

# CS 318 Principles of Operating Systems

Fall 2021

## Pintos Virtual Memory Notes

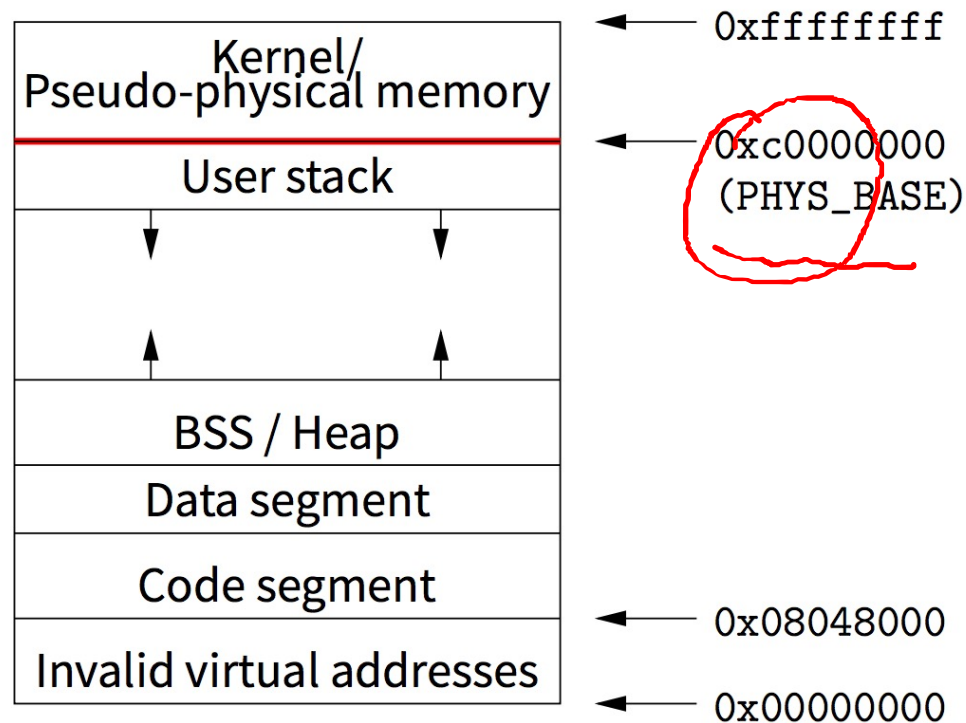
**Prof. Ryan Huang**



JOHNS HOPKINS

WHITING SCHOOL  
of ENGINEERING

# Pintos Virtual Memory Layout



**i.e., the kernel lives in every process' address space!**

## A process' virtual address space is split into two regions

- The kernel lives in the high memory region, typically highest 1 GB, i.e., from 3 to 4 GB.
- The user memory lives in the lower region, typically lower 3 GB, i.e., from 0 to 3 GB.

# User Virtual Memory

*Per process: a new page directory (pagedir) for each process*

```
struct thread
{
    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. */
    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. */
};
```

# How Is A User Process Started?

```
$ pintos -p ../../examples/echo -a echo -- -f -q run 'echo cs318'
```

```
static void
run_task (char **argv)
{
    const char *task = argv[1];

    printf ("Executing '%s':\n", task);
#ifdef USERPROG
    process_wait (process_execute (task));
#else
    run_test (task);
#endif
    printf ("Execution of '%s' complete.\n", task);
}
```

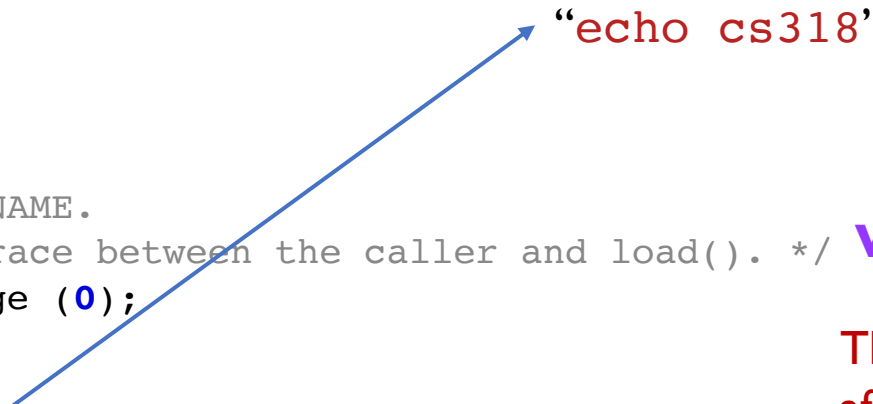
# How Is A User Process Started?

```
$ pintos -p ../../examples/echo -a echo -- -f -q run 'echo cs318'
```

```
tid_t
process_execute (const char *file_name)
{
    char *fn_copy;
    tid_t tid;

    /* Make a copy of FILE_NAME.
       Otherwise there's a race between the caller and load(). */
    fn_copy = palloc_get_page (0);
    if (fn_copy == NULL)
        return TID_ERROR;
    strcpy (fn_copy, file_name, PGSIZE);

    /* Create a new thread to execute FILE_NAME. */
    tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy);
    if (tid == TID_ERROR)
        palloc_free_page (fn_copy);
    return tid;
}
```



**Why?**

The caller might free the `file_name` after this function returns!

