Operating Systems Lab
Part 2: User Programs

KAIST

Youjip Won

Overview

- Objective

  Execute a user program in Pintos.

- Background
- Topics
  - Parameter Passing
  - System call infrastructure
  - File manipulation

Background

To run a program

- Read the executable file from the disk.
  - Filesystem issue
- Allocate memory for the program to run.
  - Virtual memory allocation
- Pass the parameters to the program.
  - Set up user stack.
- Context switch to the user program
  - OS should wait for the program to exit.

Pintos filesystem

- Create virtual disk: in `userprog/build`

```
pintos-mkdisk filesys.dsk --filesys-size=2
```

  - filesys.dsk: partition name
  - Filesystem size: 2MByte

- Format the disk

```
pintos –f –q
```

- Copy the file to the pintos filesystem
  - -p: put, -g: get, -a: target filename

```
pintos –p ../../examples/echo –a echo -- -q
```

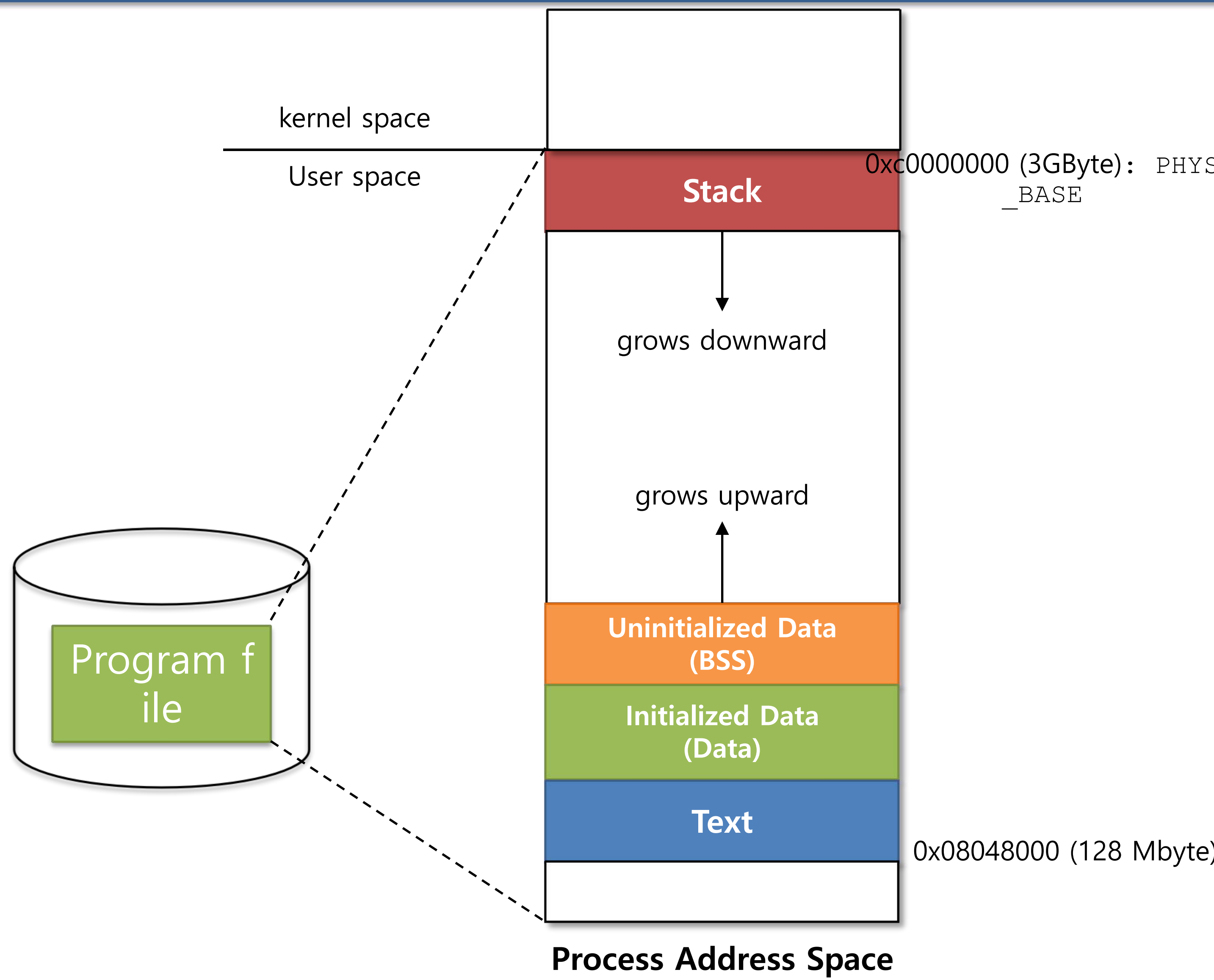
- Run the program

```
pintos –q run 'echo x'
```

- Merge the last three lines into one

```
pintos –p ../../examples/echo –a echo -- -f –q run 'echo x'
```

Pintos VM layout



Running a program in pintos

Calling "process_execute"

```
static void run_task(char ** argv)
{
...
process_wait(process_excute(argv));
...
}
```

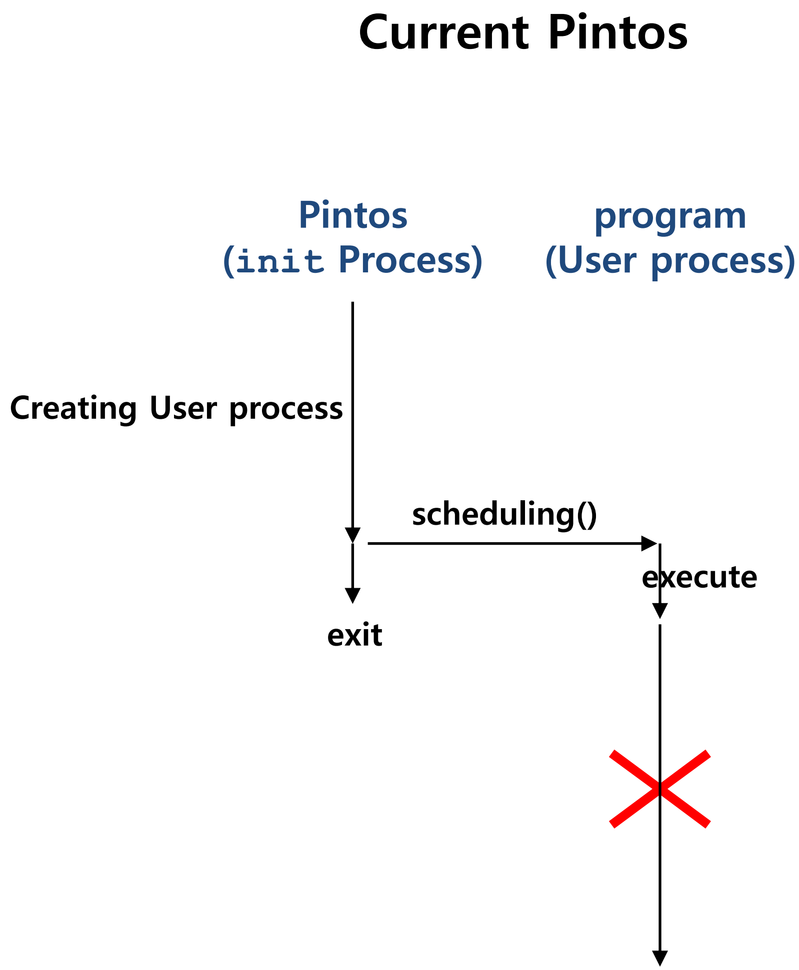
Create thread and start running a program

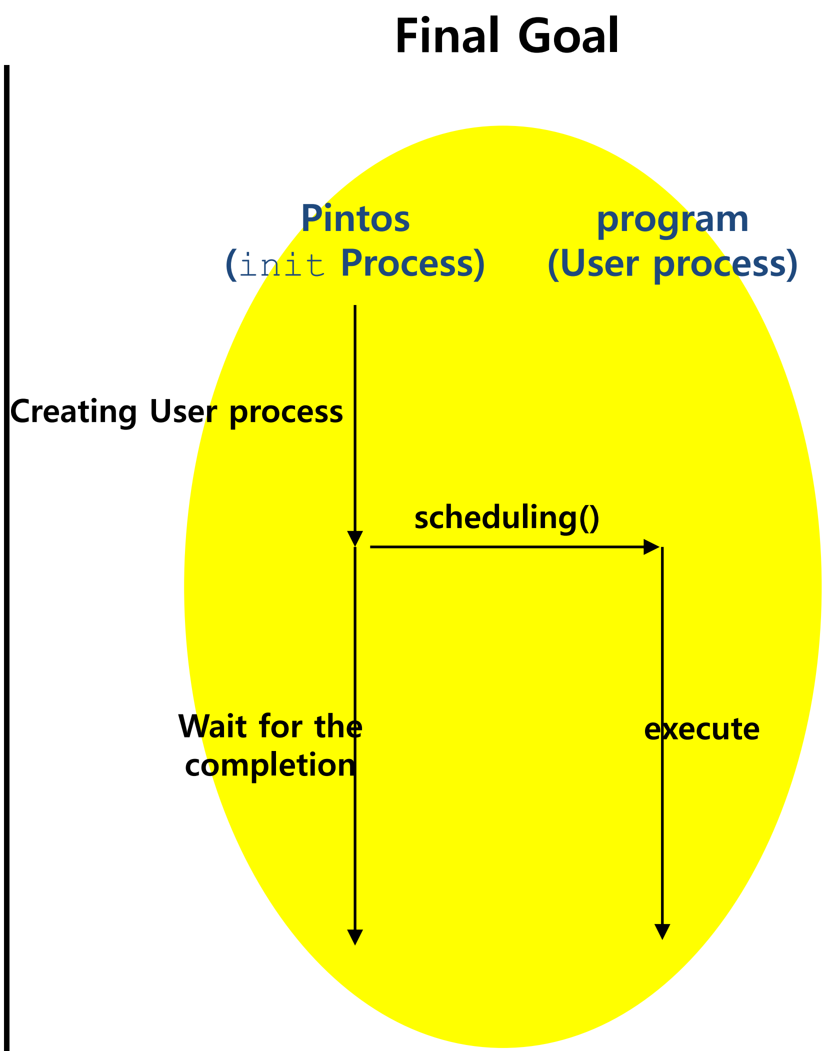
```
tid_t process_execute (const char
                       *file_name)
{
 ...
 tid = thread_create (..,start_process,.);
 ...
 return tid;
}
```

```
int process_wait (tid_t child_tid UNUSED)
{
    return -1;
}
```

The OS quits without waiting for the process to fi nish!!!

