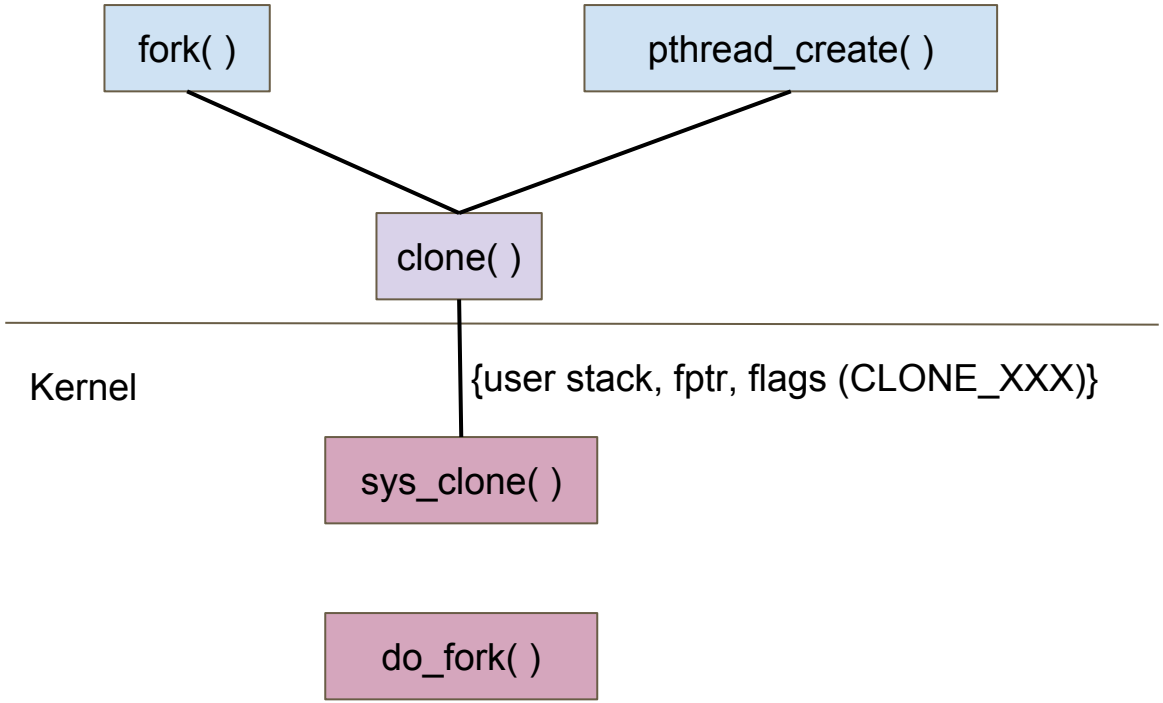
Process management

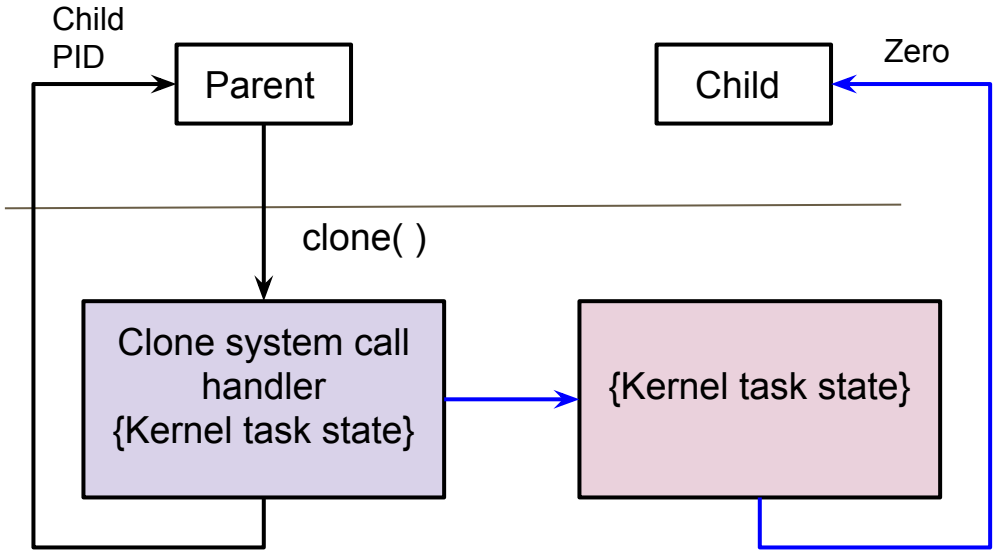
fork( ), exec( ) internals

Process, thread … seen so far



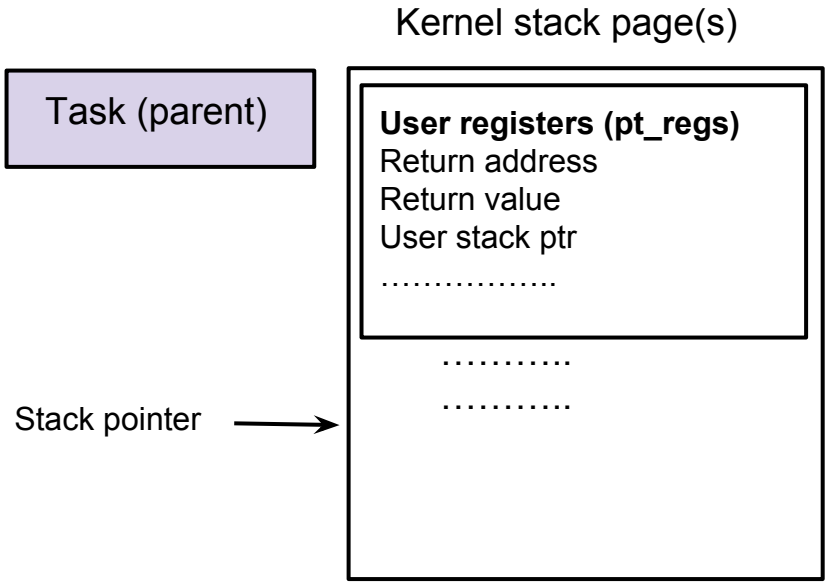
- ➔ User stack , function ptr, different resource sharing
  - CLONE_VM
  - CLONE_FILES
  - CLONE_SIGNAL
  - ….
- ➔ Experiments with CLONE_FILES