,e.g., after you implement ``exec``.

# How Is A User Process Started?

```
$ pintos -p ../../examples/echo -a echo -- -f -q run 'echo cs318'
```

```
static void
start_process (void *file_name_)
{
    char *file_name = file_name_;
    struct intr_frame if_;
    bool success;

    /* Initialize interrupt frame and load executable. */
    memset (&if_, 0, sizeof if_);
    if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
    if_.cs = SEL_UCSEG;
    if_.eflags = FLAG_IF | FLAG_MBS;
    success = load (file_name, &if_.eip, &if_.esp);

    /* If load failed, quit. */
    palloc_free_page (file_name);
    if (!success)
        thread_exit ();

    /* Start the user process by simulating a return from an interrupt */
    asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
    NOT_REACHED ();
}
```

# How Is A User Process Started?

```
$ pintos -p ../../examples/echo -a echo -- -f -q run 'echo cs318'
```

```
bool load (const char *file_name, void (**eip) (void), void **esp)
{
    struct thread *t = thread_current ();
    ...
```

```
/* Allocate and activate page directory. */
```

```
t->pagedir = pagedir_create ();
```

```
if (t->pagedir == NULL)
```

```
    goto done;
```

```
process_activate ();
```

```
/* Open executable file. */
```

```
file = filesys_open (file_name);
```

```
...
```

```
}
```

```
void process_activate (void)
```

```
{
```

```
    struct thread *t = thread_current ();
```

```
    pagedir_activate (t->pagedir);
```

```
/* Set thread's kernel stack for use in processing interrupts. */
```

```
tss_update ();
```

```
void pagedir_activate (uint32_t *pd)
```

```
{
```

```
    if (pd == NULL)
```

```
        pd = init_page_dir;
```

```
    asm volatile ("movl %0, %%cr3" : : "r" (vtop (pd)) : "memory");
```

```
}
```

After this point, the user virtual  
memory mappings changed!



# Wait, ...

We just changed the user virtual memory mappings, how is it OK for us to still access these variables we created earlier, e.g., `file_name`?

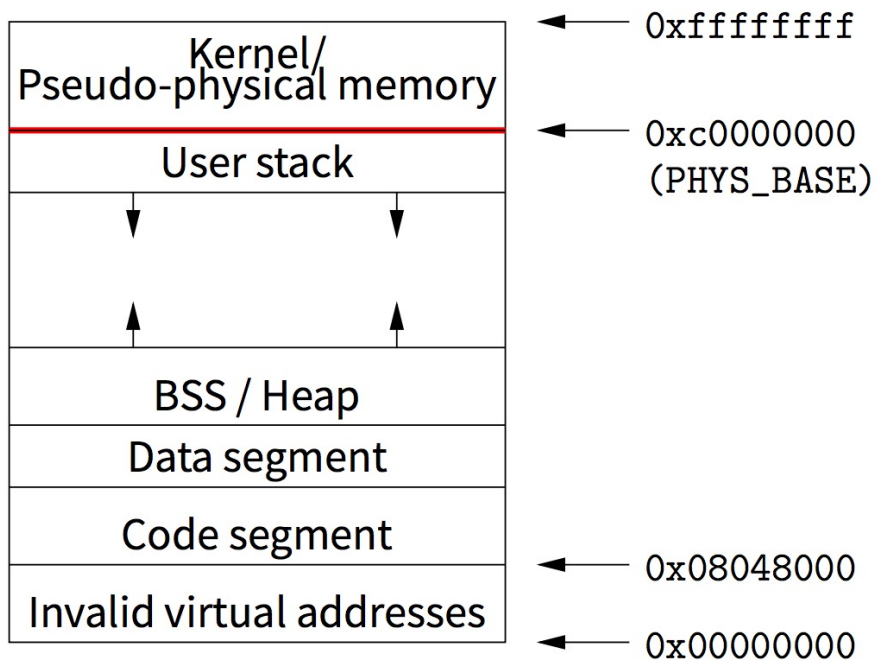
**A related concern: how to access variables across multiple processes?**

- e.g., to implement `int wait (pid_t pid)` you want to create a variable in `struct thread` to store some information for a process,
  - e.g., `thread->wait_status`,
- but how can you read/write this variable from the parent process?



