- *schedule()*
  - Before calling *schedule()*, make sure **interrupts are disabled**
  - Record the current thread in *cur*
  - Determine the next thread to run *(next)*
  - Call *switch_threads(cur, next)* to do the actual switch

```
static void
schedule (void)
{
  struct thread *cur = running_thread ();
  struct thread *next = next_thread_to_run ();
  struct thread *prev = NULL;

  ASSERT (intr_get_level () == INTR_OFF);
  ASSERT (cur->status != THREAD_RUNNING);
  ASSERT (is_thread (next));

  if (cur != next)
    prev = switch_threads (cur, next);
  thread_schedule_tail (prev);
}
```

# Source code

- ***switch_threads(cur, next)***

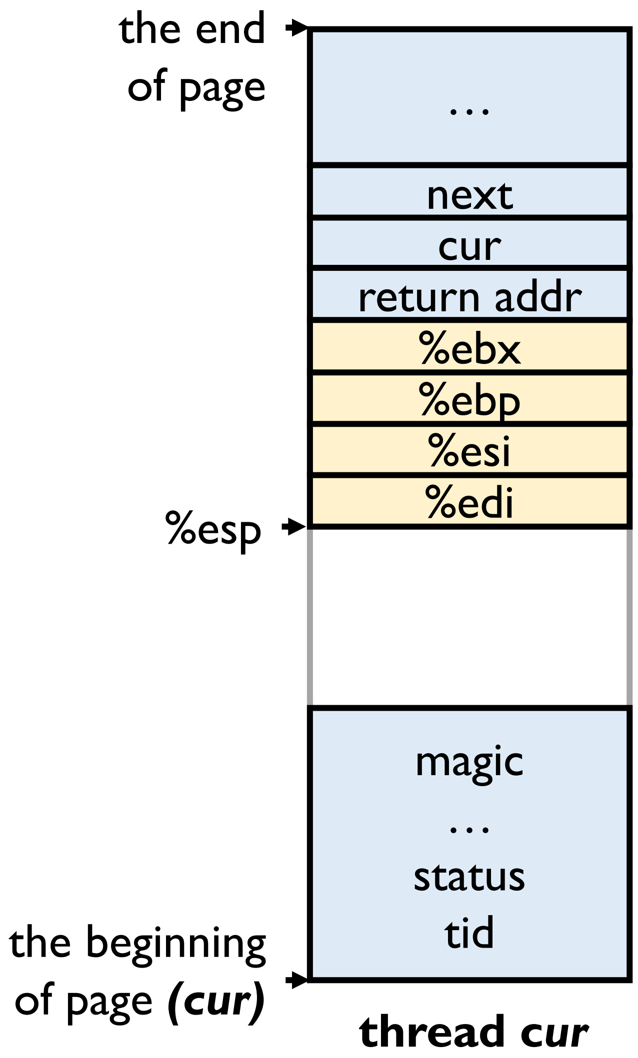



```
.globl switch_threads
.func switch_threads
switch_threads:
        # Save caller's register state.
        #
        # Note that the SVR4 ABI allows us to destroy %eax, %ecx, %edx,
        # but requires us to preserve %ebx, %ebp, %esi, %edi.  See
        # [SysV-ABI-386] pages 3-11 and 3-12 for details.
        #
        # This stack frame must match the one set up by thread_create()
        # in size.
        pushl %ebx
        pushl %ebp
        pushl %esi
        pushl %edi

        # Get offsetof (struct thread, stack).
.globl thread_stack_ofs
        mov thread_stack_ofs, %edx

        # Save current stack pointer to old thread's stack, if any.
        movl SWITCH_CUR(%esp), %eax
        movl %esp, (%eax,%edx,1)

        # Restore stack pointer from new thread's stack.
        movl SWITCH_NEXT(%esp), %ecx
        movl (%ecx,%edx,1), %esp

        # Restore caller's register state.
        popl %edi
        popl %esi
        popl %ebp
        popl %ebx
        ret
.endfunc
```

# Source code

```
/** Offset of `stack' member within `struct thread'.
    Used by switch.S, which can't figure it out on its own. */
uint32_t thread_stack_ofs = offsetof (struct thread, stack);
```

- *switch_threads(cur, next)*

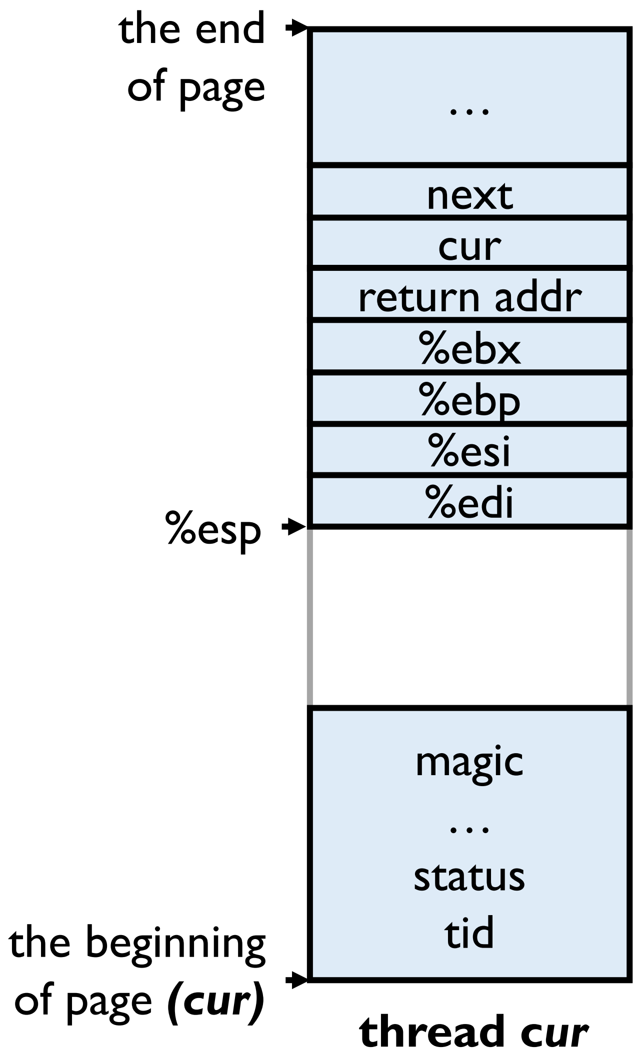



```
.globl switch_threads
.func switch_threads
switch_threads:
        # Save caller's register state.
        #
        # Note that the SVR4 ABI allows us to destroy %eax, %ecx, %edx,
        # but requires us to preserve %ebx, %ebp, %esi, %edi.  See
        # [SysV-ABI-386] pages 3-11 and 3-12 for details.
        #
        # This stack frame must match the one set up by thread_create()
        # in size.
        pushl %ebx
        pushl %ebp
        pushl %esi
        pushl %edi

        # Get offsetof (struct thread, stack).
.globl thread_stack_ofs
        mov thread_stack_ofs, %edx

        # Save current stack pointer to old thread's stack, if any.
        movl SWITCH_CUR(%esp), %eax
        movl %esp, (%eax,%edx,1)

        # Restore stack pointer from new thread's stack.
        movl SWITCH_NEXT(%esp), %ecx
        movl (%ecx,%edx,1), %esp

        # Restore caller's register state.
        popl %edi
        popl %esi
        popl %ebp
        popl %ebx
        ret
.endfunc
```

