CS 318 Principles of Operating Systems

Pintos Virtual Memory Notes

Prof. Ryan Huang
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.

---

### Pintos Virtual Memory Layout

<table>
<thead>
<tr>
<th>Kernel/ Pseudo-physical memory</th>
<th>0xffffffff</th>
</tr>
</thead>
<tbody>
<tr>
<td>User stack</td>
<td>0xc0000000 (PHYS_BASE)</td>
</tr>
<tr>
<td>BSS / Heap</td>
<td>0x08048000</td>
</tr>
<tr>
<td>Data segment</td>
<td>0x00000000</td>
</tr>
<tr>
<td>Code segment</td>
<td></td>
</tr>
<tr>
<td>Invalid virtual addresses</td>
<td></td>
</tr>
</tbody>
</table>

i.e., the kernel lives in every process’ address space!
User Virtual Memory

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

```c
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?

```c
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?

How is a user process started in Pintos? To start a user process, the kernel will execute the `process_execute` function. This function takes a file name as an argument and starts a new thread to execute the program specified in the file name.

```c
void 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;
    strlcpy(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;
}
```

The caller might free the `file_name` after this function returns! For example, after you implement `exec`, you might want to run the command `echo cs318`:

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

Why?

- The caller might free the `file_name` after this function returns!
- e.g., after you implement `exec`.

The `palloc` function is used to allocate memory for the process. If the allocation fails, the process is returned with an error code.`

10/7/21
How Is A User Process Started?

```c
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 ();
}
```

$pintos -p ../../../examples/echo -a echo -- -f -q run 'echo cs318'
How Is A User Process Started?

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

```c
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 pagedir_activate (uint32_t *pd)
{
    if (pd == NULL)
        pd = init_page_dir;
    asm volatile ("movl %0, %%cr3" : : "r" (vtop (pd)) : "memory");
}

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 ();
}
```

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?
The kernel virtual memory mappings are the same across all processes.
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?

```c
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 grows downward
- Uninitialized data segment (BSS) grows upward
- Initialized data segment
- Code segment

```
0xffffff
```

```
0xc0000000
PHYS_BASE
```

```
0x00000000
```

```
0x00000000
```
User Stack vs Kernel Stack

- Kernel stack: grows downward
- User stack: grows upward
- Initialized data segment
- Uninitialized data segment (BSS)
- Code segment

0xc0000000 (PHYS_BASE)

A new kernel thread gets a new kernel stack

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

Kernel stack grows downward

- Magic
- Status
- Tid
Minimal changes to get started:

1. `setup_stack()`:  
   
   \[ \text{*esp} = \text{PHYS\_BASE}; \quad \Rightarrow \quad \text{*esp} = \text{PHYS\_BASE} - 12; \]

2. `change process_wait()` to an infinite loop
Why setting $\text{esp}$ to $\text{PHYS\_BASE} - 12$?

A temporary setup for obeying x86 calling convention

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

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

User stack:

- Stack pointer
- Push operations
- Call
- Return address

```
    ... 0xc0000000 (PHYS_BASE)
    3
    2
    1
    return address 0xbffffe7c
    0xbffffe78
    0xbffffe74
    0xbffffe70
```

Stack pointer: 0x00000000

Return address: 0xbffffe7c

Call: 0xc0000000 (PHYS_BASE)
Why setting `esp` to `PHYS_BASE - 12`?

A temporary setup for obeying x86 calling convention

- Every user program’s entry point is:

```c
void _start (int argc, char *argv[])
{
    exit (main (argc, argv));
}
```

- minimal 3 elements on user stack, each 4 bytes = 12
Why setting $\text{esp}$ to $\text{PHYS\_BASE} - 12$?

A temporary setup for obeying x86 calling convention

- Every user program’s entry point is:

  ```c
  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 $\text{esp}$ correctly based on the actual arguments pushed on the user stack
System Call

Through trap (an interrupt frame)

- struct intr_frame
  - edi
  - esi
  - ebp
  - esp_dummy
  - ... eflags
  - esp
  - ss

How to retrieve the syscall no in syscall_handler from reading user memory at intr_frame->esp

kernel stack for kthread X

0xc0000000 (PHYS_BASE)

user stack

syscall_no

0x00000000

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 syscall_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 \texttt{addr}

- Assume \texttt{addr} is the function argument

\[
\begin{align*}
\text{movl } &4(\%esp), \%edx; & \quad \text{edx} = \texttt{addr} \\
\text{movzbl } &(%edx), \%eax; & \quad \text{eax} = [\texttt{addr}] \\
\end{align*}
\]

- **Problem:** we’ll get a page fault if \texttt{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;
}
Hardware Approach

Use the given helper function, modify page fault handler

Declaring a function to read a byte at user virtual address:

```c
/* 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!