# Answer: We're in the Kernel!

The kernel virtual memory mappings are the same across all processes

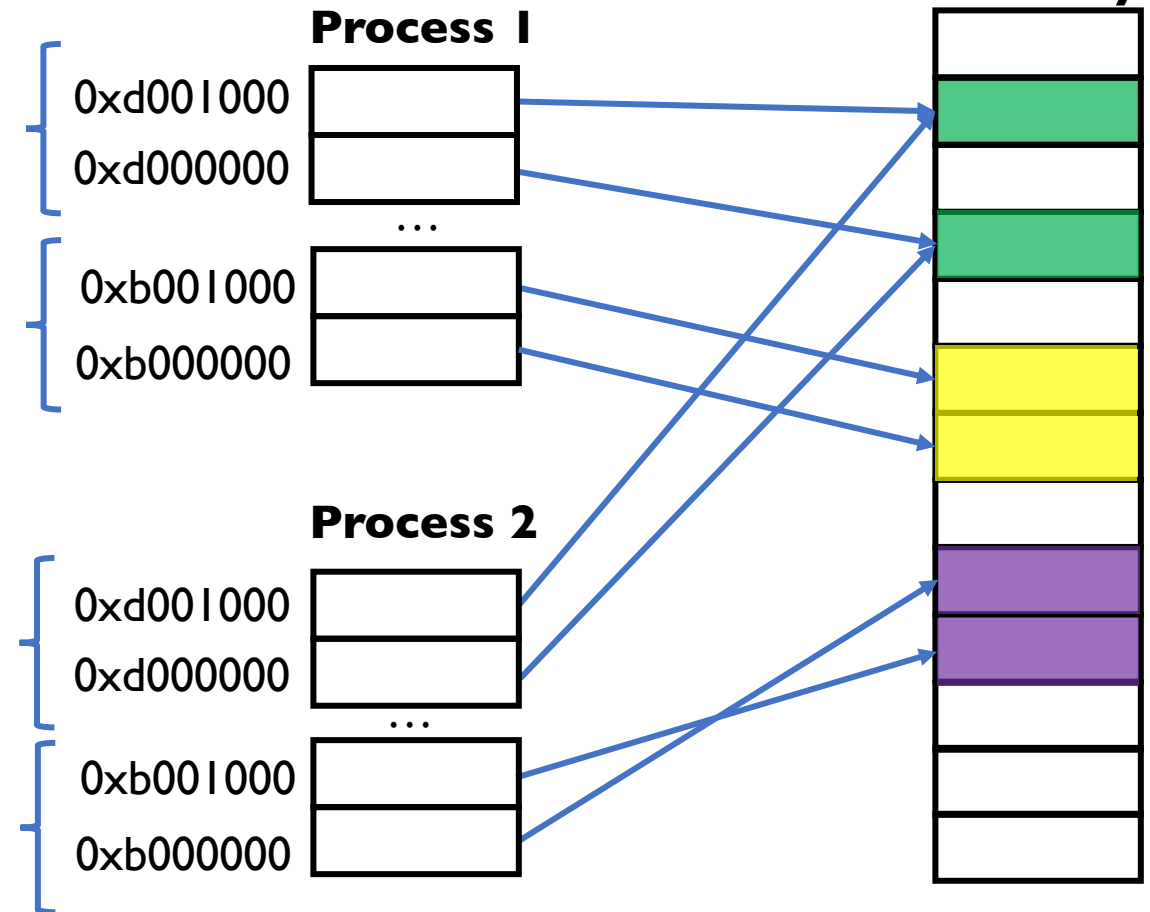


Kernel memory region

User memory region

Kernel memory region

User memory region



# Answer: *We're in the Kernel!*

The kernel virtual memory mappings are the same across all processes

## Implications:

- When we context switch to another process, although it involves changing the page tables, the kernel virtual memory addresses are still valid after the switch
- All objects created in the kernel functions are accessible across processes
  - e.g., `static struct list all_list;`      `threadX->wait_status`
- Memory for user processes will be freed when a user process exits, but memory objects allocated within the kernel code using `malloc` should be explicitly freed!