# Source code

```c
/** Offset of `stack' member within `struct thread'.
    Used by switch.S, which can't figure it out on its own. */
uint32_t thread_stack_ofs = offsetof (struct thread, stack);
```

- ***switch_threads(cur, next)***




```
.globl switch_threads
.func switch_threads
switch_threads:
        # Save caller's register state.
        #
        # Note that the SVR4 ABI allows us to destroy %eax, %ecx, %edx,
        # but requires us to preserve %ebx, %ebp, %esi, %edi.  See
        # [SysV-ABI-386] pages 3-11 and 3-12 for details.
        #
        # This stack frame must match the one set up by thread_create()
        # in size.
        pushl %ebx
        pushl %ebp
        pushl %esi
        pushl %edi

        # Get offsetof (struct thread, stack).
.globl thread_stack_ofs
        mov thread_stack_ofs, %edx

        # Save current stack pointer to old thread's stack, if any.
        movl SWITCH_CUR(%esp), %eax
        movl %esp, (%eax,%edx,1)

        # Restore stack pointer from new thread's stack.
        movl SWITCH_NEXT(%esp), %ecx
        movl (%ecx,%edx,1), %esp

        # Restore caller's register state.
        popl %edi
        popl %esi
        popl %ebp
        popl %ebx
        ret
.endfunc
```

20 — SWITCH_CUR: cur

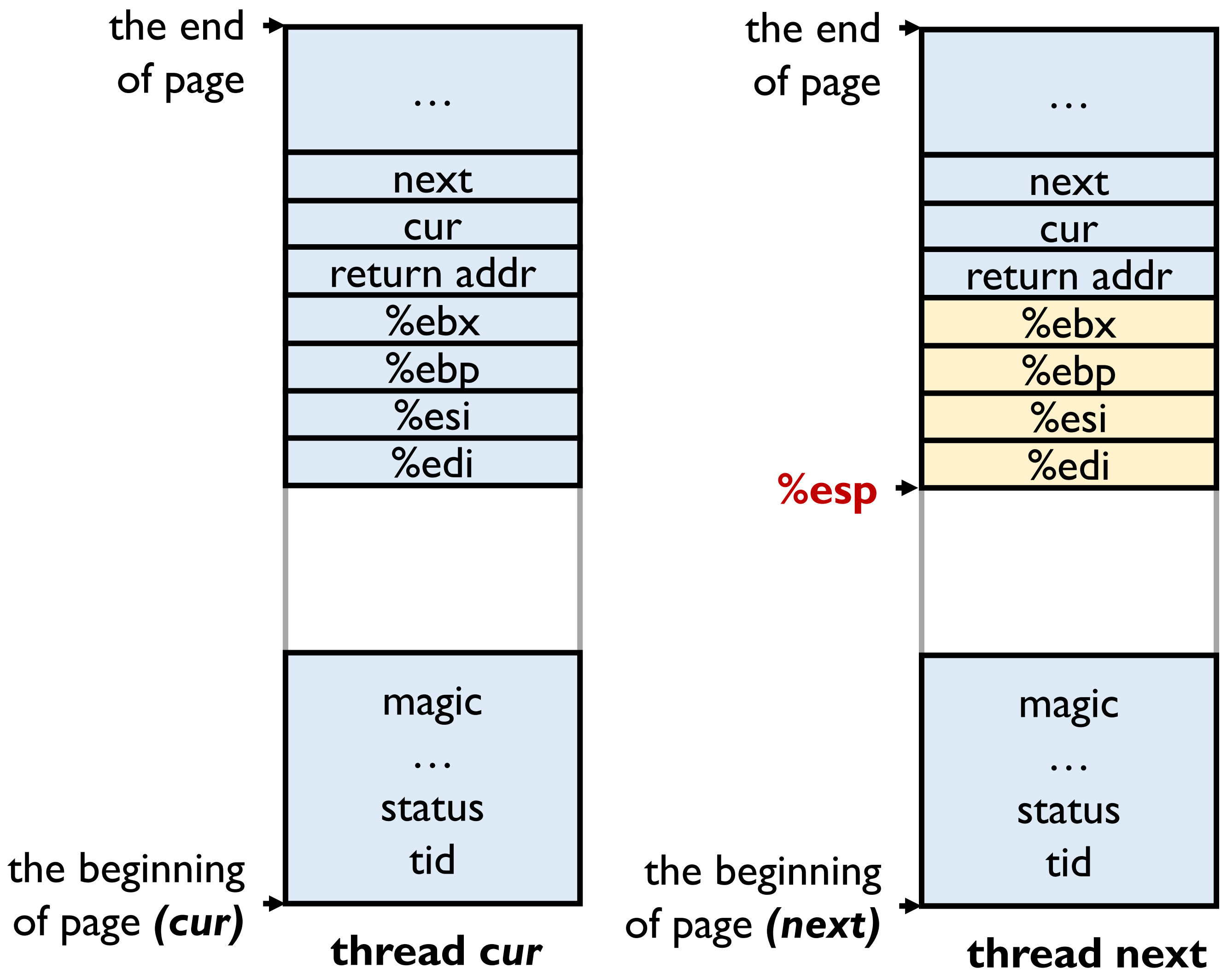
save the stack pointer of cur

24 — SWITCH_NEXT: next

# Source code

```c
/** Offset of `stack' member within `struct thread'.
   Used by switch.S, which can't figure it out on its own. */
uint32_t thread_stack_ofs = offsetof (struct thread, stack);
```

