# Operating Systems
# COMS W4118
# Lecture 19

Alexander Roth

2015 – 04 – 07

## 1 Diving into the Linux Kernel

- the macro `SYSCALL_DEFINE0` is used to define a system call with no formal parameters.
- `pid` is really the id of the `task_struct` which is used to denote the value of each thread.
- There is some virtualization built into the linux kernel.
- Linux has this concept of a *container*.
- A container is a small world within the kernel that allows for virtualization throughout the system.
- Self-contained environments for processes.
- It is possible that using a container creates different PID's from what the user sees and what the kernel actually knows about.
- These different PID's are mapped together so the kernel knows what the user is referring to.
- There are two structs in the `task_struct`: parent and real parent.
- Real parent is the process that created you. Parent is the process that is the current parent, or the process that receives the `SIGCHILD` signal.
- The distinction between `getppid` and `getpid` is the use of locking.
- We create a readlock for `getppid` in order to avoid the issue where the parent could exit by the time the function returns.
- `rcu_read_lock` is very similar to a reader/writer lock, but this goes a step further.

- It is known as read copy update lock. You can have multiple readers and one writer acting at the same time.
- When a writer wants to update a data structure, it makes a copy of the structure.
- The structure is accessed through only one pointer.
- Once the write is complete, the writer switches the pointer to the old copy to the new copy atomically.
- It is possible in rcu for the reader to read an older version of a piece of data.
- However, the reader will never read an incomplete version of the data.
- There is a mechanism to make sure that writer does not change values until all the readers complete their read of the old data page.
- There is a significant overhead by creating a reader lock every time a reader must access a variable.
- You may have to go to the top of the file to see what's going on.
- On a certain level, you need to make an abstraction and abstract some of the lower level detail.
- You need to learn how to cut some things and abstract systems away.
- You must assume some certain operations.
- You must learn how to look at things and assume that's how they work.
- Locking should be avoided as much as possible.
- If you do not need to lock the majority of cases, then you should avoid locking and find another way to correct it.

# 2 Implementing Locks

- What does it mean to yield?
- What does it mane to sleep?
- The big question is "What does it mean for a process to sleep?"
- The virtual memory space is made up of 4G.
- With the typical structure of text, followed by static, followed by the heap.
- Where the heap begins is called the *brake* or `brk`.

- The stack starts around 3G and maps down, with empty space between the heap and the stack.
- The last 1G is the kernel code. `bzImage` file is the kernel code.
- The static section of the memory along with the text memory is mapped to physical regions within the RAM of the system.
- The OS silently converts these areas of code from virtual address space to physical address space.
- These virtual addresses must be transformed in order for the machine to be able to function correctly.
- The kernel image and data sections are common to all processes.
- Thus, they are images of a single piece of the memory.
- Each process only has one copy of the kernel running.
- The kernel memory contains the `bzImage` and a stack.
- It must have a stack for each process, which is typically 8K for each process.
- The starting and ending address for each process falls on the 8K boundary.
- You would take away the last 13 bits to locate the `thread_info` struct for the current task.
- The `thread_info` has a pointer to the `task_struct`.
- Where are the task structs?
- Within the kernel space, there must be task structs for everyone.
- All the task structs are linked together through an embedded linked list in the `list_head` struct. This is a doubly linked list.
- The very first process is `init_task` with a `pid` of 0, known as the swapper thread.
- Within the `struct task_list` there is a `int` variable known as `state`.
- `state` can have a number of values.
    1. `TASK_RUNNING`
    2. `TASK_INTERUPTABLE`
    3. `TASK_UNINTERUPTABLE`
    4. `TASK_TRACED`
    5. `TASK_ZOMBIE`

- Linux uses `TASK_RUNNING` for running processes and processes that can be run.
- `TASK_INTERUPTABLE` and `TASK_UNINTERUPTABLE` are used for sleeping processes.
- The scheduler moves through the linked list and gets the next runnable task on the run queue.
- The run queue now is a $n$-something list with a method to set priorities.