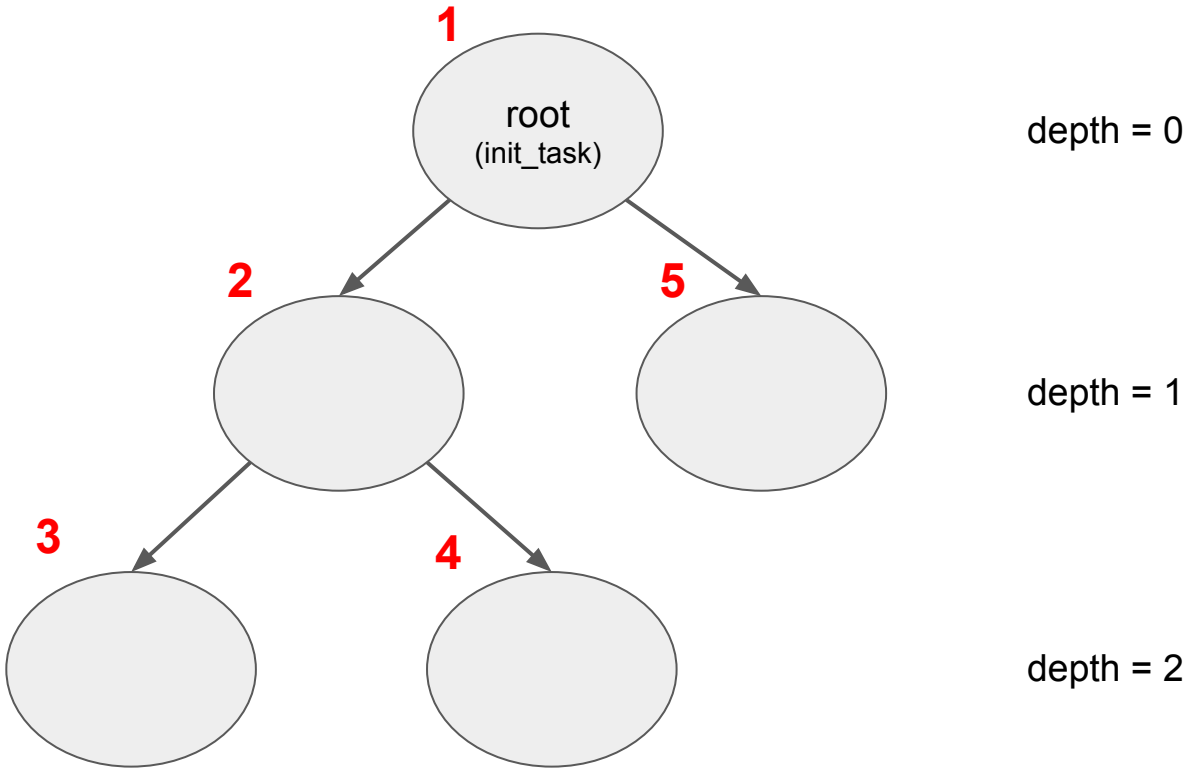
# Project 1 Help Document

March 9, 2021
SNU Operating Systems

# General Overview of Project 1

- Register a system call
  - `ssize_t ptree(struct pinfo *buf, size_t len)`
  - System call number **399**
  - You can name your function `sys_ptree`
- Traverse `task_struct` in pre-order
  - Understand how they are linked with each other
  - Copy some fields in `task_struct` structures
- Test your system call
  - Print the entire process tree keeping the hierarchy