- ***switch_threads(cur, next)***




```asm
.globl switch_threads
.func switch_threads
switch_threads:
        # Save caller's register state.
        #
        # Note that the SVR4 ABI allows us to destroy %eax, %ecx, %edx,
        # but requires us to preserve %ebx, %ebp, %esi, %edi.  See
        # [SysV-ABI-386] pages 3-11 and 3-12 for details.
        #
        # This stack frame must match the one set up by thread_create()
        # in size.
        pushl %ebx
        pushl %ebp
        pushl %esi
        pushl %edi

        # Get offsetof (struct thread, stack).
.globl thread_stack_ofs
        mov thread_stack_ofs, %edx

        # Save current stack pointer to old thread's stack, if any.
        movl SWITCH_CUR(%esp), %eax
        movl %esp, (%eax,%edx,1)

        # Restore stack pointer from new thread's stack.
        movl SWITCH_NEXT(%esp), %ecx
        movl (%ecx,%edx,1), %esp

        # Restore caller's register state.
        popl %edi
        popl %esi
        popl %ebp
        popl %ebx
        ret
.endfunc
```

20 cur

save the stack pointer of cur

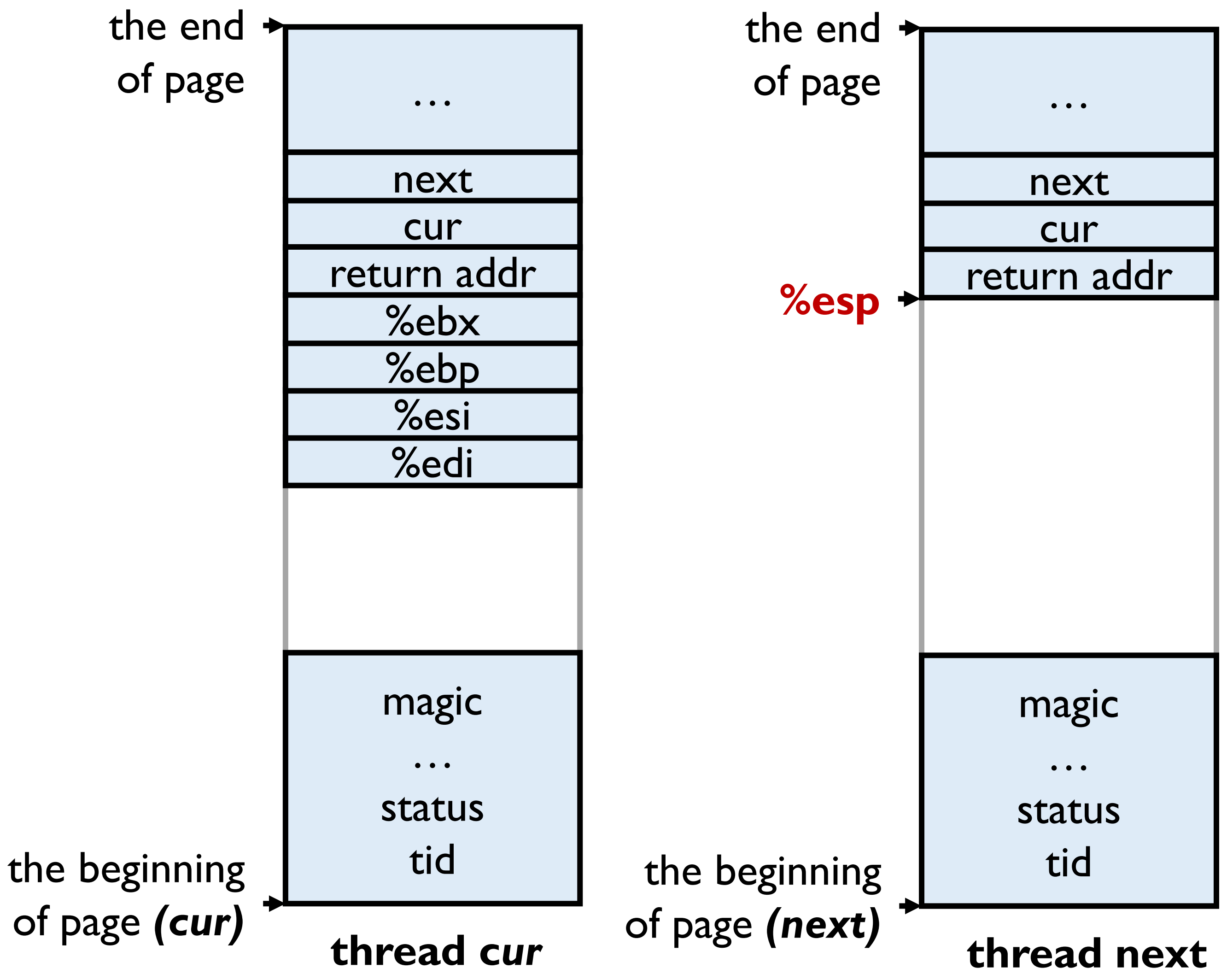
24 next

restore the stack pointer of next

# Source code

```c
/** Offset of `stack' member within `struct thread'.
    Used by switch.S, which can't figure it out on its own. */
uint32_t thread_stack_ofs = offsetof (struct thread, stack);
```

- ***switch_threads(cur, next)***




```asm
.globl switch_threads
.func switch_threads
switch_threads:
        # Save caller's register state.
        #
        # Note that the SVR4 ABI allows us to destroy %eax, %ecx, %edx,
        # but requires us to preserve %ebx, %ebp, %esi, %edi.  See
        # [SysV-ABI-386] pages 3-11 and 3-12 for details.
        #
        # This stack frame must match the one set up by thread_create()
        # in size.
        pushl %ebx
        pushl %ebp
        pushl %esi
        pushl %edi

        # Get offsetof (struct thread, stack).
.globl thread_stack_ofs
        mov thread_stack_ofs, %edx

        # Save current stack pointer to old thread's stack, if any.
```

20

```asm
        movl SWITCH_CUR(%esp), %eax
```

cur

```asm
        movl %esp, (%eax,%edx,1)
```

save the stack pointer of cur

```asm
        # Restore stack pointer from new thread's stack.
```

24

```asm
        movl SWITCH_NEXT(%esp), %ecx
```

next

```asm
        movl (%ecx,%edx,1), %esp
```

restore the stack pointer of next

```asm
        # Restore caller's register state.
        popl %edi
        popl %esi
        popl %ebp
        popl %ebx
        ret
.endfunc
```