Executing a program



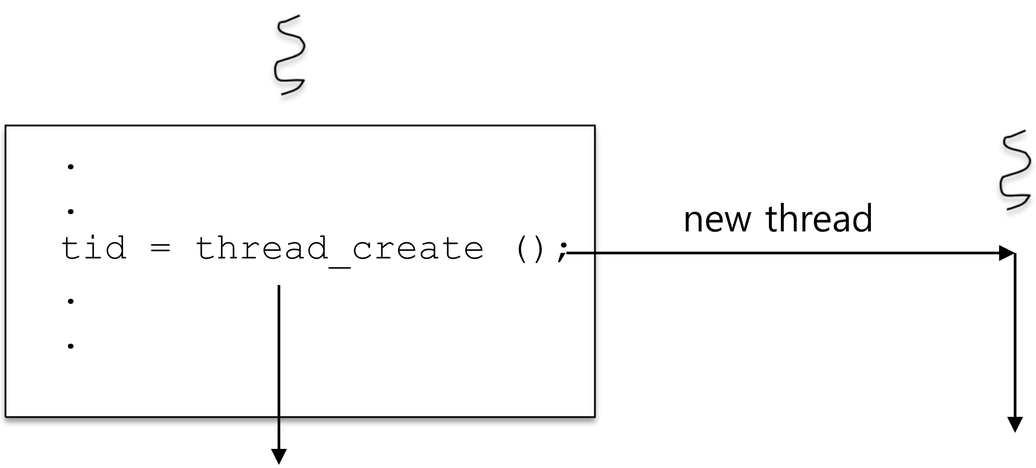


Executing a program

- Execute “file_name”.

pintos/src/userprog/process.c

```
tid_t process_execute (const char *file_name)
{
    char *fn_copy;
    tid_t tid;
    ...
    tid = thread_create (file_name, PRI_DEFAULT, start_process,
                         fn_copy);
    ...
    return tid;
}
```



Creating a thread

- `thread_create()`
  - Create "`struct thread`" and initialize it.
  - Allocate the kernel stack.
  - Register the function to run: `start_process`.
  - Add it to ready list.

Creating a thread

pintos/src/threads/thread.c – thread_create()

```
tid_t thread_create (const char *name, int priority,
                     thread_func *function, void *aux)
{
  struct thread *t;
  struct kernel_thread_frame *kf;
  ...
  t = palloc_get_page (PAL_ZERO); /* allocating one page*/
  init_thread (t, name, priority); /* initialize thread structure*/
  tid = t->tid = allocate_tid ();  /* allocate tid */
  /* Stack frame for kernel_thread(). */
  kf = alloc_frame (t, sizeof *kf);/* allocate stack */
  kf->eip = NULL;
  kf->function = function; /* function to run*/
  kf->aux = aux;           /* parameters for the function to run */
  ...
  /* Add to run queue. */
  thread_unblock (t);
  return tid;
}
```

Starting a process

- `load()`: load the program of name '`file_name`'
- If it successfully loads the program, run it. Otherwise, `exit()`.
- `thread_exit()` : quit the thread.

pintos/src/userprog/process.c

```
static void start_process (void *file_name_)
{
    char *file_name = file_name_;
    ...
    /* if_.esp: address of the top of the user stack */
    success = load (file_name, &if_.eip, &if_.esp);
    if (!success)
        thread_exit ();
    /* start user program */
    asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g"
                  (&if_) : "memory");
}
```

start_process

```
static void start_process (void *file_name_)
{
  char *file_name = file_name_;
  struct intr_frame if_;
  bool success;
  ...
  success = load (file_name, &if_.eip, &if_.esp);
  if (!success)
    thread_exit ();
  /* Start the user process */
  asm volatile ("movl %0, %%esp; jmp    intr_exi
t" : : "g" (&if_) : "memory");
}
```

```
bool load (const char *file_name, void (*
*eip) (void), void **esp)
{
  ...
  struct file *file = NULL;
  ...
  file = filesys_open (file_name);
  ...
  /* Set up stack. */
  if (!setup_stack (esp))
  ...
  success = true;
  return success;
}
```

```
void thread_exit (void)
{
  ...
  process_exit ();
  intr_disable ();
  list_remove (&thread_current()->allelem);
  thread_current ()->status = THREAD_DYING;
  schedule ();
}
```

Loading a program.

- Load a ELF file.
  - Create page table (2 level paging).
  - Open the file, read the ELF header.
  - Parse the file, load the ‘data’ to the data segment.
  - Create user stack and initialize it.

```c
bool load (const char *file_name, void (**eip) (void), void **esp) {
  struct thread *t = thread_current ();
  struct Elf32_Ehdr ehdr;
  struct file *file = NULL;
  ...
  t->pagedir = pagedir_create ();  /* create page directory */
  process_activate ();             /* set cr3 register*/
  file = filesys_open (file_name);   /* Open the file*/
/* parse the ELF file and get the ELF header*/
  if (file_read (file, &ehdr, sizeof ehdr) != sizeof ehdr
      || memcmp (ehdr.e_ident, "\177ELF\1\1\1", 7)
      || ehdr.e_type != 2
      || ehdr.e_machine != 3
      || ehdr.e_version != 1
      || ehdr.e_phentsize != sizeof (struct Elf32_Phdr)
      || ehdr.e_phnum > 1024)
  /* load segment information */
  struct Elf32_Phdr phdr;
  if (file_ofs < 0 || file_ofs > file_length (file))
      file_seek (file, file_ofs);
  if (file_read (file, &phdr, sizeof phdr) != sizeof phdr)
  ...
  /* load the executable file */
  if (!load_segment (file, file_page, (void *) mem_page,
                          read_bytes, zero_bytes, writable))
  ...
  if (!setup_stack (esp)) /* initializing user stack*/
  *eip = (void (*) (void)) ehdr.e_entry; /*initialize entry point*/
}
```

Passing the arguments and creating a thread

Overview

- For "`echo x y z`"
  - Original:
    - Thread name: "`echo x y z`"
    - Find program with file name "`echo x y z`"
    - Arguments "`echo`", "`x`", "`y`", and "`z`" are not passed
  - After modification
    - Thread name: "`echo`"
    - Find program with file name "`echo`"
    - Push the arguments to user stack.
- Files to modify
  - pintos/src/userprog/process.*

Parse the arguments and push them to the user stack

- pintos/src/userprog/process.c

```
tid_t process_excute() (const char *file_name)
```

  - Parse the string of `file_name`
  - Forward first token as name of new process to `thread_create()` function

```
static void start_process() (void *file_name_)
```

  - Parse `file_name`
  - Save tokens on user stack of new process.

Tokenizing

```
char *strtok_r (char *s, const char *delimiters,
                char **save_ptr) /* string.h */
```

- Receive a string (`s`) and `delimiters` and parse them by `delimiters`

ex) Parsing a string by the first space

```
char s[] = "String to tokenize.";
   char *token, *save_ptr;
   for (token = strtok_r (s, " ", &save_ptr); token != NULL;
        token = strtok_r (NULL, " ", &save_ptr))
     printf ("'%s'\n", token);
```

Result

```
'String'
'to'
'tokenize.'
```

Program Name

- Thread name
  - Before: Entire command line is passed to `thread_create()`
  - After modification: Forward only first token of command line to first argument of `thread_create()`
    - "echo x y z" → only use "echo" for name of process

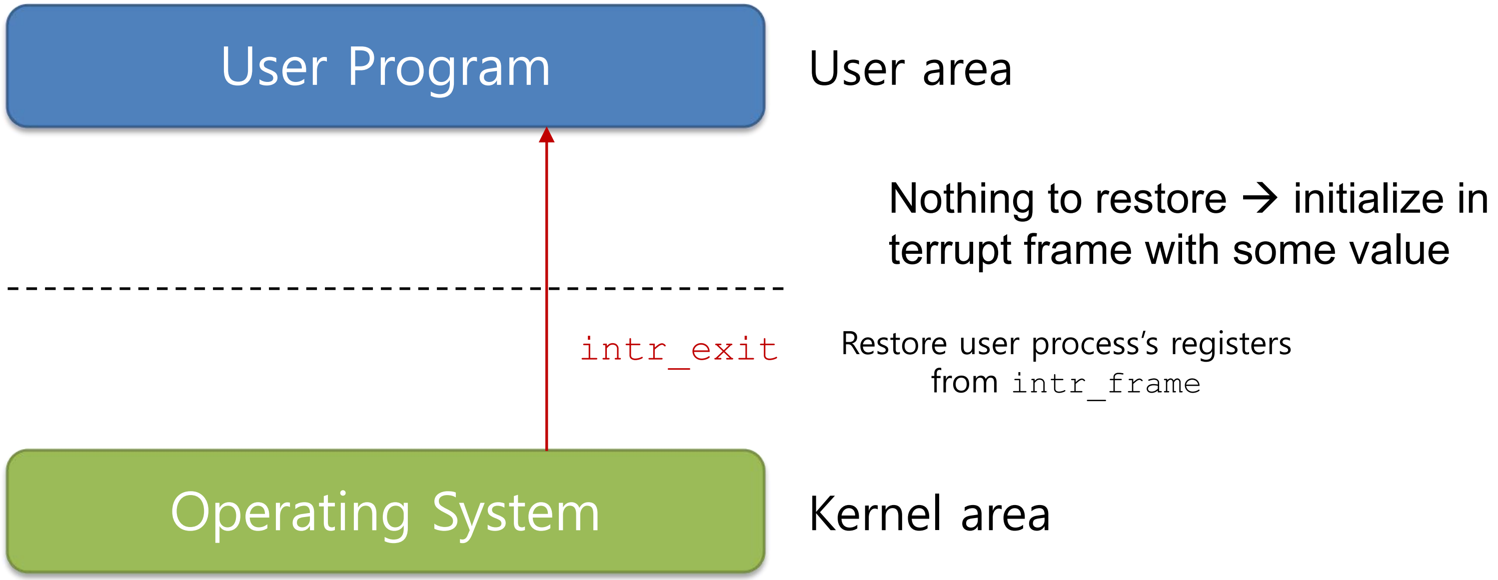
pintos/src/userprog/process.c

```
tid_t process_execute (const char *file_name)
{
  ...
  /* Parse command line and get program name */
  ...
  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create (file_name, PRI_DEFAULT, start_process,
fn_copy);
...
}
```