# How Is This Implemented?

```
bool load (const char *file_name, void (**eip) (void), void **esp)
{
    struct thread *t = thread_current ();
    ...
    /* Allocate and activate page directory. */
    t->pagedir = pagedir_create ();
    if (t->pagedir == NULL)
        goto done;
    process_activate ();

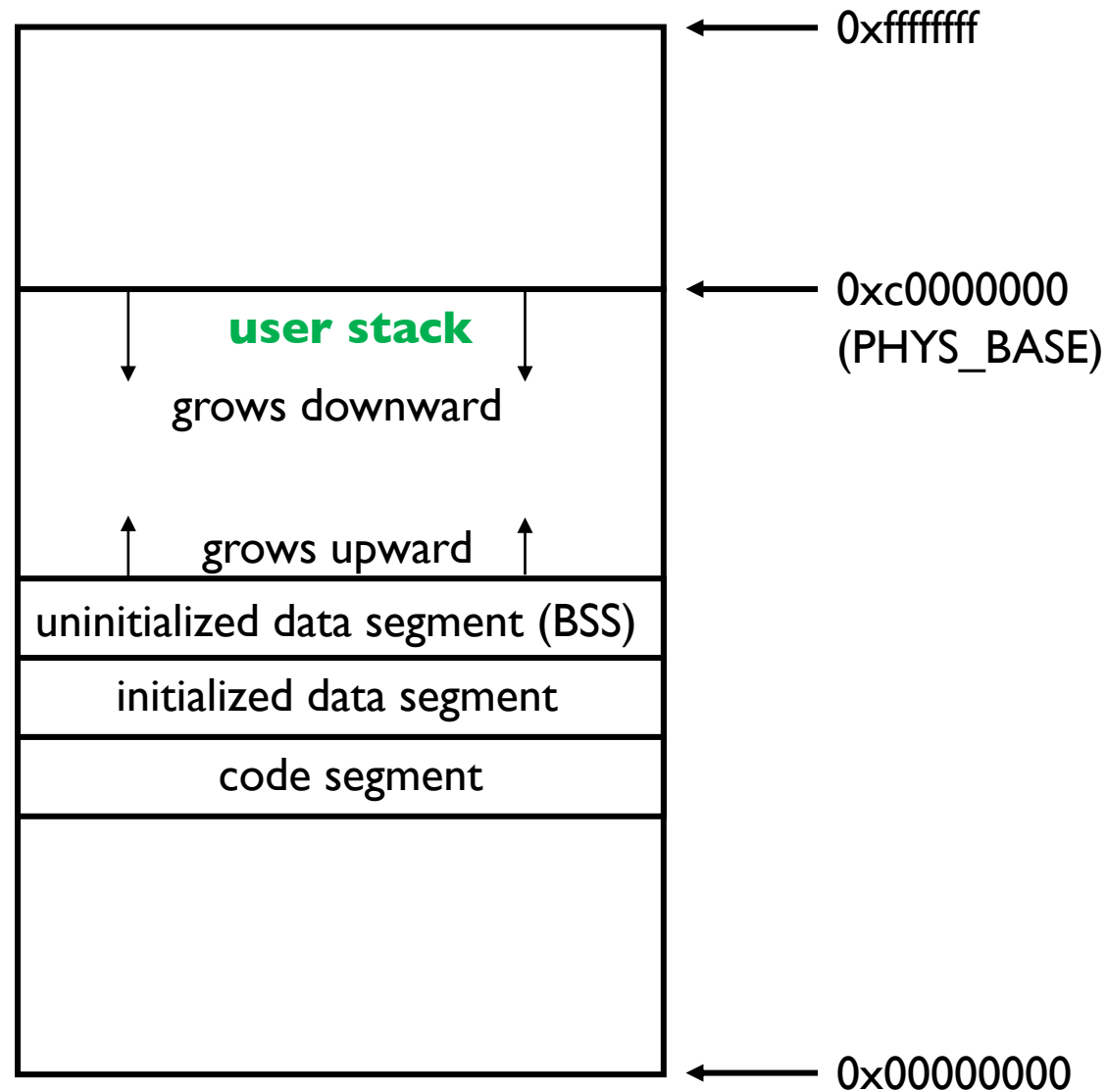
    /* Open executable file. */
    file = filesys_open (file_name);
    ...
}
```

```
uint32_t *
pagedir_create (void)
{
    uint32_t *pd = palloc_get_page (0);
    if (pd != NULL)
        memcpy (pd, init_page_dir, PGSIZE);
    return pd;
}
```