# Source code

- ***schedule()***
  - Before calling *schedule()*, make sure **interrupts are disabled**
  - Record the current thread in *cur*
  - Determine the next thread to run *(next)*
  - Call *switch_threads(cur, next)* to do the actual switch
    - Returning from *switch_threads* means ***the thread is scheduled again***
    - Return value is ***the previous thread***
  - Call *thread_schedule_tail(prev)*

```c
static void
schedule (void)
{
  struct thread *cur = running_thread ();
  struct thread *next = next_thread_to_run ();
  struct thread *prev = NULL;

  ASSERT (intr_get_level () == INTR_OFF);
  ASSERT (cur->status != THREAD_RUNNING);
  ASSERT (is_thread (next));

  if (cur != next)
    prev = switch_threads (cur, next);
  thread_schedule_tail (prev);
}
```

# Source code

- *thread_schedule_tail(prev)*
  - Mark the new thread status as *running*
  - Reset the *thread_ticks*
  - If the thread we just switched from (except initial_thread) is in the ***dying*** state, free its thread page

```c
void
thread_schedule_tail (struct thread *prev)
{
  struct thread *cur = running_thread ();

  ASSERT (intr_get_level () == INTR_OFF);

  /* Mark us as running. */
  cur->status = THREAD_RUNNING;

  /* Start new time slice. */
  thread_ticks = 0;

#ifdef USERPROG
  /* Activate the new address space. */
  process_activate ();
#endif

  /* If the thread we switched from is dying, destroy its struct
     thread.  This must happen late so that thread_exit() doesn't
     pull out the rug under itself.  (We don't free
     initial_thread because its memory was not obtained via
     palloc().) */
  if (prev != NULL && prev->status == THREAD_DYING && prev != initial_thread)
    {
      ASSERT (prev != cur);
      palloc_free_page (prev);
    }
}
```

# Source code

- ***next_thread_to_run(void)***
  - **Chooses and returns the next thread to be scheduled**
  - Should return a thread from the run queue, unless the run queue is empty
  - If the run queue is empty, return *idle_thread* which is created by *thread_start()* in *pintos_init()*

```c
static struct thread *
next_thread_to_run (void)
{
  if (list_empty (&ready_list))
    return idle_thread;
  else
    return list_entry (list_pop_front (&ready_list), struct thread, elem);
}
```

# Today

- Lab0 Booting review
- Struct *thread*
- Switching thread
- **Create a new thread**
- Synchronization
- List
- Lab1 tasks
- Q & A

# Create a new thread in Pintos

- Usually, we call ***thread_create*** to create a new thread

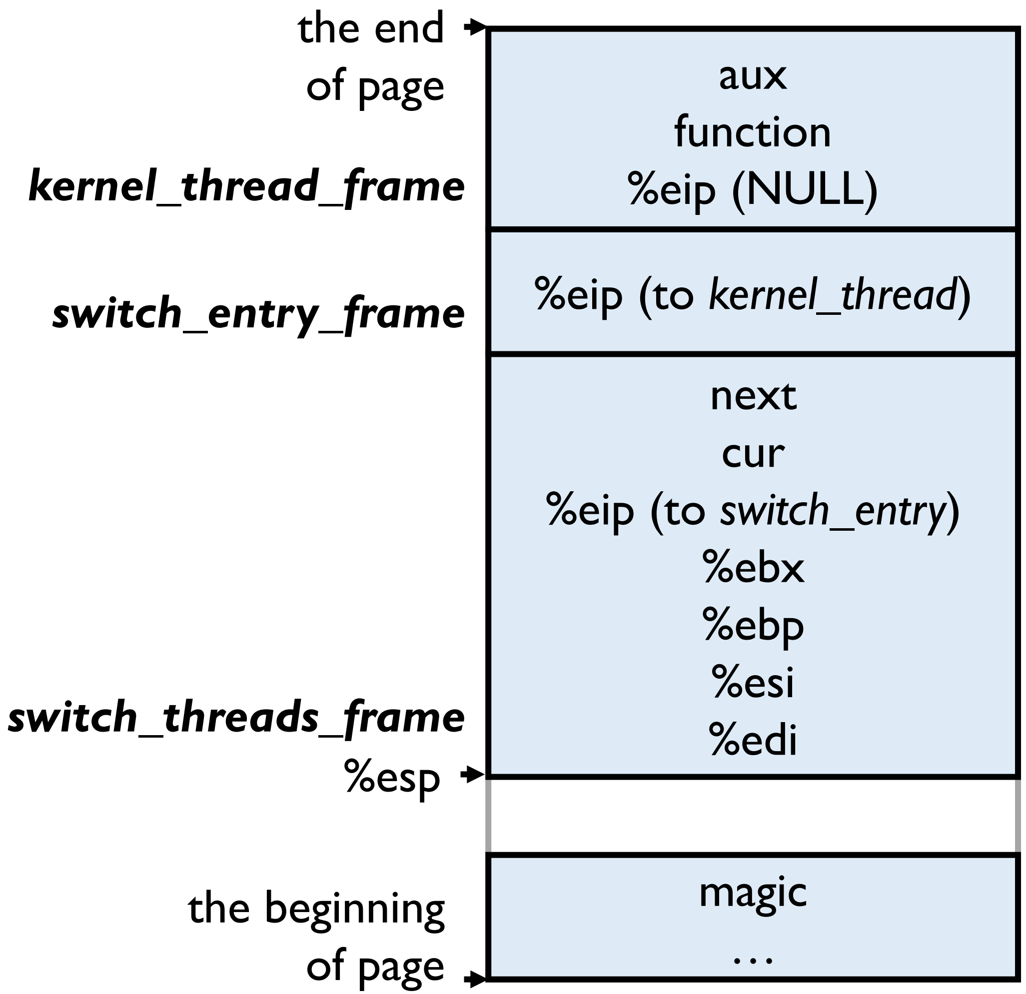
```
typedef void thread_func (void *aux);
tid_t thread_create (const char *name, int priority, thread_func *, void *);
```

- What do we expect ***thread_create*** to do?
  - Thread A calls ***thread_create*** to create thread B and return
  - Thread B starts running when scheduled for the first time, well…
    - *switch_threads* will execute several pop instructions and ret
    - The stack frame for the *thread_func* to run may have not been formed yet
    - ***Leverage some fake stack frames!***

# Create a new thread in Pintos

- Utilize 3 frames: ***kernel_thread_frame, switch_entry_frame, switch_threads_frame***