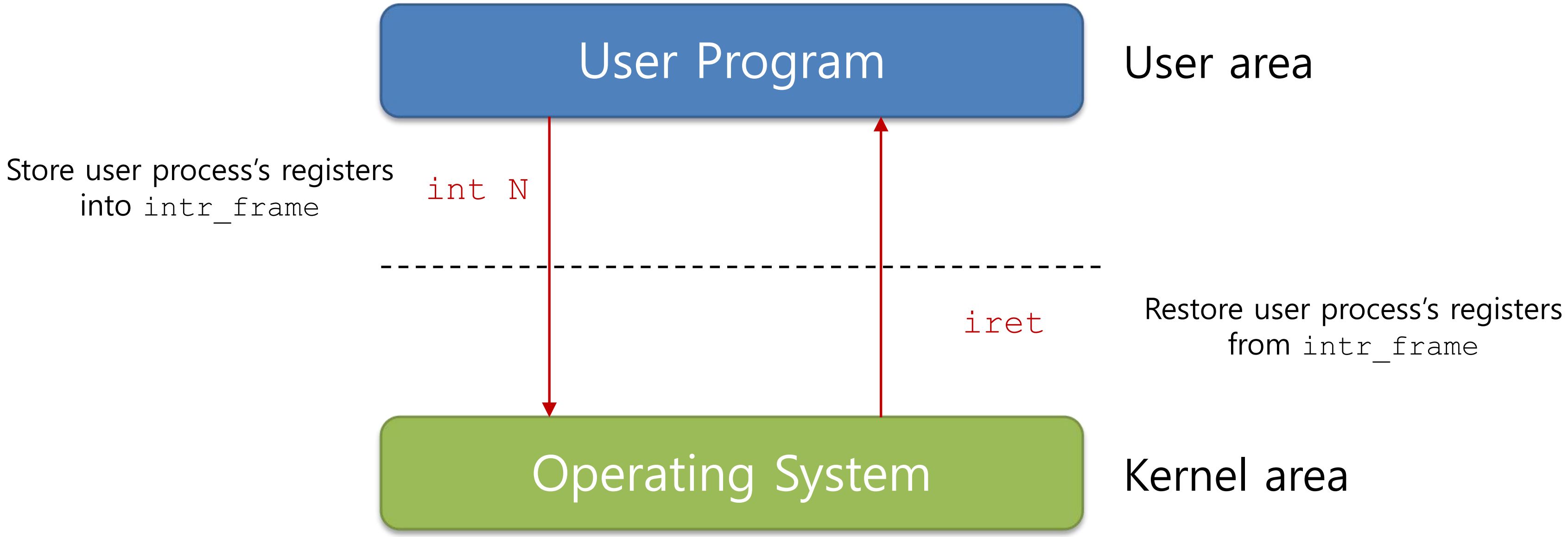
Name of thread

start_process

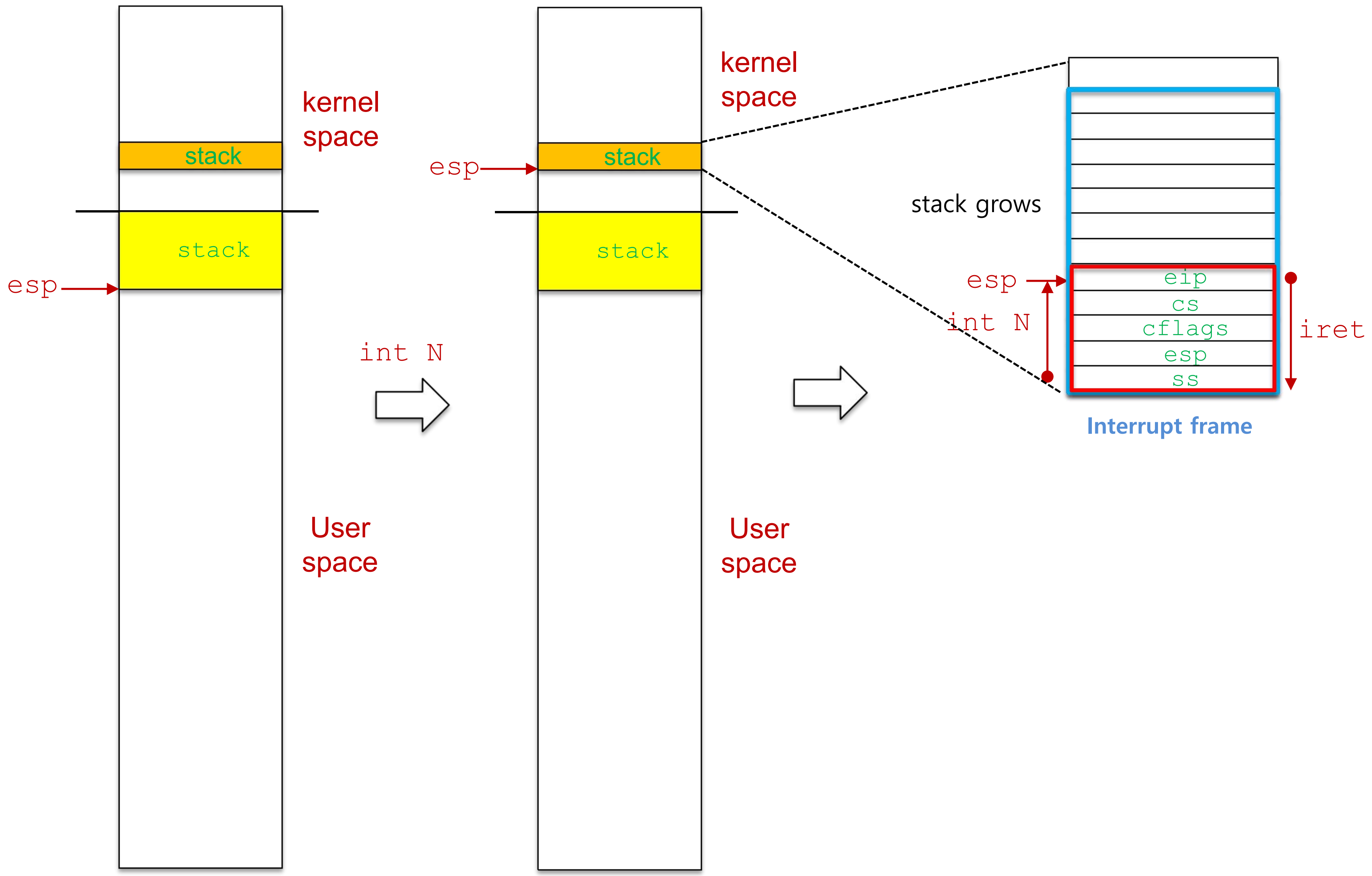
- Allocate interrupt frame.
- Load program and initialize interrupt frame and user stack.
- Setup arguments at the user stack.
- Jump to the user program through `interrupt_exit`.



Getting into and out of kernel



Getting into and out of kernel



struct intr_frame

```
struct intr_frame {
    /* Pushed by intr_entry in intr-stubs.S.
       These are the interrupted task's saved registers. */
    uint32_t edi;               /* Saved EDI. */
    uint32_t esi;               /* Saved ESI. */
    uint32_t ebp;               /* Saved EBP. */
    uint32_t esp_dummy;         /* Not used. */
    uint32_t ebx;               /* Saved EBX. */
    uint32_t edx;               /* Saved EDX. */
    uint32_t ecx;               /* Saved ECX. */
    uint32_t eax;               /* Saved EAX. */
    uint16_t gs, :16;           /* Saved GS segment register. */
    uint16_t fs, :16;           /* Saved FS segment register. */
    uint16_t es, :16;           /* Saved ES segment register. */
    uint16_t ds, :16;           /* Saved DS segment register. */

    /* Pushed by intrNN_stub in intr-stubs.S. */
    uint32_t vec_no;            /* Interrupt vector number. */

    /* Sometimes pushed by the CPU,
       otherwise for consistency pushed as 0 by intrNN_stub.
       The CPU puts it just under `eip', but we move it here. */
    uint32_t error_code;        /* Error code. */

    /* Pushed by intrNN_stub in intr-stubs.S.
       This frame pointer eases interpretation of backtraces. */
    void *frame_pointer;        /* Saved EBP (frame pointer). */

    /* Pushed by the CPU.
       These are the interrupted task's saved registers. */
    void (*eip) (void);         /* Next instruction to execute. */
    uint16_t cs, :16;           /* Code segment for eip. */
    uint32_t eflags;            /* Saved CPU flags. */
    void *esp;                  /* Saved stack pointer. */
    uint16_t ss, :16;           /* Data segment for esp. */
};
```

Stack grows.

- It is in the kernel stack.
- It stores user process' registe rs.

Getting into kernel.

`int n`

- when execute the kernel function, e.g. interrupt handler, system call, the OS saves the registers of currently executing process.
- Where: at the kernel stack of the executing process.
- execution
  1. Set the `esp` to point to kernel stack
  2. Pushes registers.

Entering the kernel

```
struct intr_frame {
    /* Pushed by intr_entry in intr-stubs.S.
       These are the interrupted task's saved registers. */
    uint32_t edi;               /* Saved EDI. */
    uint32_t esi;               /* Saved ESI. */
    uint32_t ebp;               /* Saved EBP. */
    uint32_t esp_dummy;         /* Not used. */
    uint32_t ebx;               /* Saved EBX. */
    uint32_t edx;               /* Saved EDX. */
    uint32_t ecx;               /* Saved ECX. */
    uint32_t eax;               /* Saved EAX. */
    uint16_t gs, :16;           /* Saved GS segment register. */
    uint16_t fs, :16;           /* Saved FS segment register. */
    uint16_t es, :16;           /* Saved ES segment register. */
    uint16_t ds, :16;           /* Saved DS segment register. */

    /* Pushed by intrNN_stub in intr-stubs.S. */
    uint32_t vec_no;            /* Interrupt vector number. */

    /* Sometimes pushed by the CPU,
       otherwise for consistency pushed as 0 by intrNN_stub.
       The CPU puts it just under `eip', but we move it here. */
    uint32_t error_code;        /* Error code. */

    /* Pushed by intrNN_stub in intr-stubs.S.
       This frame pointer eases interpretation of backtraces. */
    void *frame_pointer;        /* Saved EBP (frame pointer). */

    /* Pushed by the CPU.
       These are the interrupted task's saved registers. */
    void (*eip) (void);         /* Next instruction to execute. */
    uint16_t cs, :16;           /* Code segment for eip. */
    uint32_t eflags;            /* Saved CPU flags. */
    void *esp;                  /* Saved stack pointer. */
    uint16_t ss, :16;           /* Data segment for esp. */
};
```

esp — After interrupt handler, intr_entry

esp — After interrupt handler of intr N

esp — After int instruction.

time

Loading

- Load the program
  - Pass the program name to 'load()'.
  - "Load()" find executable file, using name of file and load it onto memory.