  - File close/open visible across processes

Process state replication



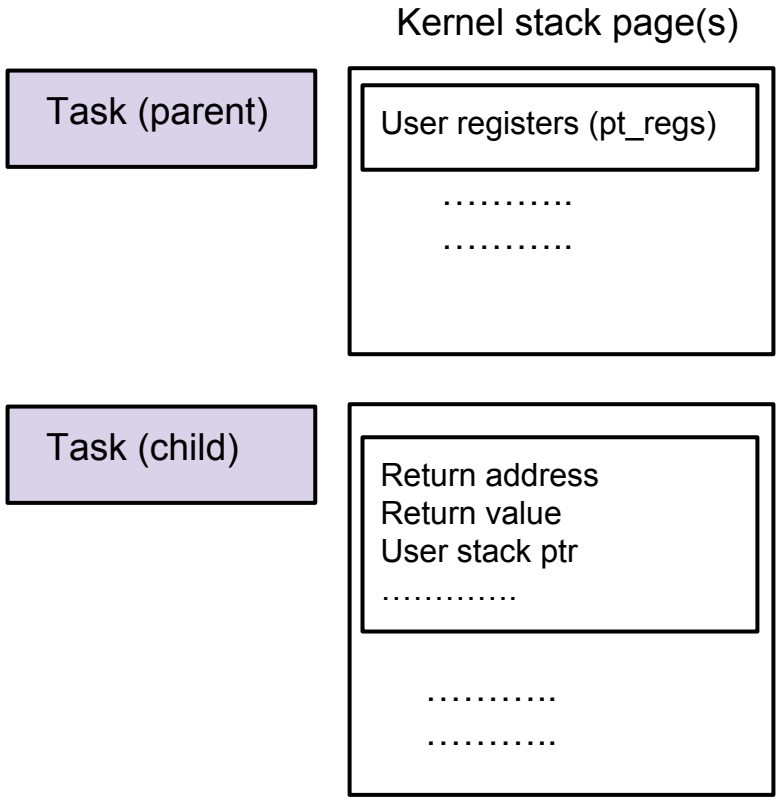
- How the kernel task state is created?
  - Replication depends on CLONE_XXX flags
- Other subtleties
  - Kernel stack for child
  - Different return values
  - Different return addresses
  - Different user stacks