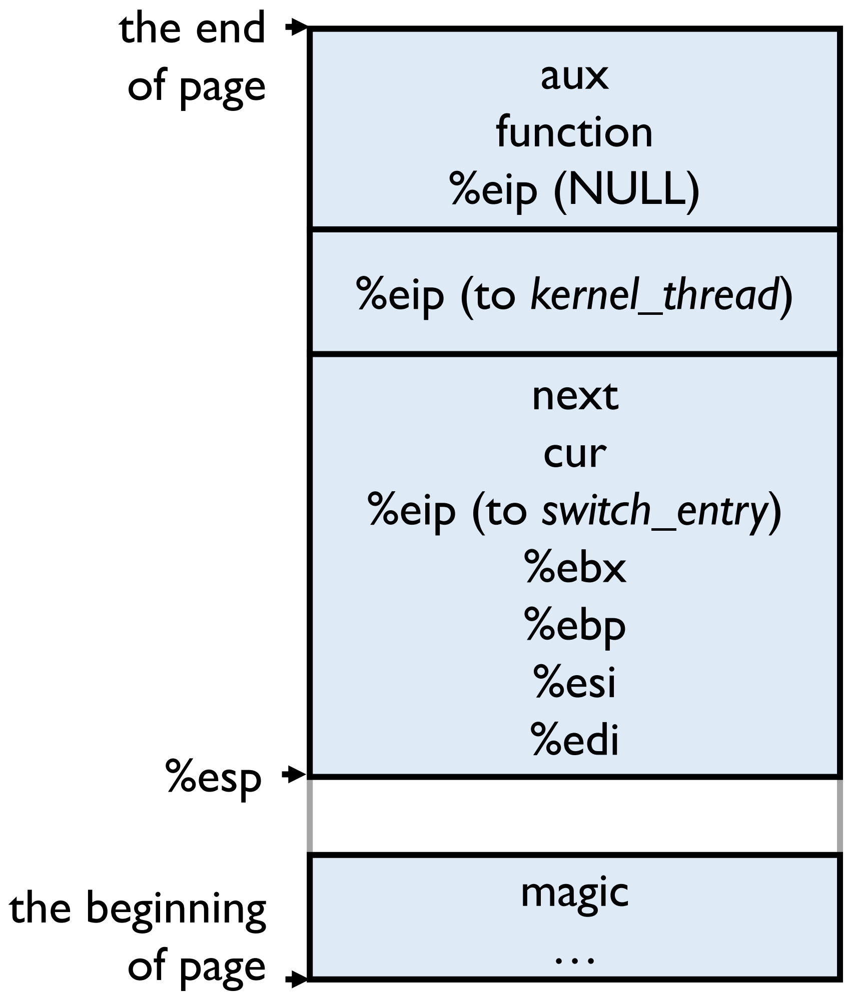
```
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
  struct thread *t;
  struct kernel_thread_frame *kf;
  struct switch_entry_frame *ef;
  struct switch_threads_frame *sf;
  tid_t tid;
```




**thread scheduled for the first time**

# Create a new thread in Pintos

- The thread is scheduled for the first time

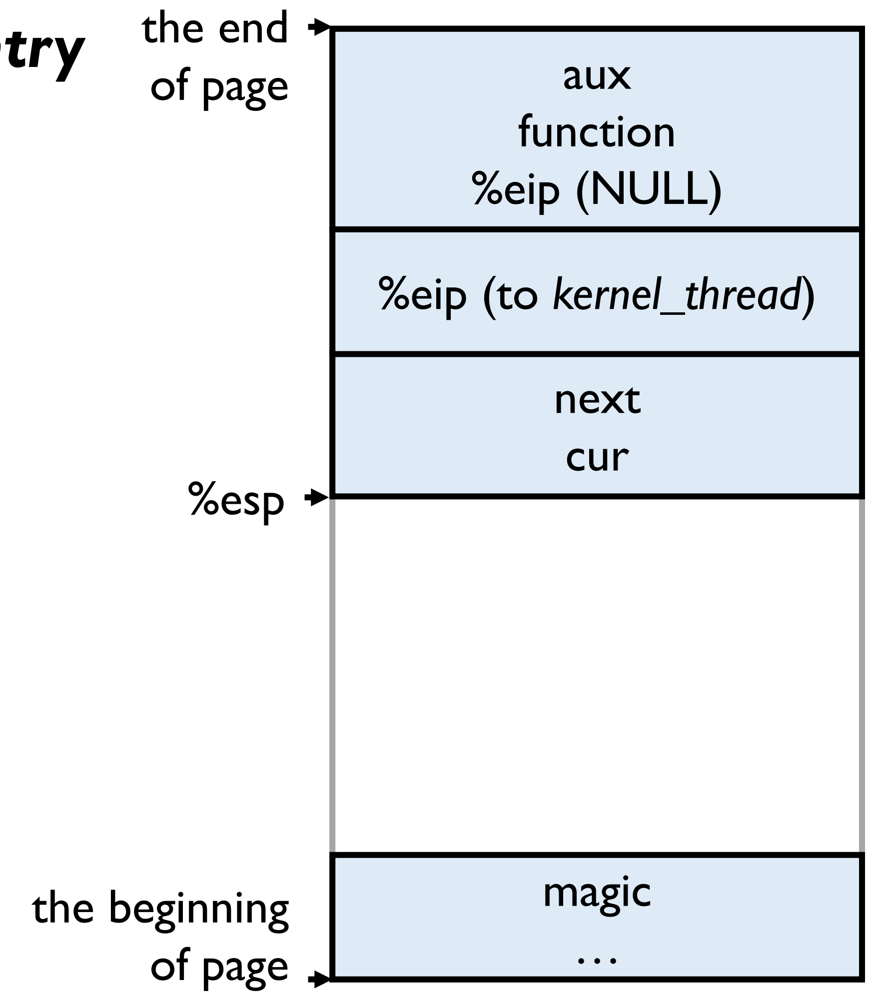


**thread scheduled for the first time**

# Create a new thread in Pintos

- The thread is scheduled for the first time
- Return from ***switch_threads*** *and execute* ***switch_entry***

```
.globl switch_entry
.func switch_entry
switch_entry:
        # Discard switch_threads() arguments.
        addl $8, %esp

        # Call thread_schedule_tail(prev).
        pushl %eax
.globl thread_schedule_tail
        call thread_schedule_tail
        addl $4, %esp

        # Start thread proper.
        ret
.endfunc
```