pintos/src/userprog/process.c

```
static void start_process (void *file_name_)
{
    char *file_name = file_name_;
    struct intr_frame if_;
    bool success;
    ...
    /* Parse the command line (Use strtok_r()) */

    /* Initialize interrupt frame and load executable. */
    memset (&if_, 0, sizeof if_);
    ...
    success = load (file_name, &if_.eip, &if_.esp);
    ...
}
```

program name Function entry point Stack top (user stack)

```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 ();
  /*missing parts!!! set up stack */
  /* Start the user process by simulating a return from an
     interrupt, implemented by intr_exit (in
     threads/intr-stubs.S).  Because intr_exit takes all of its
     arguments on the stack in the form of a `struct intr_frame',
     we just point the stack pointer (%esp) to our stack frame
     and jump to it. */
  asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
  NOT_REACHED ();
}
```

Getting out of the kernel

```
asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
```

```
movl %0, %%esp
```

Set the esp to the top of the interrupt frame.

```
jmp intr_exit
```

executes intr_exit

```
.globl intr_exit
.func intr_exit
intr_exit:
        /* Restore caller's registers. */
        popal
        popl %gs
        popl %fs
        popl %es
        popl %ds

        /* Discard `struct intr_frame' vec_no, error_code,
           frame_pointer members. */
        addl $12, %esp

        /* Return to caller. */
        iret
.endfunc
```

Getting out of the kernel

```
struct intr_frame {
    /* Pushed by intr_entry in intr-stubs.S.
       These are the interrupted task's saved registers. */
    uint32_t edi;               /* Saved EDI. */
    uint32_t esi;               /* Saved ESI. */
    uint32_t ebp;               /* Saved EBP. */
    uint32_t esp_dummy;         /* Not used. */
    uint32_t ebx;               /* Saved EBX. */
    uint32_t edx;               /* Saved EDX. */
    uint32_t ecx;               /* Saved ECX. */
    uint32_t eax;               /* Saved EAX. */
    uint16_t gs, :16;           /* Saved GS segment register. */
    uint16_t fs, :16;           /* Saved FS segment register. */
    uint16_t es, :16;           /* Saved ES segment register. */
    uint16_t ds, :16;           /* Saved DS segment register. */

    /* Pushed by intrNN_stub in intr-stubs.S. */
    uint32_t vec_no;            /* Interrupt vector number. */

    /* Sometimes pushed by the CPU,
       otherwise for consistency pushed as 0 by intrNN_stub.
       The CPU puts it just under `eip', but we move it here. */
    uint32_t error_code;        /* Error code. */

    /* Pushed by intrNN_stub in intr-stubs.S.
       This frame pointer eases interpretation of backtraces. */
    void *frame_pointer;        /* Saved EBP (frame pointer). */

    /* Pushed by the CPU.
       These are the interrupted task's saved registers. */
    void (*eip) (void);         /* Next instruction to execute. */
    uint16_t cs, :16;           /* Code segment for eip. */
    uint32_t eflags;            /* Saved CPU flags. */
    void *esp;                  /* Saved stack pointer. */
    uint16_t ss, :16;           /* Data segment for esp. */
};
```

esp

time

esp After `intr_exit`

esp After `iret` instruction

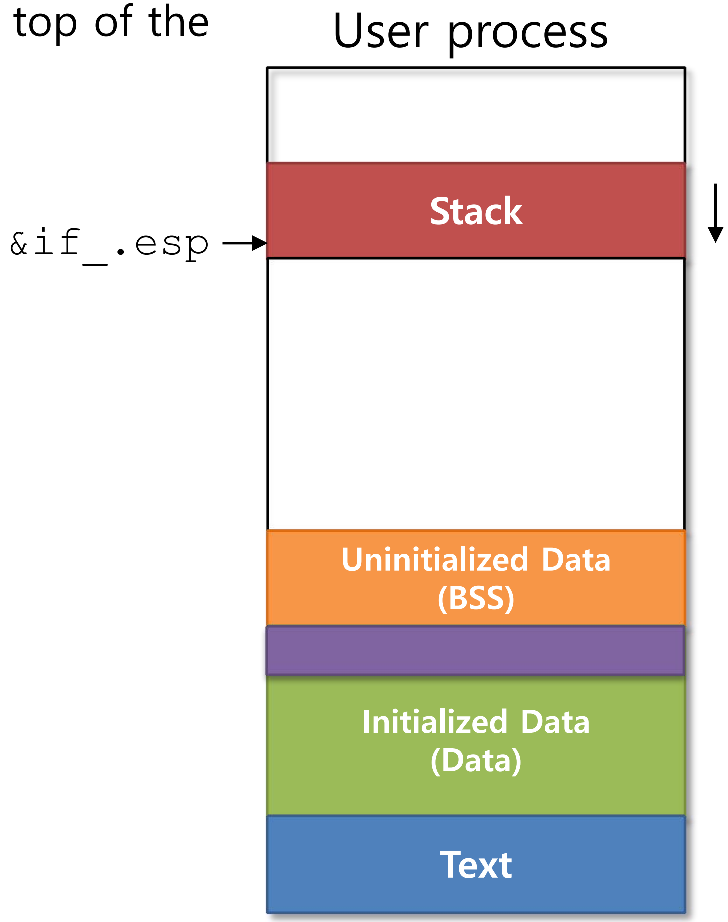
Write a function that sets up a stack.

"esp" field of the interrupt frame contains the stack top of the user stack.

```
sample_function(int argc,
                char* argv[],
                void **stackpointer)
```

Current stack top: `&if_.esp`

Start from `&if_.esp - 4`



Process Address Space

80x86 Calling Convention

```
%bin/ls -l foo bar
```

```
argc=4

argv[0] = "bin/ls", argv[1]= "-l", argv[2] = "foo", argv[3] = "bar"
```

1. Push arguments
   1. Push character strings from left to write.
   2. Place padding if necessary to align it by 4 Byte.
   3. Push start address of the character strings.
2. Push argc and argv
   1. Push argv
   2. Push argc
3. Push the address of the next instruction (return address).

User stack layout in function call

```
%bin/ls -l foo bar
```

Address	Name	Data	Type	
0xbffffffc	argv[3][...]	'bar₩0'	char[4]	Argument(string): 19 B
0xbffffff8	argv[2][...]	'foo₩0'	char[4]	
0xbffffff5	argv[1][...]	'-l₩0'	char[3]	
0xbfffffed	argv[0][...]	'/bin/ls₩0'	char[8]	
0xbfffffec	word-align	0	uint8_t	1Byte padding
0xbfffffe8	argv[4]	0	char *	Argument's address
0xbfffffe4	argv[3]	0xbffffffc	char *	
0xbfffffe0	argv[2]	0xbffffff8	char *	
0xbfffffdc	argv[1]	0xbffffff5	char *	
0xbfffffd8	argv[0]	0xbfffffed	char *	
0xbfffffd4	argv	0xbfffffd8	char **	main(int argc , char **argv)
0xbfffffd0	argc	4	int	
stack top → 0xbfffffcc	return address	0 (fake address)	void (*) ()	→ fake address(0)

grows ↓

Why is "return address" here is 0?

Interim Check

- Print the program's stack by using `hex_dump()` (`stdio.h`)
  - Print memory dump in hexadecimal form
  - Check if arguments are correctluy pushed on user stack.

pintos/src/userprog/process.c

```
static void start_process (void *file_name_)
{
    ...
    success = load (file_name, &if_.eip, &if_.esp);
    ...                                                    추가
    argument_stack(parse , count , &if_.esp);
    hex_dump(if_.esp , if_.esp , PHYS_BASE - if_.esp , tru
e);

    asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g"
(&if_) : "memory");
    NOT_REACHED ();
}
```

Intermediate Check (Cont.)

- Result

```
$pintos -v -- run 'echo x'
```

```
Execution of 'echo x' complete.
'echo'
'x'
Success : 1
esp : bfffffe0
bfffffe0  00 00 00 00 02 00 00 00-ec ff ff bf f9 ff ff bf |................|
bffffff0  fe ff ff bf 00 00 00 00-00 65 63 68 6f 00 78 00 |.........echo.x.|
system call!
```

return address (fake)