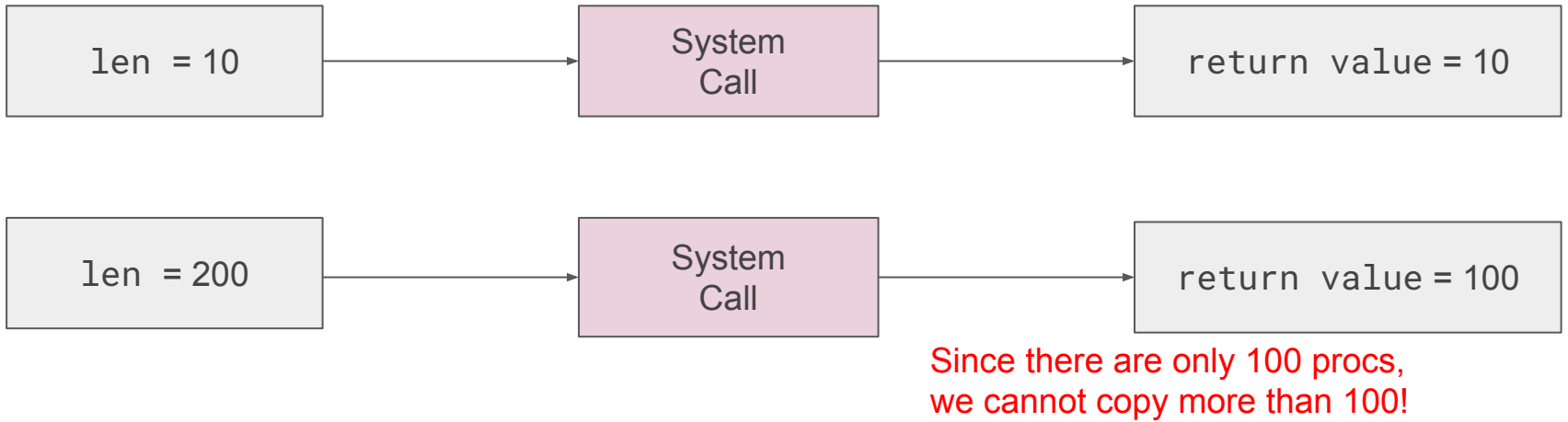
# Print Process Tree



# Return Value

- Success
  - Your system call should return the actual number of entries copied.
  - May be smaller than `len`

Assume there are 100 procs in total...

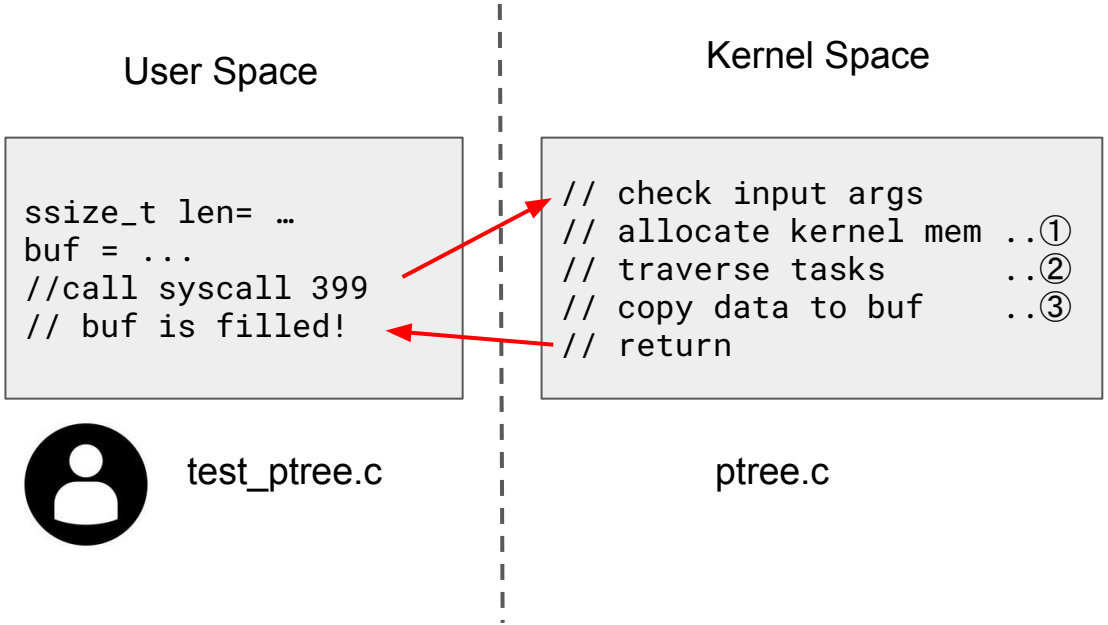


# Return Value

- Error
  - `-EINVAL`
    - If `buf` is `NULL` or `len` == 0
  - `-EFAULT`
    - If `buf` is outside the accessible address space
  - You may handle other errors but we will not check them for grading
  - Defined in `include/uapi/asm-generic/errno-base.h`
- How to print error messages?
  - Use `errno` and `perror()`

# Overall Codeflow

```
ssize_t ptree(struct pinfo *buf, size_t len)
```



# 1. Allocate Kernel Memory

- Kernel stack size is far smaller than user stack
  - Be cautious when allocating local arrays or having recursive calls
  - `kmalloc` & `kfree` is more recommended
- `kmalloc` is similar to `malloc`, but has an additional flag parameter
  - Defined in `linux/slab.h`
  - `void *kmalloc(size_t size, int flags)`
  - `flags` controls kmalloc behavior
- `kfree` is similar to `free`

# 2. Doubly Linked List in Linux Kernel

- What is `task_struct`?
  - Struct defined in `include/linux/sched.h`
  - Contain all information you need for Proj 1!
- `children` and `sibling`: implemented as **doubly linked lists**

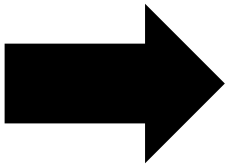
```
/* Real parent process: */
struct task_struct __rcu        *real_parent;

/* Recipient of SIGCHLD, wait4() reports: */
struct task_struct __rcu        *parent;

/*
 * Children/sibling form the list of natural children:
 */
struct list_head                children;
struct list_head                sibling;
struct task_struct              *group_leader;
```

# 2. Doubly Linked List in Linux Kernel

- Unlike common linked lists, kernel list nodes are stored ***inside*** data
- Linux kernel has a doubly linked list implementation for kernel programming
  - Extensively used across all Linux kernel code

```
struct list_head {
    struct list_head *next, *prev;
}
```