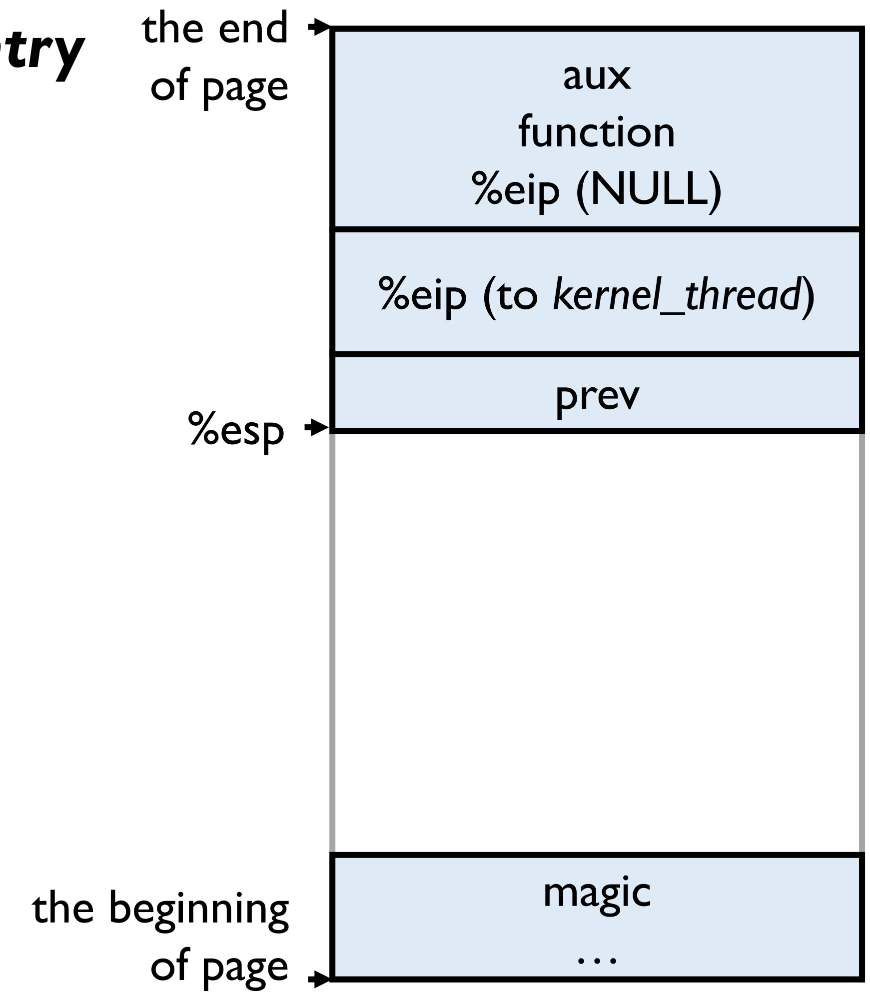



**thread scheduled for the first time**

# Create a new thread in Pintos

- The thread is scheduled for the first time
- Return from ***switch_threads*** *and execute* ***switch_entry***

```
.globl switch_entry
.func switch_entry
switch_entry:
        # Discard switch_threads() arguments.
        addl $8, %esp

        # Call thread_schedule_tail(prev).
        pushl %eax
.globl thread_schedule_tail
        call thread_schedule_tail
        addl $4, %esp

        # Start thread proper.
        ret
.endfunc
```

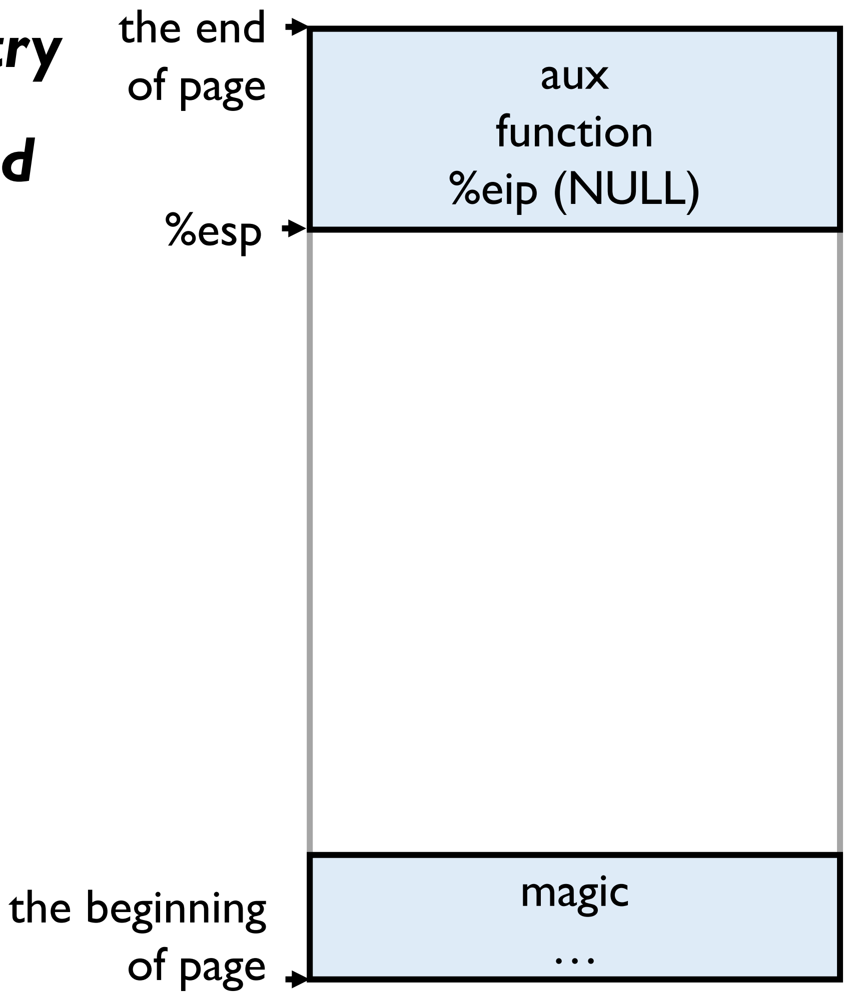



**thread scheduled for the first time**

# Create a new thread in Pintos

- The thread is scheduled for the first time
- Return from ***switch_threads*** *and execute* ***switch_entry***
- Return from ***switch_entry*** *and execute* ***kernel_thread***
    - ***enable interrupt***
    - call ***function(aux)***
    - call ***thread_exit()***

```
/** Function used as the basis for a kernel thread. */
static void
kernel_thread (thread_func *function, void *aux)
{
  ASSERT (function != NULL);

  intr_enable ();       /**< The scheduler runs with interrupts off. */
  function (aux);       /**< Execute the thread function. */
  thread_exit ();       /**< If function() returns, kill the thread. */
}
```