argc

argv

echo

x

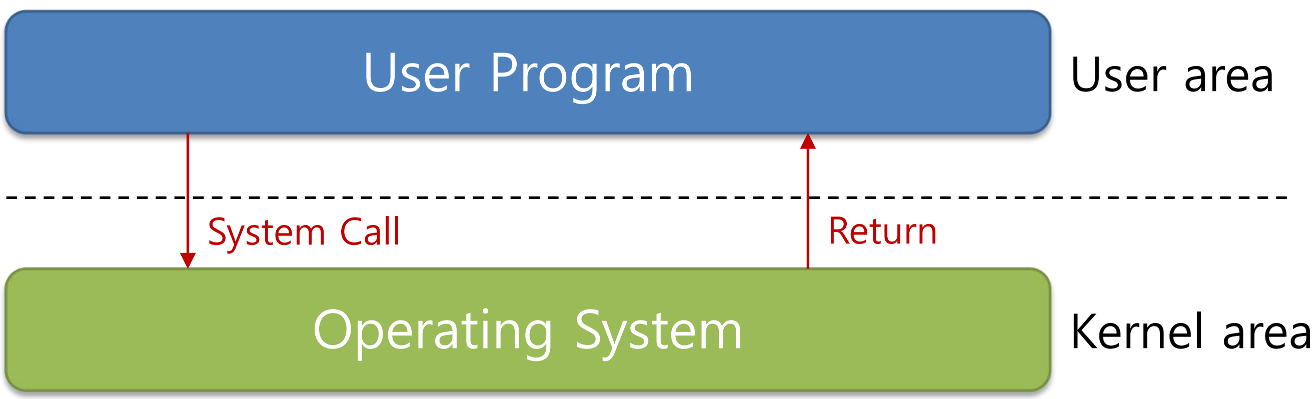
System Calls and Handlers

Overview

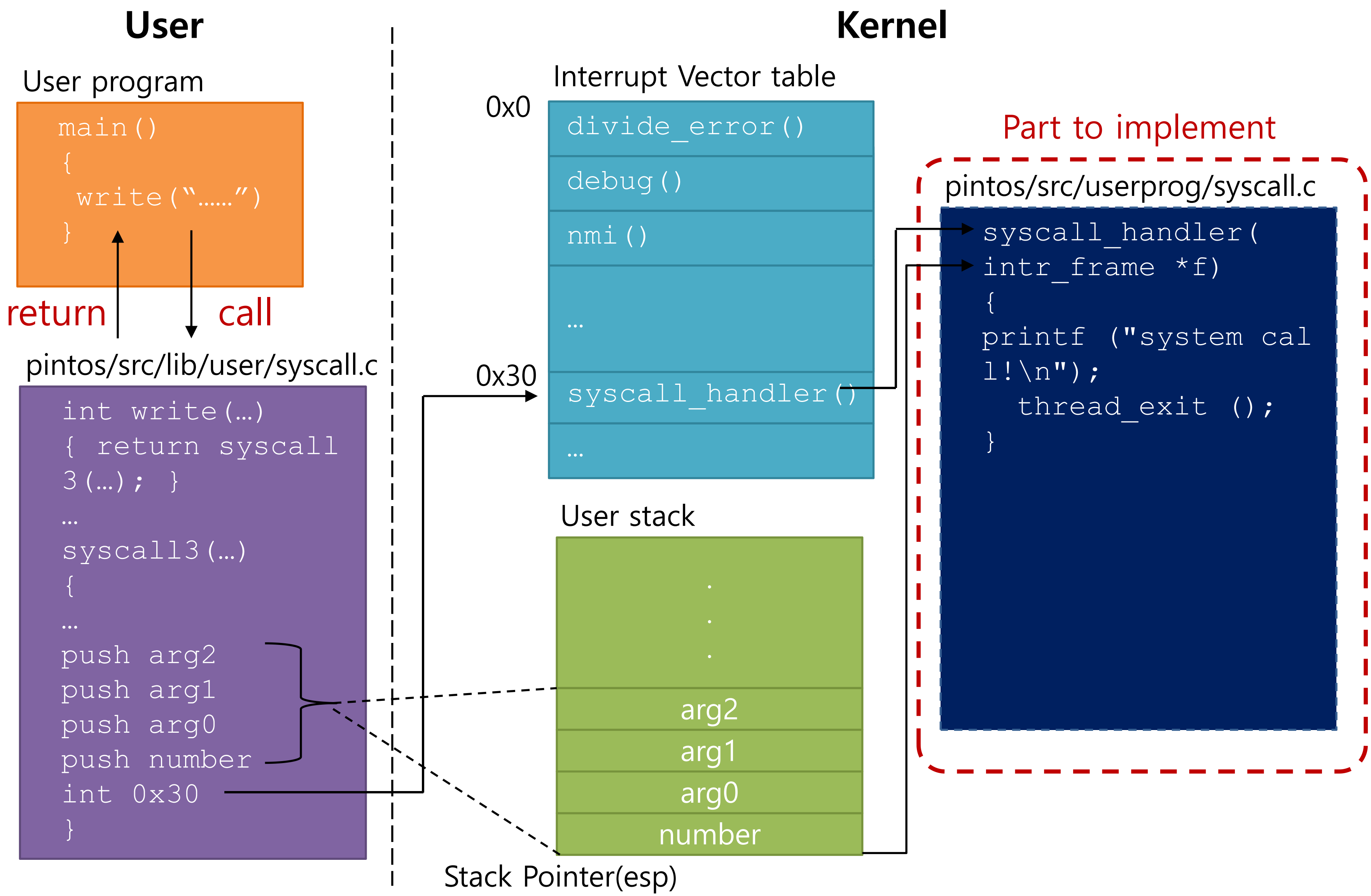
- Main goal
  - Original: system call handler table is empty.
  - After modification:
    - Fill system call handler of pintos out.
    - Add system calls to provide services to users
      - Process related: `halt, exit, exec, wait`
      - File related: `create, remove, open, filesize, read, write, seek, tell, close`
- Files to modify
  - pintos/src/threads/thread.*
  - pintos/src/userprog/syscall.*
  - pintos/src/userprog/process.*

System call

- Programming interface for services provided by the operating system
- Allow user mode programs to use kernel features
- System calls run on kernel mode and return to user mode
- Key point of system call is that priority of execution mode is raised to the spe cial mode as hardware interrupts are generated to call system call

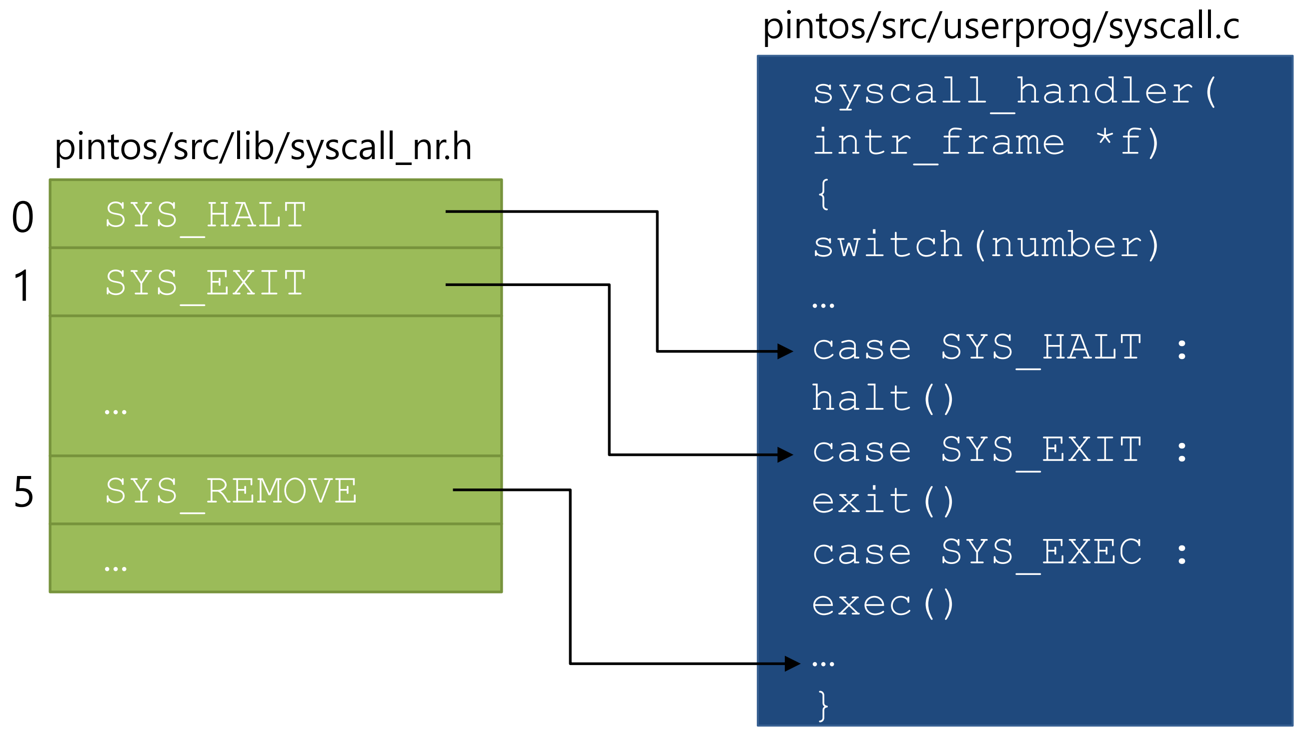


Call process of System call (Pintos)



System call handler

- Call the system call from the system call handler using the system call number.
  - The system call number is defined in pintos/src/lib/syscall_nr.h



Requirement for System Call handler

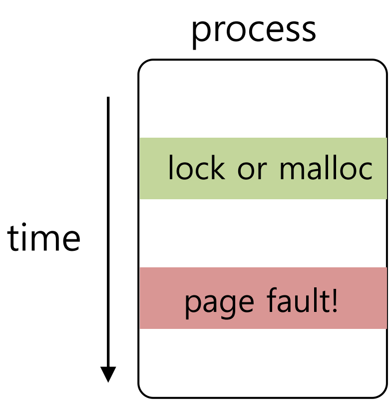
- Implement system call handler
  - Make system call handler call system call using system call number
  - Check validation of the pointers in the parameter list.
    - These pointers must point to user area, not kernel area.
    - If these pointers don't point the valid address, it is page fault
  - Copy arguments on the user stack to the kernel.
  - Save return value of system call at `eax` register.

Address Validation

- User can pass invalid pointers through the systemcall.
  - A null pointer / A pointer to unmapped virtual memory
  - A pointer to kernel virtual memory address space (above **PHYS_BASE**)
- Kernel need to detect invalidity of pointers and terminating process without harm to the kernel or other running processes.
- How to detect?
  - Method 1: Verify the validity of a user-provided pointer.
    - The simplest way to handle user memory access.
    - Use the functions in 'userprog/pagedir.c' and in 'threads/vaddr.h'
  - Method 2: Check only that a user points below PHYS_BASE.
    - An invalid pointer will cause 'page_fault'. You can handle by modifying the code for `page_fault()`.
    - Normally faster than first one, Because it takes advantage of the MMU.
    - It tends to be used in real kernel.

Accessing User Memory (cont.)

- In either case, make sure not to "leak" resource.



In the case, before terminating, we need to be sure release the lock or free the page.

- The first technique is straightforward.
  - Lock or allocate the page only after verifying the validity of pointers.
- The second one is more difficult.
  - Because there`s no way to return an error code from a memory access.
  - You can use provided functions to handle these cases. (functions are in next slide.)

Accessing User Memory (cont.)

```
/* 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;
}
```

```
/* Writes BYTE to user address UDST.
   UDST must be below PHYS_BASE.
   Returns true if successful, false if a segfault occurred.*/
static bool
put_user (uint8_t *udst, uint8_t byte)
{
    int error_code;
    asm ("movl $1f, %0; movb %b2, %1; 1:"
        : "=&a" (error_code), "=m" (*udst) : "q" (byte));
    return error_code != -1;
}
```