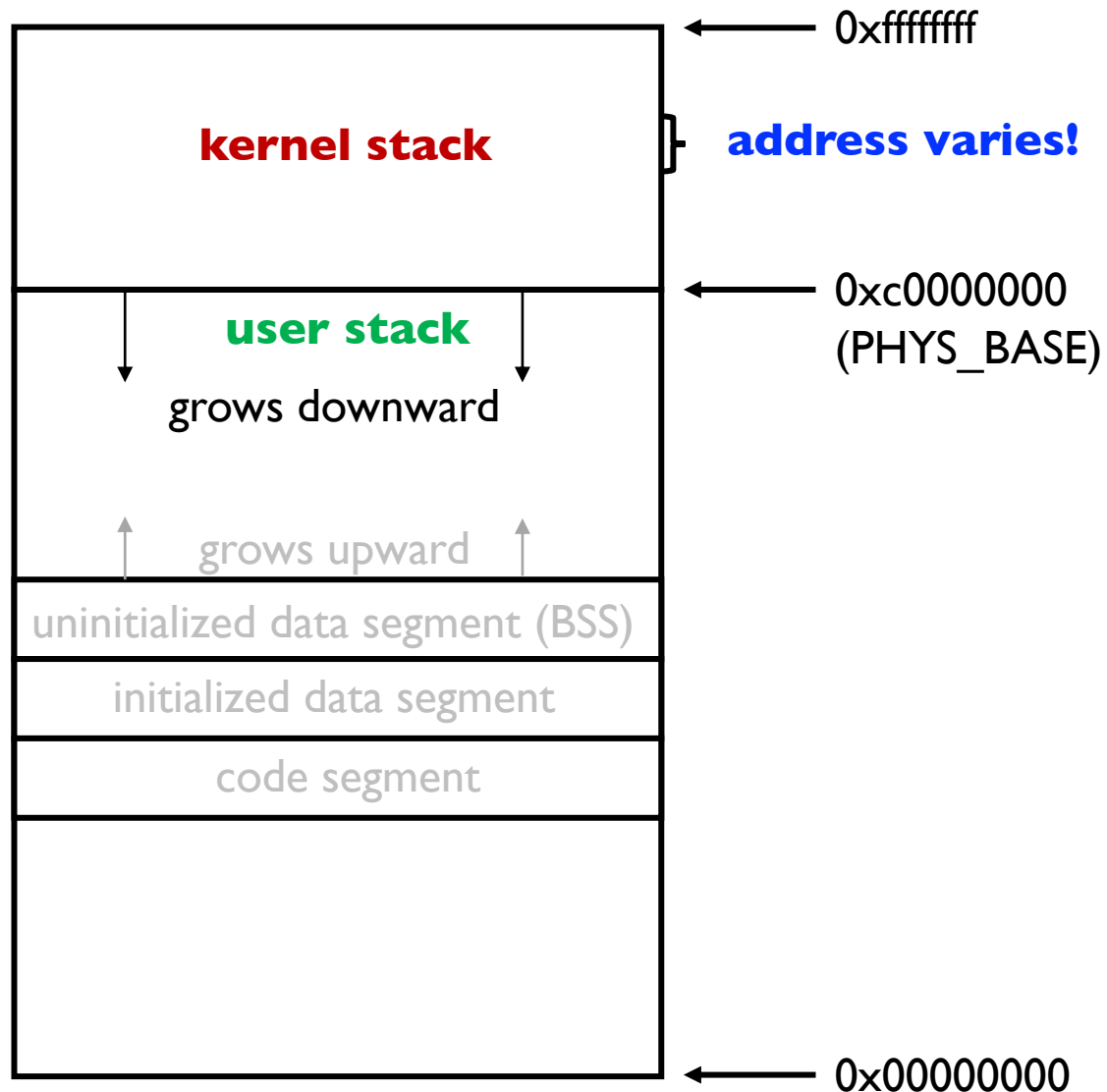
Initialized in [paging\\_init\(\) in thread.c](#)



# User Stack vs Kernel Stack



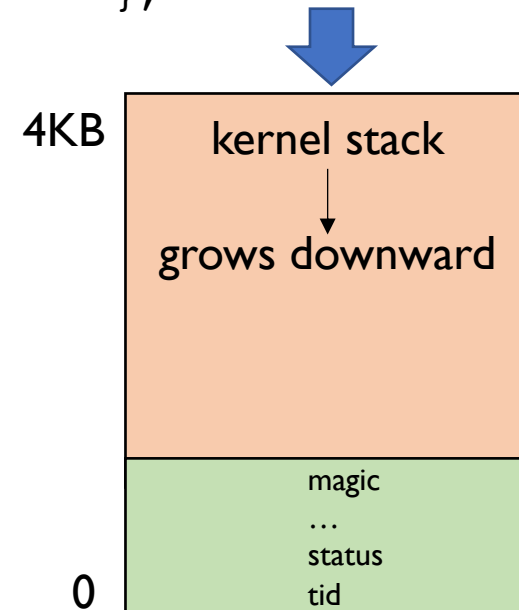
# User Stack vs Kernel Stack



**A new kernel thread gets a new kernel stack**

```

struct thread {
    tid_t tid;
    enum thread_status status;
    ...
    unsigned magic;
};
    
```