**thread scheduled for the first time**

# Today

- Lab0 Booting review
- Struct *thread*
- Switching thread
- Create a new thread
- **Synchronization**
- List
- Lab1 tasks
- Q & A

# Synchronization

- **disable interrupts**
- **semaphore**
- **lock**
  - based on semaphore
- **condition and monitor**
- **optimization barrier ***

```c
/** A counting semaphore. */
struct semaphore
  {
    unsigned value;             /**< Current value. */
    struct list waiters;        /**< List of waiting threads. */
  };
```

```c
/** Lock. */
struct lock
  {
    struct thread *holder;      /**< Thread holding lock (for debugging). */
    struct semaphore semaphore; /**< Binary semaphore controlling access. */
  };
```

```c
/** Condition variable. */
struct condition
  {
    struct list waiters;        /**< List of waiting threads. */
  };
```

# Today

- Lab0 Booting review
- Struct *thread*
- Switching thread
- Create a new thread
- Synchronization
- **List**
- Lab1 tasks
- Q & A

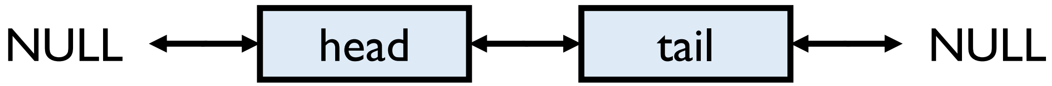
# List in Pintos

- ***doubly linked list***

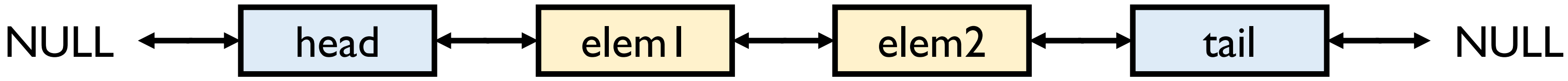
```c
/** List element. */
struct list_elem
  {
    struct list_elem *prev;     /**< Previous list element. */
    struct list_elem *next;     /**< Next list element. */
  };
```

```c
/** List. */
struct list
  {
    struct list_elem head;      /**< List head. */
    struct list_elem tail;      /**< List tail. */
  };
```

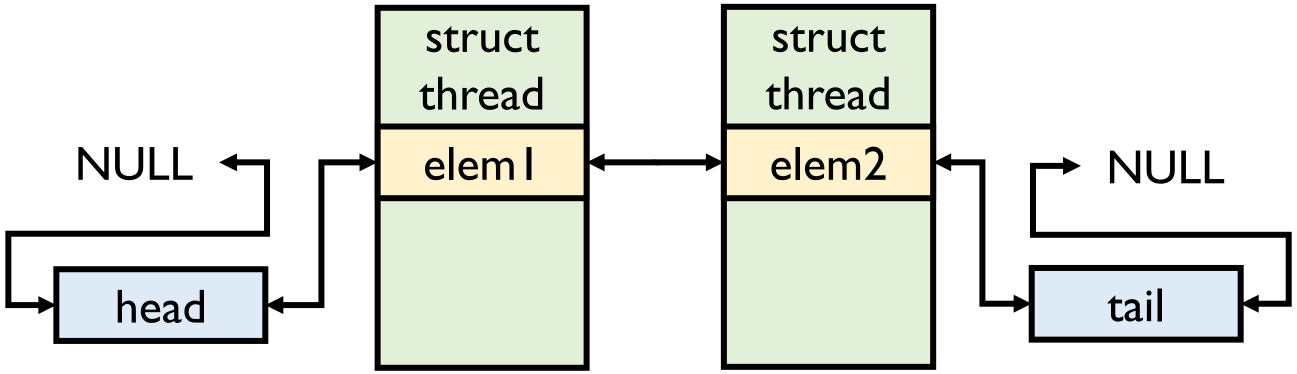
- **an empty list**




- **a list with 2 elements**




- **a list of complex struct**
  - embed a *list_elem* in the struct
  - manipulate the inner *list_elem*



# List in Pintos

- **example: ready_list**

```c
void
thread_yield (void)
{
  struct thread *cur = thread_current ();
  enum intr_level old_level;

  ASSERT (!intr_context ());

  old_level = intr_disable ();
  if (cur != idle_thread)
    list_push_back (&ready_list, &cur->elem);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

```c
static struct thread *
next_thread_to_run (void)
{
  if (list_empty (&ready_list))
    return idle_thread;
  else
    return list_entry (list_pop_front (&ready_list), struct thread, elem);
}
```

- ***list_entry***: convert pointer to LIST_ELEM into the pointer to the struct containing it

```c
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /**< Thread identifier. */
    enum thread_status status;          /**< Thread state. */
    char name[16];                      /**< Name (for debugging purposes). */
    uint8_t *stack;                     /**< Saved stack pointer. */
    int priority;                       /**< Priority. */
    struct list_elem allelem;           /**< List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /**< List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /**< Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /**< Detects stack overflow. */
  };
```

```c
/** List of processes in THREAD_READY state, that is, processes
   that are ready to run but not actually running. */
static struct list ready_list;
```

```c
#define list_entry(LIST_ELEM, STRUCT, MEMBER)           \
        ((STRUCT *) ((uint8_t *) &(LIST_ELEM)->next     \
                     - offsetof (STRUCT, MEMBER.next)))
```

# List in Pintos

- **some handy interfaces**
  - *lib/kernel/list.h*

```c
/** List traversal. */
struct list_elem *list_begin (struct list *);
struct list_elem *list_next (struct list_elem *);
struct list_elem *list_end (struct list *);

struct list_elem *list_rbegin (struct list *);
struct list_elem *list_prev (struct list_elem *);
struct list_elem *list_rend (struct list *);

struct list_elem *list_head (struct list *);
struct list_elem *list_tail (struct list *);

/** List insertion. */
void list_insert (struct list_elem *, struct list_elem *);
void list_splice (struct list_elem *before,
                  struct list_elem *first, struct list_elem *last);
void list_push_front (struct list *, struct list_elem *);
void list_push_back (struct list *, struct list_elem *);