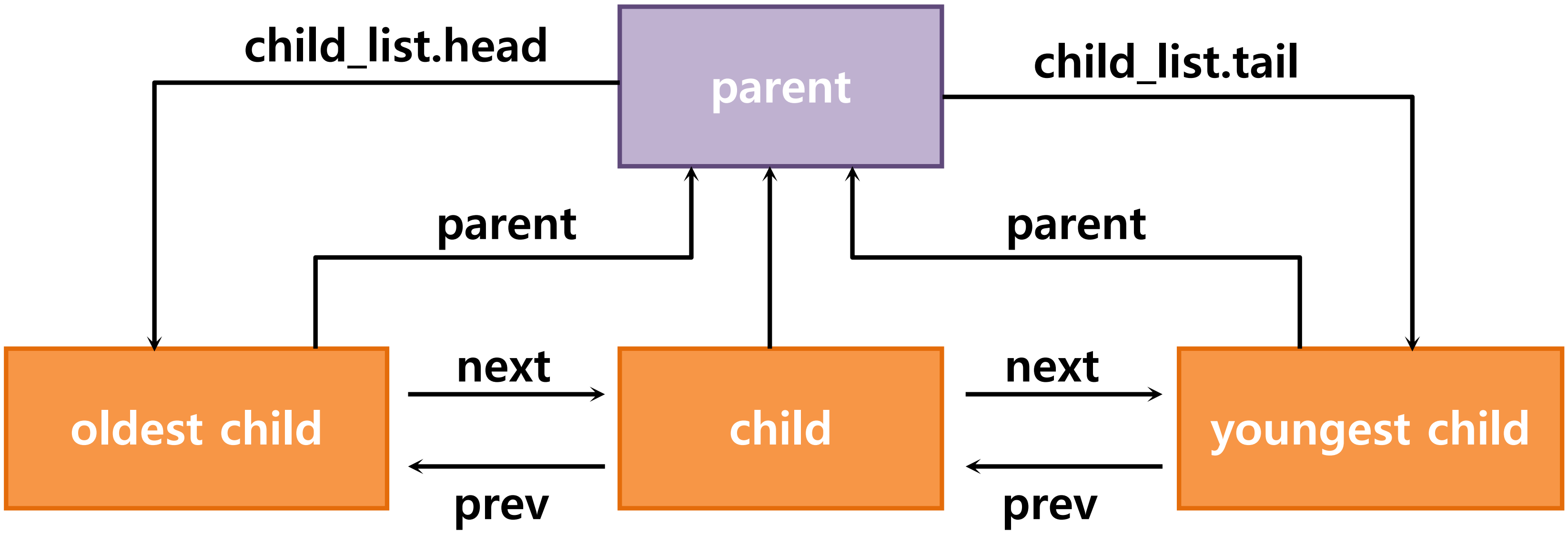
- You also modify the `page_fault ()`: set `eax` to `0xffffffff` and copies its former value into `eip`.

Add system calls: Process related system calls

- `void halt(void)`
  - Shutdown pintos
  - Use `void shutdown_power_off(void)`
- `void exit(int status)`
  - Exit process
  - Use `void thread_exit(void)`
  - It should print message “Name of process: exit(status)”.
- `pid_t exec (const char *cmd_line)`
  - Create child process and execute program corresponds to `cmd_line` on it
- `int wait (pid_t pid)`
  - Wait for termination of child process whose process id is `pid`

Process Hierarchy

- Augment the existing process with the process hierarchy.
- To represent the relationship between parent & child,
  - Pointer to parent process: `struct thread*`
  - Pointers to the sibling. `struct list`
  - Pointers to the children: `struct list_elem`



wait() system call

- `int wait(pid_t pid)`
  - Wait for a child process `pid` to exit and retrieve the child's exit status.
  - If `pid` is alive, wait till it terminates. Returns the status that `pid` passed to e xit.
  - If `pid` did not call `exit`, but was terminated by the kernel, return -1.
  - A parent process can call wait for the child process that has terminated.

    → return exit status of the terminated child process.
  - After the child terminates, the parent should deallocate its process descripto r
  - `wait` fails and return -1 if
    - `pid` does not refer to a direct child of the calling process.
    - The process that calls wait has already called wait on `pid`.

Kernel function for wait – `process_wait`

`int process_wait (tid_t child_tid UNUSED)`

- It is currently empty.

```
int
process_wait (tid_t child_tid UNUSED)
{
  return -1;
}
```

- Insert the infinite loop so that the kernel does not finish. For now…

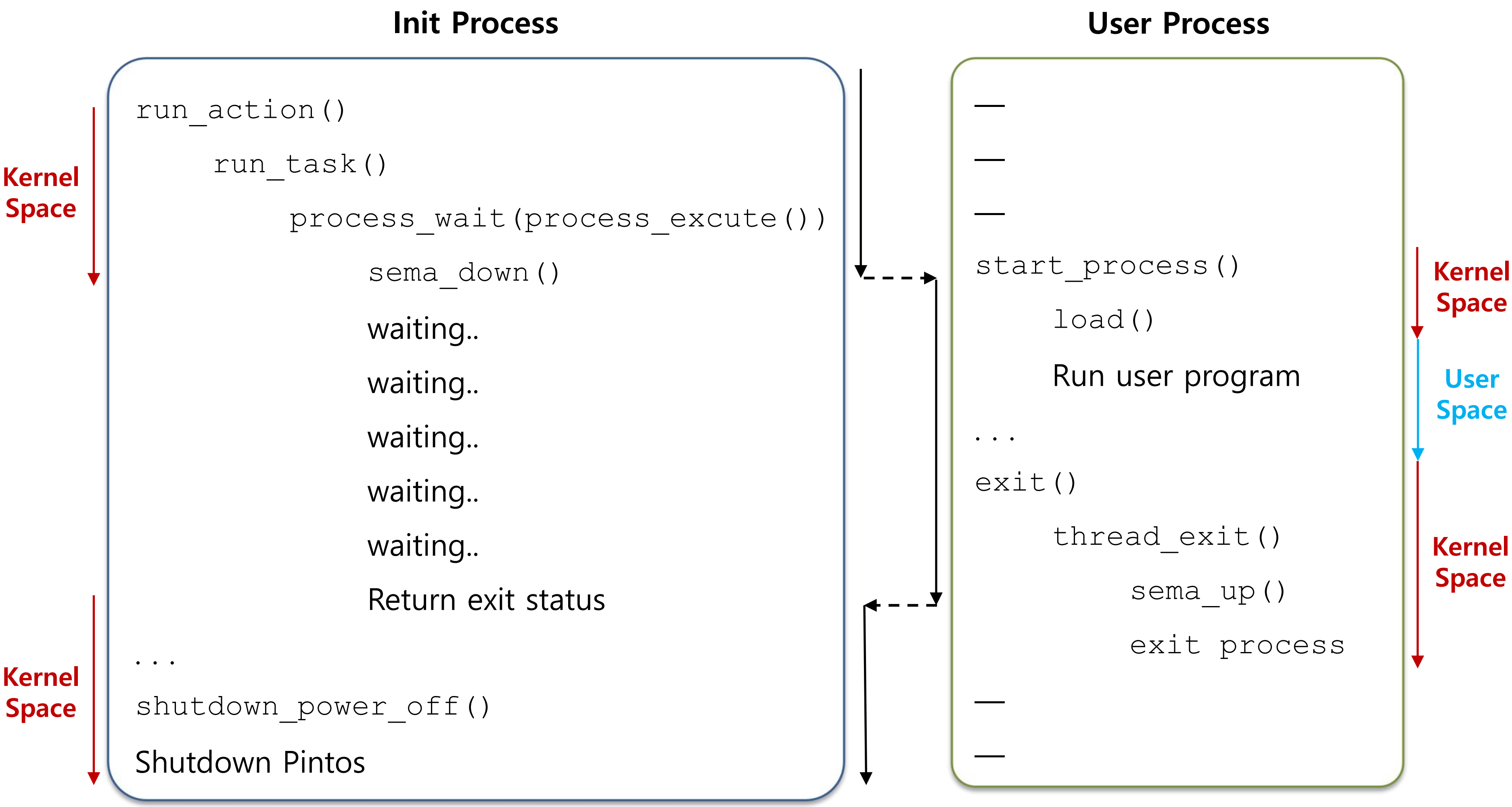
Correct implementation: `process_wait()`

- process_wait()
  - Search the descriptor of the child process by using child_tid.
  - The caller blocks until the child process exits.
  - Once child process exits, deallocate the descriptor of child process and returns exit status of the child process.
- Semaphore
  - Add a semaphore for “wait” to thread structure.
  - Semaphore is initialized to 0 when the thread is first created.
  - In wait(`tid`), call `sema_down` for the semaphore of `tid`.
  - In exit() of process `tid`, call `sema_up`.
  - Where do we need to place `sema_down` and `sema_up`?
- Exit status
  - Add a field to denote the exit status to the thread structure.

Flow of parent calling wait and child

- Flow of user program execution

Flow
Scheduling



`exec()` system call

`pid_t exec(const *cmd_line)`

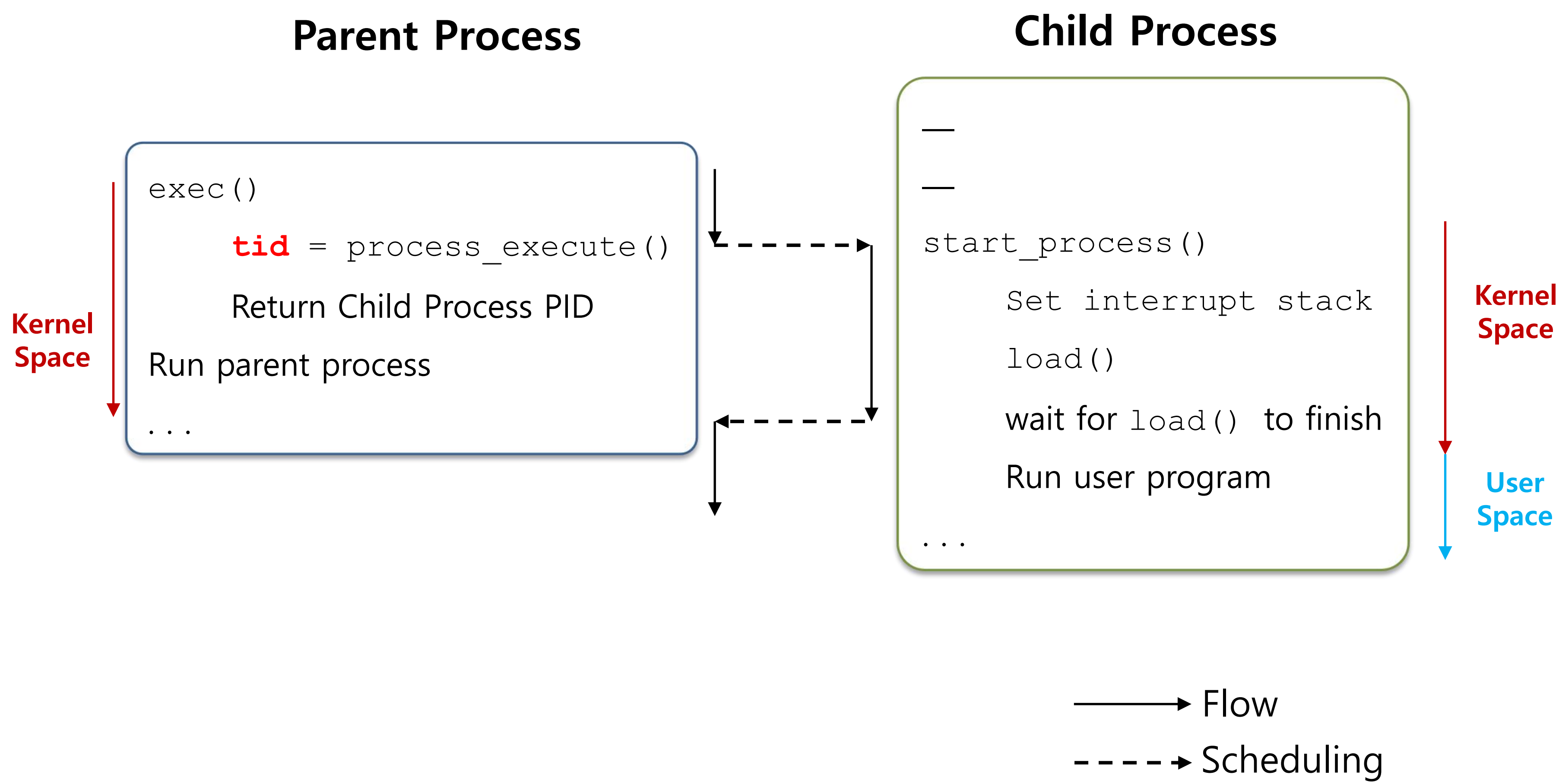
- Run program which execute `cmd_line`.
- Create thread and run. `exec()` in pintos is equivalent to `fork()+exec()` in Unix
.
- Pass the arguments to program to be executed.
- Return `pid` of the new child process.
- If it fails to load the program or to create a process, return -1.
- Parent process calling exec should wait until child process is created and loads t he executable completely.