# Lab 2

## Minimal changes to get started:

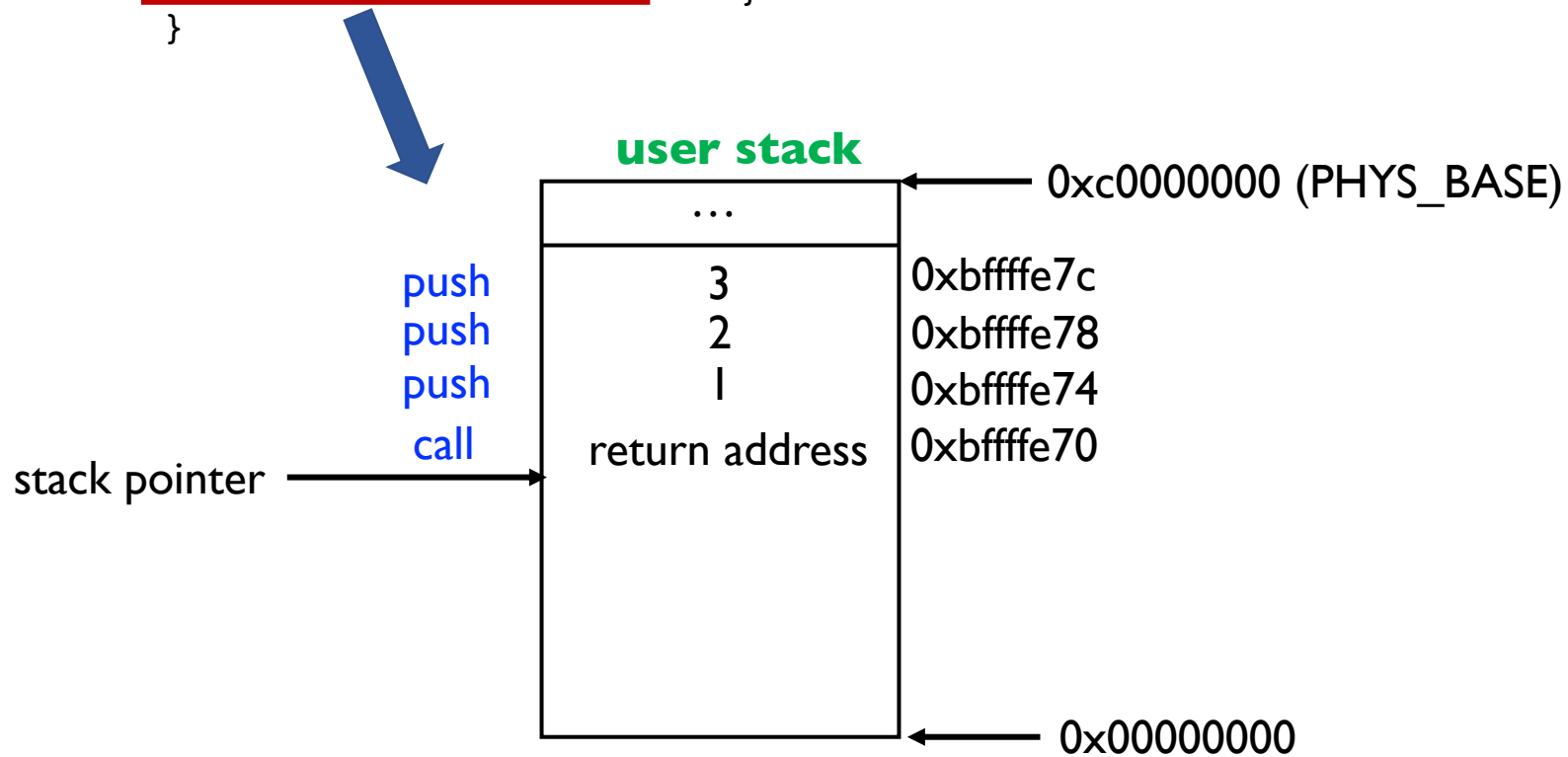
1. `setup_stack()`: `*esp = PHYS_BASE;` → `*esp = PHYS_BASE - 12;`
2. change `process_wait()` to an infinite loop

# Why setting `esp` to `PHYS_BASE - 12` ?

## A temporary setup for obeying x86 calling convention

```
void bar()
{
    int ret;
    ret = foo(1, 2, 3);
}

int foo(int a, int b, int c)
{
    ...
}
```