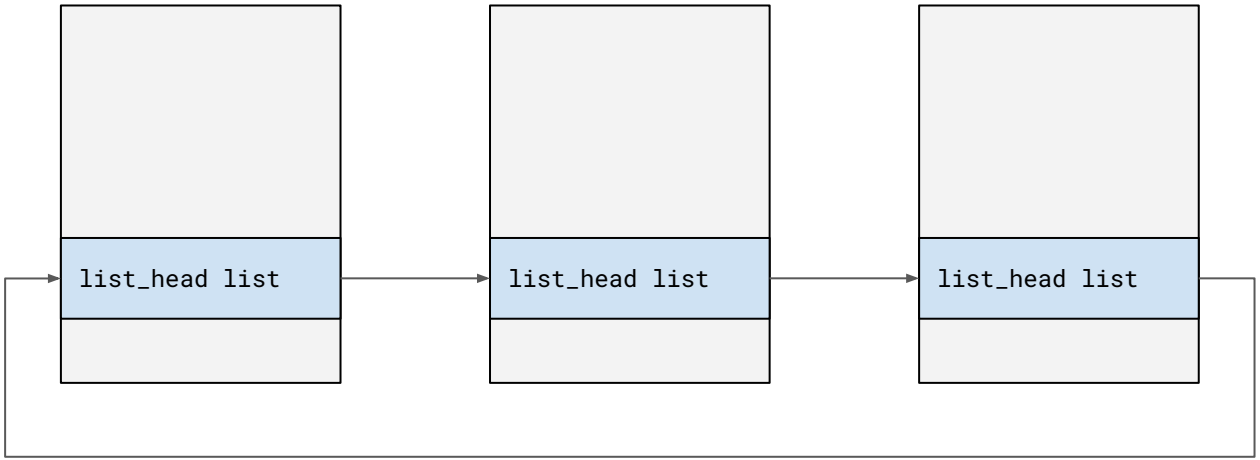
```
struct student {
    char* name;
    char* student_id;
    struct student *prev;
    struct student *next;
};
```



```
struct student {
    char* name;
    char* student_id;
    struct list_head list;
};
```

# 2. Doubly Linked List in Linux Kernel

```
struct list_head {
    struct list_head *next, *prev;
}
```



# 2. Helpful Macros for Doubly Linked List

- Initializing a list node (must be declared beforehand)
  - `INIT_LIST_HEAD(&first_student->list)`
  - `LIST_HEAD(student_list)` ← Declaration + Initialization
- Add or delete from linked list
  - `list_add` / `list_add_tail`: adds a node to a list
  - `list_del` : deletes a node from a list
- Iterate the entries in the linked list
  - `list_for_each_entry`: iterates over a list
  - `container_of`: returns the item given a list node
- More about Linux kernel list (highly recommended)
  - Have a look at `include/linux/list.h`
  - http://www.makelinux.net/ldd3/chp-11-sect-5.shtml

# 3. Access to User Memory

- In kernel mode, you should avoid directly accessing user memory space
  - Can result in kernel panic
- `include/asm/uaccess.h` provides macros for this
  - `get_user` / `put_user`: copies simple variables
  - `copy_from_user` / `copy_to_user`: copies a block of data
  - More on http://www.ibm.com/developerworks/library/l-kernel-memory-access/

# Example Program Output

- `swapper/0` (pid 0)
  - The first ever process created
  - Used to represent the state of ‘not working’
- `systemd` (pid 1)
  - Manages all the processes
- `kthreadd` (pid 2)
  - Kernel thread daemon
  - `kthread_create`

```
sh-3.2# ./proj1
swapper/0,0,0,0,1,0,0
        systemd,1,1,0,167,2,0
                systemd-journal,167,1,1,0,185,0
                systemd-udevd,185,1,1,0,241,0
                dbus-daemon,241,1,1,0,297,81
                amd,297,1,1,0,298,301
                dlog_logger,298,1,1,0,307,1901
                buxton2d,307,1,1,0,313,375
                key-manager,313,1,1,0,325,444
```

```
        kthreadd,2,1,0,3,0,0
                 kworker/0:0,3,1026,2,0,4,0
                 kworker/0:0H,4,1026,2,0,5,0
                 kworker/u8:0,5,1026,2,0,6,0
                 mm_percpu_wq,6,1026,2,0,7,0
                 ksoftirqd/0,7,1,2,0,8,0
                 rcu_preempt,8,1026,2,0,9,0
                 rcu_sched,9,1026,2,0,10,0
```

# About Submission (IMPORTANT!)

- Don't be late!
  - TA will not grade the commits after the deadline.
- Write concise README.md
  - Describe how to build your kernel
  - Describe the high-level design and implementation
  - Investigation of process tree
  - **Any lessons learned** ←very important!

# Check Before Submission!

- Check you handle all invalid input arguments
- Check unsafe access to user space memory
- Whether you follow the project specifications (final check!)
- Whether you have delineated all unspecified/different implementation details in README
- **Both Black-box and White-box test will be held for this project**

# Q & A