Kernel function for `exec()`: `process_execute()`

- Parent should wait until it knows the child process has successfully created and the binary file is successfully loaded.
- Semaphore
  - Add a semaphore for "exec()" to thread structure.
  - Semaphore is initialized by 0 when the thread is first created.
  - Call `sema_down` to wait for the successful load of the executable file of the child process.
  - Call `sema_up` when the executable file is successfully loaded.
  - Where do we need to place `sema_down` and `sema_up`?
- load status
  - In the thread structure, we need a field to represent whether the file is successfully loaded or not.

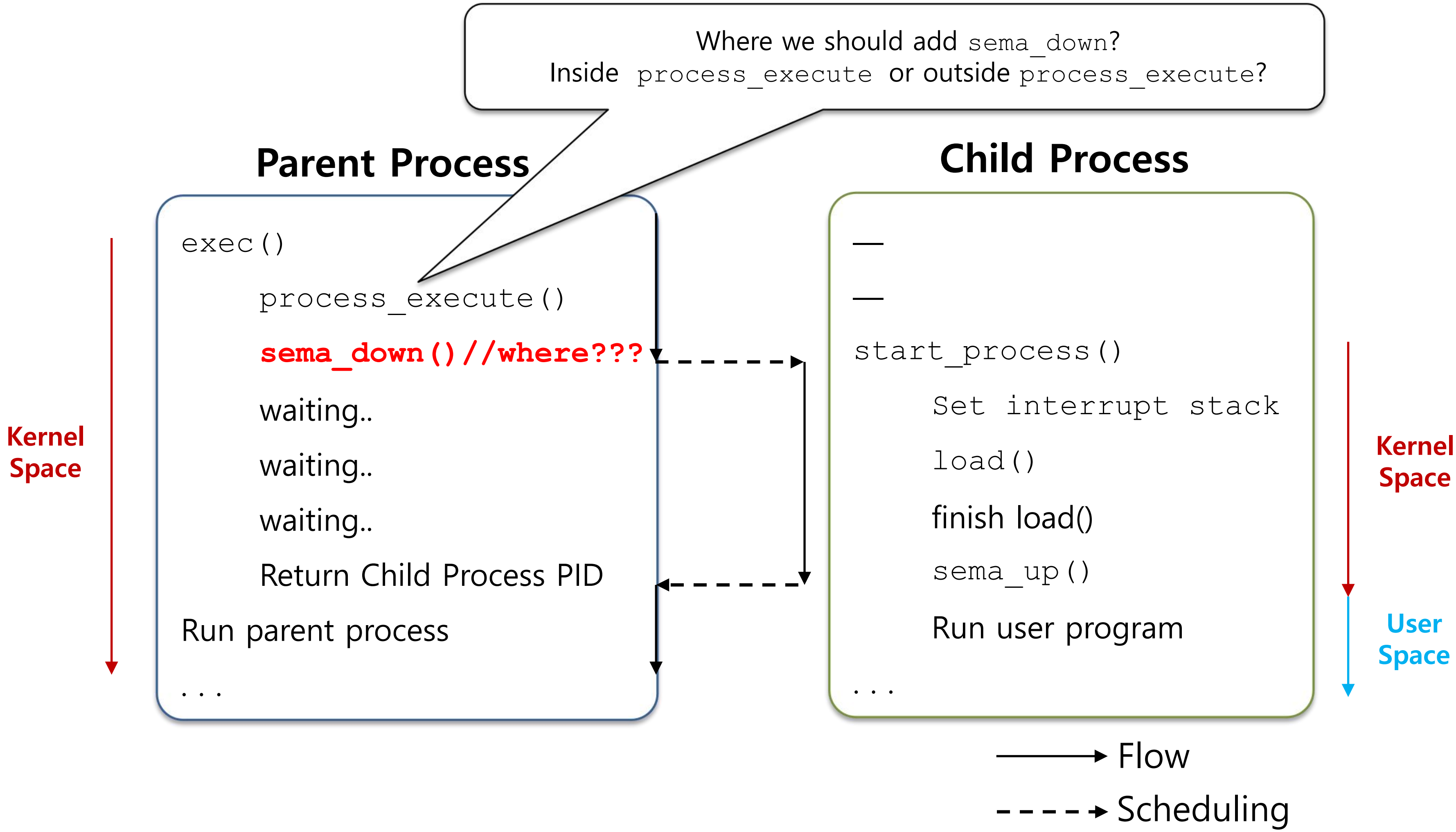
Current flow of the parent calling exec and the child

- `exec()` return itself only after child is completely loaded.
- `tid` can have valid value even the load has failed.



Correct Flow of the parent calling exec and the child

- `exec()` return itself only after child is completely loaded.



exit()

- Terminate the current user program, returning status to the kernel.
- If the process' parent waits for it, this is the status that will be returned.

```
void exit (int status)
{
    struct thread *cur = thread_current ();
    /* Save exit status at process descriptor */
    printf("%s: exit(%d)\n" , cur -> name , status);
    thread_exit();
}
```

Kernel function for `exit()`: `thread_exit`

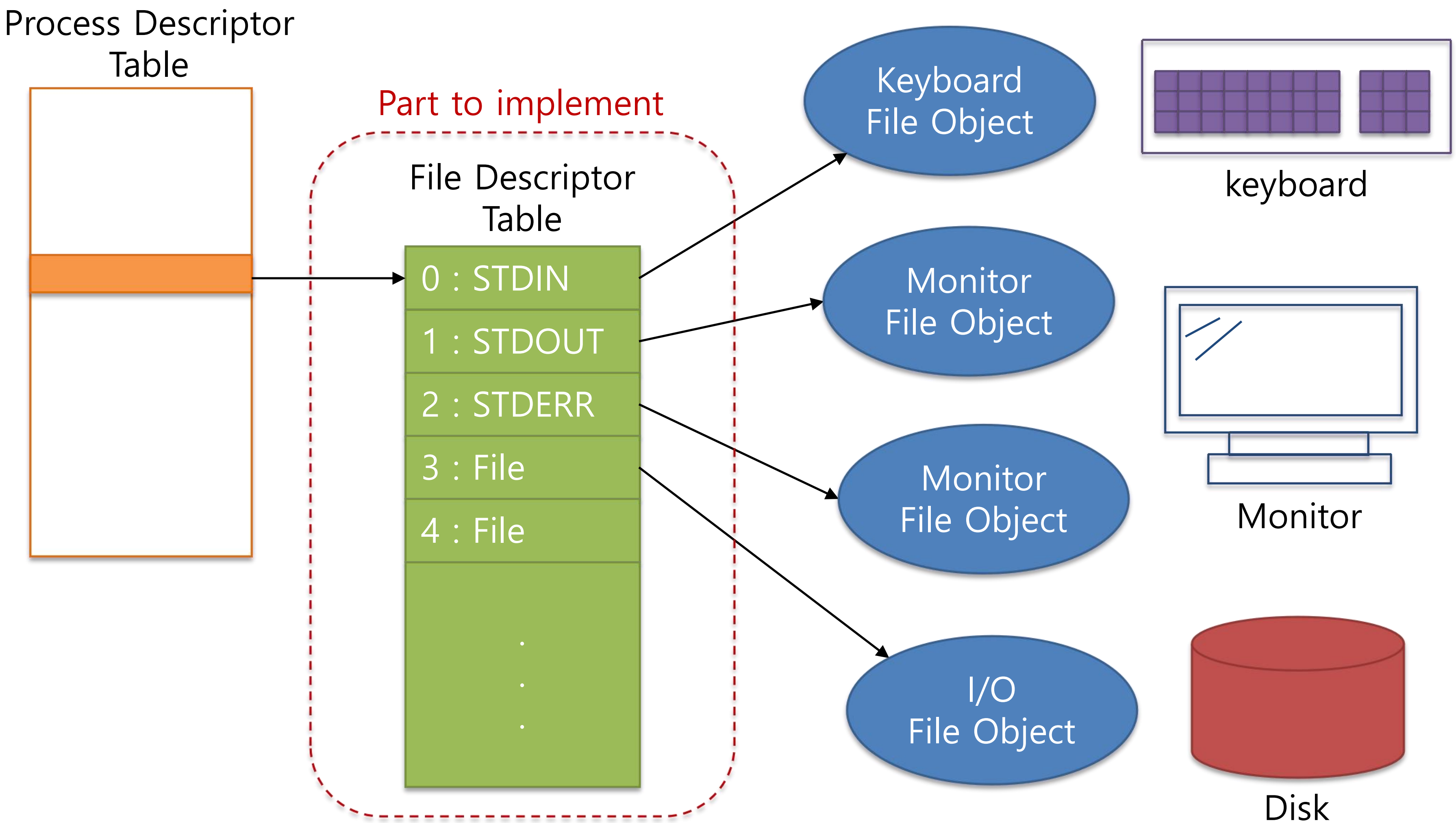
- Exit status
  - Store the status to the status of process.
- Semaphore
  - Call `sema_up` for the current process.

```
/* Deschedules the current thread and destroys it.  Never
   returns to the caller. */
void
thread_exit (void)
{
  ASSERT (!intr_context ());

#ifdef USERPROG
  process_exit ();
#endif

  /* Remove thread from all threads list, set our status to dying,
     and schedule another process.  That process will destroy us
     when it calls thread_schedule_tail(). */
  intr_disable ();
  list_remove (&thread_current()->allelem);
  thread_current ()->status = THREAD_DYING;
  schedule ();
  NOT_REACHED ();
}
```

File Manipulation

File Descriptor in Unix

- Access to File by using File Descriptor



File Descriptor Table

- Implement File Descriptor Table.
  - Each process has its own file descriptor table (Maximum size: 64 entry).
  - File descriptor table is an array of pointer to struct file.
  - FD is index of the file descriptor table, and it is allocated sequentially.
    - FD 0 and 1 are allocated for `stdin` and `stdout`, respectively.
  - `open()` returns fd.
  - `close()` set 0 at file descriptor entry at index fd.