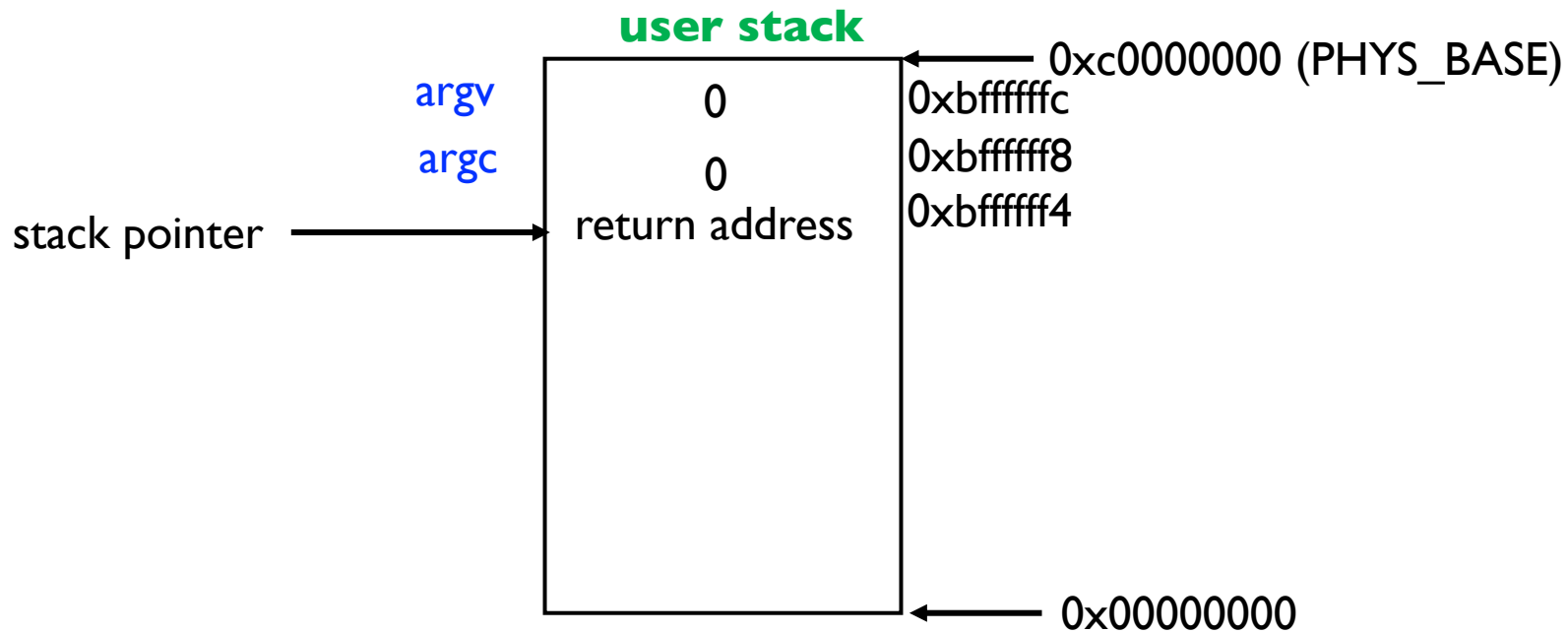


# Why setting `esp` to `PHYS_BASE - 12` ?

## A temporary setup for obeying x86 calling convention

- Every user program's entry point is:

```
void _start (int argc, char *argv[])  
{  
    exit (main (argc, argv));  
}
```
- minimal 3 elements on user stack, each 4 bytes = 12





# Why setting `esp` to `PHYS_BASE - 12` ?

## A temporary setup for obeying x86 calling convention

- Every user program's entry point is:

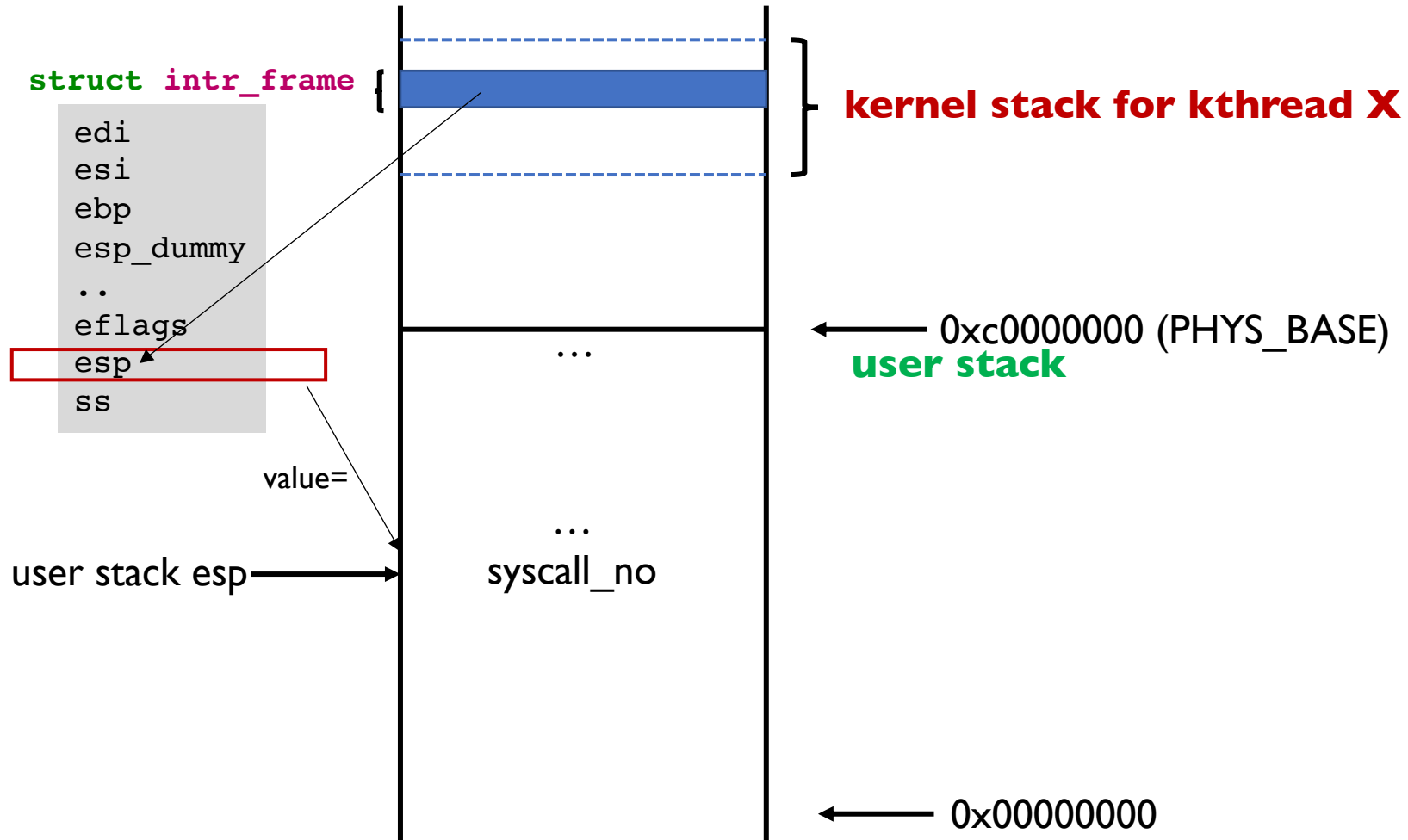
```
void _start (int argc, char *argv[])  
{  
    exit (main (argc, argv));  
}
```
- minimal 3 elements on user stack, each 4 bytes = 12

## Note: this is only a temporary setup

- Once you implement argument parsing, you should set `esp` correctly based on the actual arguments pushed on the user stack

# System Call

## Through trap (an interrupt frame)



How to retrieve the syscall no in syscall\_handler?

from reading user memory at intr\_frame->esp

# User Memory Access

## Upon system call, *no* page directory switch

- i.e., in `sycall_handler`, the kernel can directly access user memory by dereferencing it
- However, must carefully check each user memory address for robustness!

## Two approaches for checking + accessing user memory

- Hardware approach: leveraging page fault to detect invalid address
- Software approach: using `pagedir` methods to check validity of an address

# Hardware Approach

Try loading the memory from a given address `addr`

- Assume `addr` is the function argument

```
movl 4(%esp), %edx;    ==> edx = addr  
movzbl (%edx), %eax;   ==> eax = [addr]
```

- **Problem:** we'll get a page fault if `addr` is invalid
- **Idea:** let page fault handler inform us, how?

# Hardware Approach

## Use the given helper function, modify page fault handler

```
/* Reads a byte at user virtual address UADDR.  
   UADDR must be below PHYS_BASE.  
   Returns the byte value if successful, -1 if a segfault  
   occurred. */
```

```
static int get_user (const uint8_t *uaddr)  
{  
    int result;  
    asm ("movl $1f, %0; movzbl %1, %0; 1:"  
        : "=a" (result) : "m" (*uaddr));  
    return result;  
}
```

compilation  
→

```
get_user:  
    movl 4(%esp), %edx;  
    movl $1f, %eax;  
    movzbl (%edx), %eax;  
    1:  
    ret
```

- If addr is valid, eax has the value
- If addr is invalid, the page fault handler will
  - set eip to address of label 1 (stored in eax now)
  - set eax to be -1 (0xffffffff);
  - resume to `ret`

# Hardware Approach

Use the given helper function, modify page fault handler

```
/* Reads a byte at user virtual address UADDR.
   UADDR must be below PHYS_BASE.
   Returns the byte value if successful, -1 if a segfault
   occurred. */
static int get_user (const uint8_t *uaddr)
{
    int result;
    asm ("movl $1f, %0; movzbl %1, %0; 1:"
         : "=a" (result) : "m" (*uaddr));
    return result;
}
```

But what if the value at `uaddr` is `-1`? We can't tell if it's invalid or not!

- **Solution:** read one byte at a time!
  - If value is valid, at most can be 255 (0xff).
  - How to represent a valid `-1`? Read four bytes (call `get_user` four times), convert them to an integer!