Process state replication: x86_64



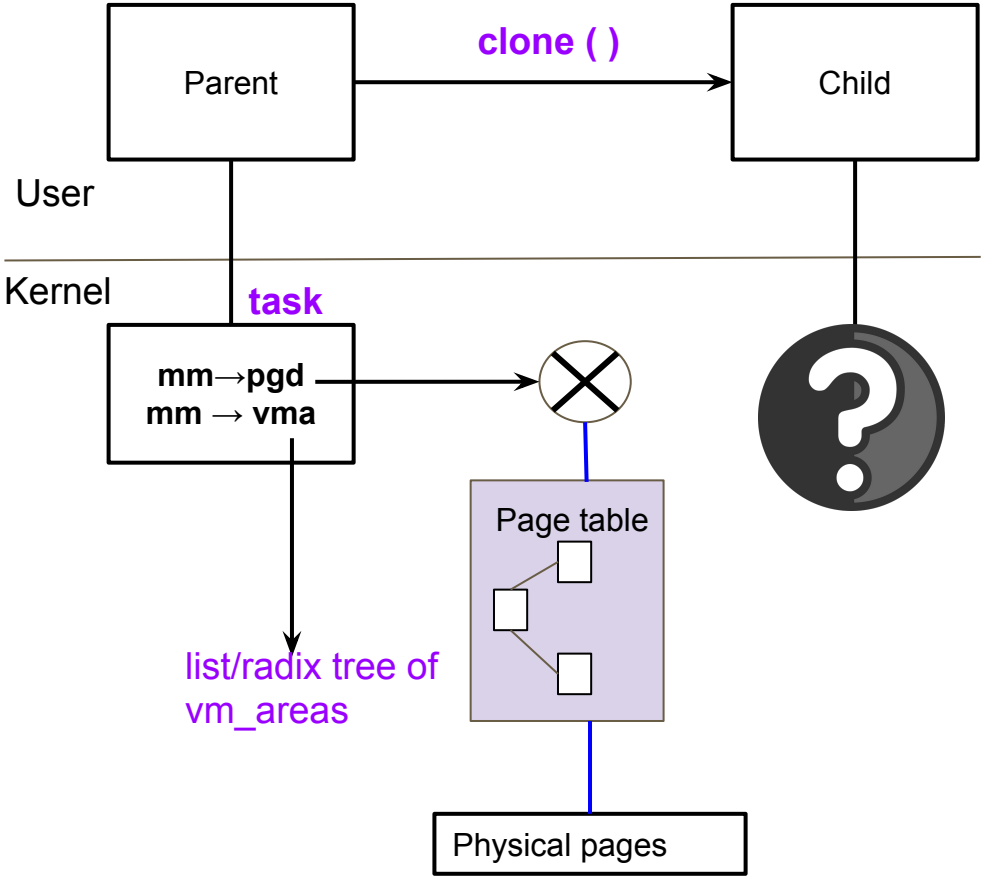
- ➔ Which mode child starts its execution? Why?
- ➔ What must change in kernel state of newly created process? Why?
- ➔ Any other changes
  - ◆ Depends on clone parameters
  - ◆ E.g., user stack pointer, return address etc.

Process state replication: x86_64



- ➔ Kernel stack for child process to be allocated and initialized
- ➔ User registers are appropriately modified
- ➔ At some point, child will be scheduled (in kernel)
  - ◆ But, what is the current stack frame?
  - ◆ Where is the (kernel) return address?
  - ◆ State restore on another CPU should be seamless

Kernel state replication: mm, vmas, page tables



- ➔ Which structures are replicated?
- ➔ Configurable levels of replication
  - ◆ Why?
  - ◆ What are the use cases?
- ➔ Copy-on-write (CoW)
- ➔ Relevant CLONE flags
  - ◆ CLONE_VM
  - ◆ CLONE_VFORK

Kernel state replication: mm, vmas, page tables

DEMO

Loading a new binary: exec( )

- ➔ execve ( ) system call
  - Path name of the executable → VFS layer calls required
  - Binary format, how would kernel know?
  - What would be the pt_regs modification? User instruction pointer?
- ➔ Original state cleanup
  - When?
  - Memory mappings
  - Open files?
  - Signal handlers?
  - Refer do_execveat_common ( ) in fs/exec.c

Exec: process state change

DEMO