Allocate File Descriptor Table

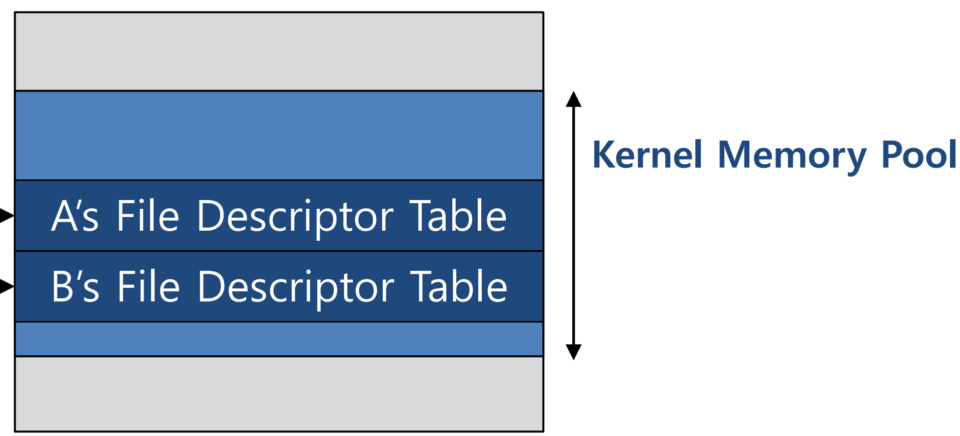
- Define FDT as a part of thread structure.

```
struct thread {
        …
        struct file fdt[64];
        int next_fd;
        …
};
```
Thread A

- Allocate FDT at kernel memory area, and add the associated pointer to at the thread structure.

```
struct thread {
        …
        struct file **fdt;
        int next_fd;
        …
};
```
Thread A

```
struct thread {
        …
        struct file **fdt;
        int next_fd;
        …
};
```
Thread B



File Descriptor Table

- When the thread is created,
  - Allocate File Descriptor table.
  - Initialize pointer to file descriptor table.
  - Reserve fd0, fd1 for stdin and stdout.
- When thread is terminated,
  - Close all files.
  - Deallocate the file descriptor table.
- Use global lock to avoid race condition on file,
  - Define a global lock on syscall.h (`struct lock filesys_lock`).
  - Initialize the lock on `syscall_init()` (Use `lock_init()`).
  - Protect filesystem related code by global lock.

Modify page_fault() for test

- Some tests check whether your kernel handles the bad process properly.
- Pintos needs to kill the process and print the thread name and the exit status -1 when page fault occurs.
- We have to modify `page_fault()` to satisfy test's requirements.

pintos/src/userprog/exception.c

```
static void page_fault (struct intr_frame *f)
{
    ...
    not_present = (f->error_code & PF_P) == 0;
    write = (f->error_code & PF_W) != 0;
    user = (f->error_code & PF_U) != 0;

    /* Call exit(-1) */
    ...
}
```

Add system calls: File related system calls

- `bool create(const char *file, unsigned initial_size)`
  - Create file which have size of `initial_size`.
  - Use `bool filesys_create(const char *name, off_t initial_size)`.
  - Return true if it is succeeded or false if it is not.
- `bool remove(const char *file)`
  - Remove file whose name is `file`.
  - Use `bool filesys_remove(const char *name)`.
  - Return true if it is succeeded or false if it is not.
  - File is removed regardless of whether it is open or closed.
- `int open(const char *file)`
  - Open the file corresponds to path in “`file`”.
  - Return its fd.
  - Use `struct file *filesys_open(const char *name)`.

Add system calls: File related system calls (Cont.)

- `int filesize(int fd)`
  - Return the size, in bytes, of the file open as `fd`.
  - Use `off_t file_length(struct file *file)`.
- `int read(int fd, void *buffer, unsigned size)`
  - Read `size` bytes from the file open as `fd` into `buffer`.
  - Return the number of bytes actually read (0 at end of file), or -1 if fails.
  - If fd is 0, it reads from keyboard using `input_getc()`, otherwise reads from file using `file_read()` function.
    - `uint8_t input_getc(void)`
    - `off_t file_read(struct file *file, void *buffer, off_t size)`

Add system calls: File related system calls (Cont.)

- `int write(int fd, const void *buffer, unsigned size)`
  - Writes `size` bytes from `buffer` to the open file `fd`.
  - Returns the number of bytes actually written.
  - If fd is 1, it writes to the console using `putbuf()`, otherwise write to the file using `file_write()` function.
    - `void putbuf(const char *buffer, size_t n)`
    - `off_t file_write(struct file *file, const void *buffer, off_t size)`
- `void seek(int fd, unsigned position)`
  - Changes the next byte to be read or written in open file `fd` to `position`.
  - Use `void file_seek(struct file *file, off_t new_pos)`.

Add system calls: File related system calls (Cont.)

- `unsigned tell(int fd)`
  - Return the position of the next byte to be read or written in open file `fd`.
  - Use `off_t file_tell(struct file *file)`.
- `void close(int fd)`
  - Close file descriptor `fd`.
  - Use `void file_close(struct file *file)`.

Denying writes to executable

- What if the OS tries to execute the file that is being modified?
- Do not allow the file to be modified when it is opened for execution.
- Approach
  - When the file is loaded for execution, call `file_deny_write()`.
  - When the file finishes execution, call `file_allow_write()`.

```
static bool load (const char *cmdline, void (**eip) (void), void **esp)
```

- Call `file_deny_write()` when program file is opened.
- Add a running file structure to `thread` structure.

```
void process_exit (void)
```

- Modify current process to close the running file.

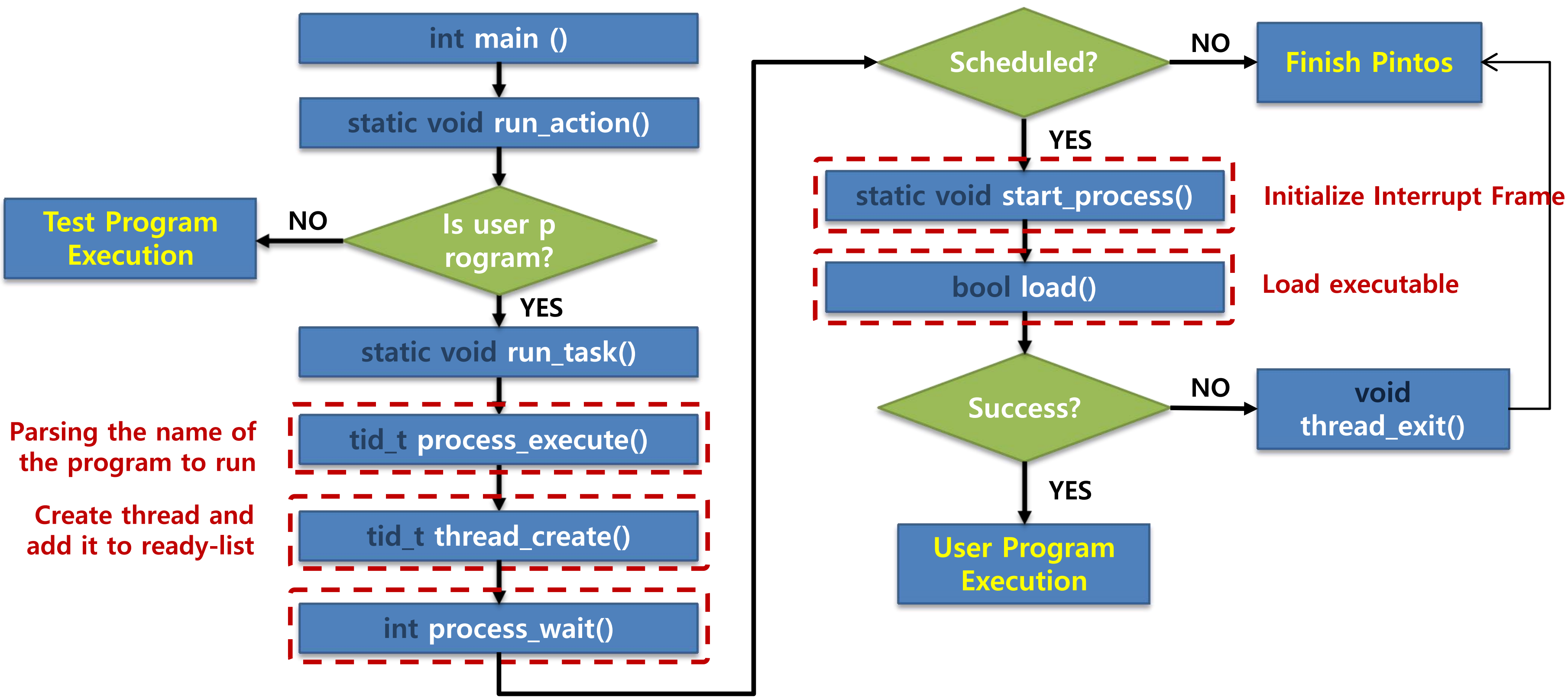
Result

- Check the result of all tests.
  - Path: pintos/src/userprog

```
$make grade
```

```
SUMMARY BY TEST SET

Test Set                                      Pts Max  % Ttl  % Max
--------------------------------------------- --- --- ------ ------
tests/userprog/Rubric.functionality           108/108  35.0%/ 35.0%
tests/userprog/Rubric.robustness               88/ 88  25.0%/ 25.0%
tests/userprog/no-vm/Rubric                     1/  1  10.0%/ 10.0%
tests/filesys/base/Rubric                      30/ 30  30.0%/ 30.0%
--------------------------------------------- --- --- ------ ------
Total                                                 100.0%/100.0%
```

Summary



int main ()
static void run_action()
Is user p rogram?
NO
Test Program Execution
YES
static void run_task()
Parsing the name of the program to run
tid_t process_execute()
Create thread and add it to ready-list
tid_t thread_create()
int process_wait()
Scheduled?
NO
Finish Pintos
YES
static void start_process()
Initialize Interrupt Frame
bool load()
Load executable
Success?
NO
void thread_exit()
YES
User Program Execution