/** List removal. */
struct list_elem *list_remove (struct list_elem *);
struct list_elem *list_pop_front (struct list *);
struct list_elem *list_pop_back (struct list *);

/** List elements. */
struct list_elem *list_front (struct list *);
struct list_elem *list_back (struct list *);
```

```c
/** List properties. */
size_t list_size (struct list *);
bool list_empty (struct list *);

/** Miscellaneous. */
void list_reverse (struct list *);

/** Compares the value of two list elements A and B, given
   auxiliary data AUX.  Returns true if A is less than B, or
   false if A is greater than or equal to B. */
typedef bool list_less_func (const struct list_elem *a,
                             const struct list_elem *b,
                             void *aux);

/** Operations on lists with ordered elements. */
void list_sort (struct list *,
                list_less_func *, void *aux);
void list_insert_ordered (struct list *, struct list_elem *,
                          list_less_func *, void *aux);
void list_unique (struct list *, struct list *duplicates,
                  list_less_func *, void *aux);

/** Max and min. */
struct list_elem *list_max (struct list *, list_less_func *, void *aux);
struct list_elem *list_min (struct list *, list_less_func *, void *aux);
```

# Today

- Lab0 Booting review
- Struct *thread*
- Switching thread
- Create a new thread
- Synchronization
- List
- **Lab1 tasks**
- Q & A

# Lab1 Task 1: Alarm Clock

**Exercise 1.1**

Reimplement **`timer_sleep()`**, defined in `devices/timer.c`.

- Although a working implementation is provided, it "busy waits," that is, it spins in a loop checking the current time and calling `thread_yield()` until enough time has gone by.
- Reimplement it to **avoid busy waiting**.

- ***hint***
  - Synchronization
  - Read FAQ

```c
/** Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);
  while (timer_elapsed (start) < ticks)
    thread_yield ();
}
```

# Lab1 Task 2: Priority Scheduling

**Exercise 2.1**

**Implement *priority scheduling* in Pintos.**

- When a thread is added to the ready list that has a higher priority than the currently running thread, the current thread should **immediately yield** the processor to the new thread.
- Similarly, when threads are waiting for a lock, semaphore, or condition variable, **the highest priority waiting thread should be awakened first**.
- A thread may raise or lower its own priority at any time, but **lowering its priority such that it no longer has the highest priority must cause it to immediately yield the CPU**.

- ***hint***
  - Consider ***all the scenarios*** where the priority must be enforced
  - 2 types of ***yield***
  - Read FAQ

# Lab1 Task 2: Priority Scheduling

**Exercise 2.2.1**

**Implement priority donation.**

- You will need to **account for all different situations in which priority donation is required**.
- You must implement priority donation **for locks**. You need **not** implement priority donation for the other Pintos synchronization constructs.
- You do need to **implement *priority scheduling* in all cases**.
- Be sure to **handle multiple donations**, in which multiple priorities are donated to a single thread.

- ***hint***
  - Implement ***priority donation only for locks*** but ***priority scheduling for all!***
  - Read FAQ

# Lab1 Task 2: Priority Scheduling

**Exercise 2.2.2**

**Support nested priority donation:**

- if H is waiting on a lock that M holds and M is waiting on a lock that L holds, then both M and L should be boosted to H's priority.
- If necessary, you may impose **a reasonable limit** on depth of nested priority donation, such as 8 levels.

- ***hint***
  - Implement ***priority donation only for locks*** but ***priority scheduling for all!***
  - Read FAQ

# Lab1 Task 2: Priority Scheduling

**Exercise 2.3**

**Implement the following functions that allow a thread to examine and modify its own priority.**

Skeletons for these functions are provided in **threads/thread.c**.

- **Function: void thread_set_priority (int new_priority)**
  - **Sets the current thread's priority to *new_priority*.**
  - If the current thread no longer has the highest priority, **yields**.
- **Function: int thread_get_priority (void)**
  - **Returns the current thread's priority.** In the presence of priority donation, returns the higher (donated) priority.

- ***hint***
  - Implement ***priority donation only for locks*** but ***priority scheduling for all!***
  - Read FAQ

# Lab1 Task 3: Advanced Scheduler

> **Exercise 3.1**
>
> **Implement a multilevel feedback queue scheduler** similar to the 4.4BSD scheduler to reduce the average response time for running jobs on your system.
>
> See section **4.4BSD Scheduler**, for detailed requirements.

- ***hint***
  - When to update
  - How to update
  - Read FAQ

# About tests

- **All the test code in lab1 run as part of the kernel**
- **Efficiency Problem**
  - esp. mlfqs-load-60, mlfqs-load-avg
  - *load_avg* is dynamically updated and **sensitive to the some properties**, e.g., the number of threads that are either running or ready to run at time of update
  - If your implementation is not efficient, then it will effect the system properties and the final output
  - The acceptable error range for *load_avg* is within ±2.5 which can be known from *.ck* file.

```
mlfqs_compare ("time", "%.2f", \@actual, \@expected, 2.5, [2, 178, 2],
               "Some load average values were missing or "
               . "differed from those expected "
               . "by more than 2.5.");
```

```
The expected output is this (some margin of error is allowed):

After 0 seconds, load average=1.00.
After 2 seconds, load average=2.95.
After 4 seconds, load average=4.84.
After 6 seconds, load average=6.66.
After 8 seconds, load average=8.42.
After 10 seconds, load average=10.13.
After 12 seconds, load average=11.78.
After 14 seconds, load average=13.37.
After 16 seconds, load average=14.91.
After 18 seconds, load average=16.40.
After 20 seconds, load average=17.84.
After 22 seconds, load average=19.24.
After 24 seconds, load average=20.58.
After 26 seconds, load average=21.89.
After 28 seconds, load average=23.15.
After 30 seconds, load average=24.37.
After 32 seconds, load average=25.54.
After 34 seconds, load average=26.68.
After 36 seconds, load average=27.78.
After 38 seconds, load average=28.85.
After 40 seconds, load average=29.88.
After 42 seconds, load average=30.87.
After 44 seconds, load average=31.84.
After 46 seconds, load average=32.77.
After 48 seconds, load average=33.67.
After 50 seconds, load average=34.54.
```

# Today

- Lab0 Booting review
- Struct *thread*
- Switching thread
- Create a new thread
- Synchronization
- List
- Lab1 tasks
- **Q & A**

Thanks for
your listening.